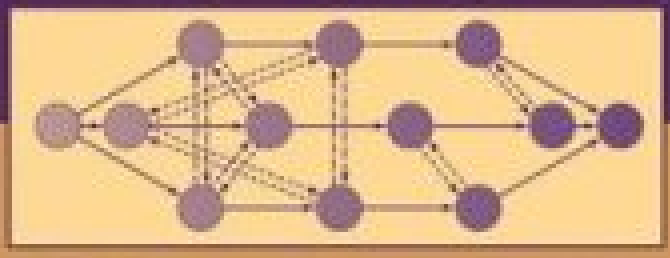
Michael L. Pinedo



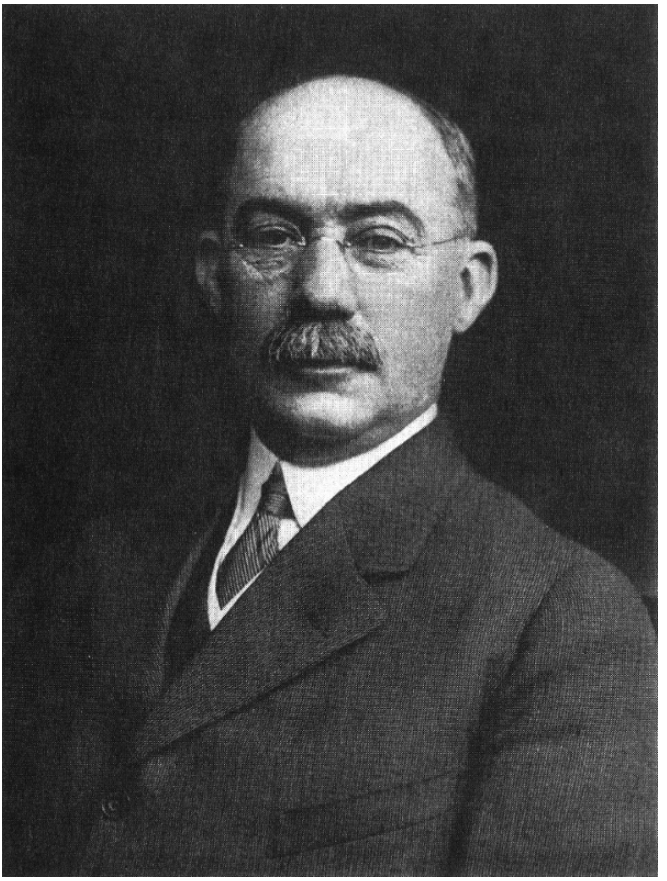
Scheduling

Theory, Algorithms, and Systems

Third Edition

Springer

Scheduling



HENRY LAURENCE GANTT
(1861–1919)

Henry Laurence Gantt was an industrial engineer and a disciple of Frederick W. Taylor. He developed his now famous charts during World War I to compare production schedules with their realizations. Gantt discussed the underlying principles in his paper "Efficiency and Democracy," which he presented at the annual meeting of the American Society of Mechanical Engineers in 1918. The Gantt charts currently in use are typically a simplification of the originals, both in purpose and in design.

Michael L. Pinedo

Scheduling

Theory, Algorithms, and Systems

Third Edition

INCLUDES
CD-ROM

Springer

Michael L. Pinedo
Stern School of Business
New York University
New York, NY
USA
mpinedo@stern.nyu.edu

ISBN: 978-0-387-78934-7 e-ISBN: 978-0-387-78935-4
DOI: 10.1007/978-0-387-78935-4

Library of Congress Control Number: 2008929515

Original edition published by Prentice Hall

Printed on acid-free paper

9 8 7 6 5 4 3 2 1

springer.com

To Paula,
Esti, Jaclyn, and Danielle,
Eddie and Jeffrey,
Franciniti and Morris.

Preface

Preface to the First Edition

Sequencing and scheduling is a form of decision-making that plays a crucial role in manufacturing and service industries. In the current competitive environment effective sequencing and scheduling has become a necessity for survival in the market-place. Companies have to meet shipping dates that have been committed to customers, as failure to do so may result in a significant loss of goodwill. They also have to schedule activities in such a way as to use the resources available in an efficient manner.

Scheduling began to be taken seriously in manufacturing at the beginning of this century with the work of Henry Gantt and other pioneers. However, it took many years for the first scheduling publications to appear in the industrial engineering and operations research literature. Some of the first publications appeared in Naval Research Logistics Quarterly in the early fifties and contained results by W.E. Smith, S.M. Johnson and J.R. Jackson. During the sixties a significant amount of work was done on dynamic programming and integer programming formulations of scheduling problems. After Richard Karp's famous paper on complexity theory, the research in the seventies focused mainly on the complexity hierarchy of scheduling problems. In the eighties several different directions were pursued in academia and industry with an increasing amount of attention paid to stochastic scheduling problems. Also, as personal computers started to permeate manufacturing facilities, scheduling systems were being developed for the generation of usable schedules in practice. This system design and development was, and is, being done by computer scientists, operations researchers and industrial engineers.

This book is the result of the development of courses in scheduling theory and applications at Columbia University. The book deals primarily with machine scheduling models. The first part covers deterministic models and the second part stochastic models. The third and final part deals with applications. In this last part scheduling problems in practice are discussed and the relevance of the theory to the real world is examined. From this examination it becomes

clear that the advances in scheduling theory have had only a limited impact on scheduling problems in practice. Hopefully there will be in a couple of years a second edition in which the applications part will be expanded, showing a stronger connection with the more theoretical parts of the text.

This book has benefited from careful reading by numerous people. Reha Uzsoy and Alan Scheller Wolf went through the manuscript with a fine tooth comb. Len Adler, Sid Browne, Xiuli Chao, Paul Glasserman, Chung-Yee Lee, Young-Hoon Lee, Joseph Leung, Elizabeth Leventhal, Rajesh Sah, Paul Shapiro, Jim Thompson, Barry Wolf, and the hundreds of students who had to take the (required) scheduling courses at Columbia provided many helpful comments which improved the manuscript.

The author is grateful to the National Science Foundation for its continued summer support, which made it possible to complete this project.

Michael L. Pinedo
New York, 1994.

Preface to the Second Edition

The book has been extended in a meaningful way. Five chapters have been added. In the deterministic part it is the treatment of the single machine, the job shop and the open shop that have been expanded considerably. In the stochastic part a completely new chapter focuses on single machine scheduling with release dates. This chapter has been included because of multiple requests from instructors who wanted to see a connection between stochastic scheduling and priority queues. This chapter establishes such a link. The applications part, Part III, has been expanded the most. Instead of a single chapter on general purpose procedures, there are now two chapters. The second chapter covers various techniques that are relatively new and that have started to receive a fair amount of attention over the last couple of years. There is also an additional chapter on the design and development of scheduling systems. This chapter focuses on rescheduling, learning mechanisms, and so on. The chapter with the examples of systems implementations is completely new. All systems described are of recent vintage. The last chapter contains a discussion on research topics that could become of interest in the next couple of years.

The book has a website:

`http://www.stern.nyu.edu/~mpinedo`

The intention is to keep the site as up-to-date as possible, including links to other sites that are potentially useful to instructors as well as students.

Many instructors who have used the book over the last couple of years have sent very useful comments and suggestions. Almost all of these comments have led to improvements in the manuscript.

Reha Uzsoy, as usual, went with a fine tooth comb through the manuscript. Salah Elmaghraby, John Fowler, Celia Glass, Chung-Yee Lee, Sigrid Knust,

Joseph Leung, Chris Potts, Levent Tuncel, Amy Ward, and Guochuan Zhang all made comments that led to substantial improvements.

A number of students, including Gabriel Adei, Yo Huh, Maher Lahmar, Sonia Leach, Michele Pfund, Edgar Possani, and Aysegul Toptal, have pointed out various errors in the original manuscript.

Without the help of a number of people from industry, it would not have been possible to produce a meaningful chapter on industrial implementations. Thanks are due to Heinrich Braun and Stephan Kreipl of SAP, Rama Akkiraju of IBM, Margie Bell of i2, Emanuela Rusconi and Fabio Tiozzo of Cybertec, and Paul Bender of SynQuest.

Michael L. Pinedo
New York, 2001.

Preface to the Third Edition

The basic structure of the book has not been changed in this new edition. The book still consists of three parts and a string of Appendixes. However, several chapters have been extended in a meaningful way, covering additional topics that have become recently of interest. Some of the new topics are more methodological, whereas others represent new classes of models.

The more methodological aspects that are receiving more attention include Polynomial Time Approximation Schemes (PTAS) and Constraint Programming. These extensions involve new material in the regular chapters as well as in the Appendixes. Since the field of online scheduling has received an enormous amount of attention in recent years, a section focusing on online scheduling has been added to the chapter on parallel machine scheduling.

Two new classes of models are introduced in the chapter on more advanced single machine scheduling, namely single machine scheduling with batch processing and single machine scheduling with job families.

Of course, as in any new edition, the chapter that describes implementations and applications had to be revamped and made up-to-date. That has happened here as well. Two new software systems have been introduced, namely a system that is currently being implemented at AMD (Advanced Micro Devices) and a generic system developed by Taylor Software.

For the first time, a CD-ROM has been included with the book. The CD-ROM contains various sets of power point slides, minicases provided by companies, the LEKIN Scheduling system, and two movies. The power point slides were developed by Julius Atlason (when he taught a scheduling course at the University of Michigan-Ann Arbor), Johann Hurink (from the University of Twente in Holland), Rakesh Nagi (from the State University of New York at Buffalo), Uwe Schwiegelshohn (from the University of Dortmund in Germany), Natalia Shakhlevich (from the University of Leeds in England).

A website will be maintained for this book at

`http://www.stern.nyu.edu/~mpinedo`

The intention is to keep this website as up-to-date as possible, including links to other sites that are potentially useful to instructors as well as to students.

A hardcopy of a solutions manual is available from the author for instructors who adopt the book. The solutions provided in this manual have been prepared by Clifford Stein (Columbia University), Julias Atlason (Michigan), Jim Geelen (Waterloo), Natalia Shakhlevich (Leeds), Levent Tuncel (Waterloo), and Martin Savelsbergh (Georgia Tech).

I am very grateful to a number of colleagues and students in academia who have gone over the new sections and have provided some very useful comments, namely Alessandro Agnetis (Siena), Ionut Aron (T.J. Watson Research Laboratories, IBM), Dirk Briskhorn (Kiel), John Fowler (Arizona), Jim Geelen (Waterloo), Johann Hurink (TU Twente, the Netherlands), Detlef Pabst (AMD), Gianluca de Pascale (Siena, Italy), Jacob Jan Paulus (TU Twente, the Netherlands), Jiri Sgall (Charles University, Prague), and Gerhard Woeginger (TU Eindhoven). Gerhard provided me with the chapters he wrote on Polynomial Time Approximation Schemes. His material has been incredibly useful.

Without the help of a number of people from industry, it would not have been possible to produce a meaningful chapter on industrial implementations. Thanks are due to Stephan Kreipl of SAP, Shekar Krishnaswamy and Peng Qu of AMD, and Robert MacDonald of Taylor Software.

The technical production of the book would not have been possible without the invalualable help from Adam Lewenberg (Stanford University) and Achi Dosanjh (Springer). Without the continued support of the National Science Foundation this book would never have been written.

Michael L. Pinedo
Spring 2008
New York

Contents

CD-ROM Contents

1. Slides from Academia

(a) University of Michigan, Ann Arbor (Julius Atlason)
(b) Technical University of Twente (Johann Hurink)
(c) State University of New York at Buffalo (Rakesh Nagi)
(d) University Dortmund (Uwe Schwiegelshohn)
(e) University of Leeds (Natalia Shakhlevich)

2. Scheduling Systems

(a) LEKIN (New York University - Michael L. Pinedo and Andrew Feldman)
(b) LiSA (Universitat Magdeburg - Heidemarie Braesel)
(c) TORSCHE (Czech Technical University - Michal Kutil)

3. Scheduling Case

(a) Scheduling in the Time-Shared Jet Business (Carnegie-Mellon University - Pinar Keskinocak and Sridhar Tayur)

4. Mini-Cases

(a) BCM Kosmetik (Taylor Software)
(b) Beaver Plastics (Taylor Software)
(c) Fountain Set (Holdings) Limited (Taylor Software)
(d) Lexmark (Taylor Software)
(e) Major Pharmaceuticals Manufacturer (Taylor Software)
(f) Major Printing Supplies Company (Taylor Software)
(g) Grammer (SAP)
(h) Mittal Steel Germany (SAP)
(i) mySAP Supply Chain Management (SAP)

5. Handouts

(a) University of Leeds (Natalia Shakhlevich)

6. Movies

(a) SAIGA - Scheduling at the Paris Airports (ILOG)
(b) Scheduling at United Airlines

Chapter 1

Introduction

1.1 The Role of Scheduling

Scheduling is a decision-making process that is used on a regular basis in many manufacturing and services industries. It deals with the allocation of resources to tasks over given time periods and its goal is to optimize one or more objectives.

The resources and tasks in an organization can take many different forms. The resources may be machines in a workshop, runways at an airport, crews at a construction site, processing units in a computing environment, and so on. The tasks may be operations in a production process, take-offs and landings at an airport, stages in a construction project, executions of computer programs, and so on. Each task may have a certain priority level, an earliest possible starting time and a due date. The objectives can also take many different forms. One objective may be the minimization of the completion time of the last task and another may be the minimization of the number of tasks completed after their respective due dates.

Scheduling, as a decision-making process, plays an important role in most manufacturing and production systems as well as in most information processing environments. It is also important in transportation and distribution settings and in other types of service industries. The following examples illustrate the role of scheduling in a number of real world environments.

Example 1.1.1 (A Paper Bag Factory)

Consider a factory that produces paper bags for cement, charcoal, dog food, and so on. The basic raw material for such an operation are rolls of paper. The production process consists of three stages: the printing of the logo, the gluing of the side of the bag, and the sewing of one end or both ends of the

M.L. Pinedo, *Scheduling*, DOI: 10.1007/978-0-387-78935-4_1,

Chapter 1

Introduction

1.1 The Role of Scheduling

Scheduling is a decision-making process that is used on a regular basis in many manufacturing and services industries. It deals with the allocation of resources to tasks over given time periods and its goal is to optimize one or more objectives.

The resources and tasks in an organization can take many different forms. The resources may be machines in a workshop, runways at an airport, crews at a construction site, processing units in a computing environment, and so on. The tasks may be operations in a production process, take-offs and landings at an airport, stages in a construction project, executions of computer programs, and so on. Each task may have a certain priority level, an earliest possible starting time and a due date. The objectives can also take many different forms. One objective may be the minimization of the completion time of the last task and another may be the minimization of the number of tasks completed after their respective due dates.

Scheduling, as a decision-making process, plays an important role in most manufacturing and production systems as well as in most information processing environments. It is also important in transportation and distribution settings and in other types of service industries. The following examples illustrate the role of scheduling in a number of real world environments.

Example 1.1.1 (A Paper Bag Factory)

Consider a factory that produces paper bags for cement, charcoal, dog food, and so on. The basic raw material for such an operation are rolls of paper. The production process consists of three stages: the printing of the logo, the gluing of the side of the bag, and the sewing of one end or both ends of the

M.L. Pinedo, *Scheduling*, DOI: 10.1007/978-0-387-78935-4_1,

bag. Each stage consists of a number of machines which are not necessarily identical. The machines at a stage may differ slightly in the speed at which they operate, the number of colors they can print or the size of bag they can produce. Each production order indicates a given quantity of a specific bag that has to be produced and shipped by a committed shipping date or due date. The processing times for the different operations are proportional to the size of the order, i.e., the number of bags ordered.

A late delivery implies a penalty in the form of loss of goodwill and the magnitude of the penalty depends on the importance of the order or the client and the tardiness of the delivery. One of the objectives of the scheduling system is to minimize the sum of these penalties.

When a machine is switched over from one type of bag to another a setup is required. The length of the setup time on the machine depends on the similarities between the two consecutive orders (the number of colors in common, the differences in bag size and so on). An important objective of the scheduling system is the minimization of the total time spent on setups. ||

Example 1.1.2 (A Semiconductor Manufacturing Facility)

Semiconductors are manufactured in highly specialized facilities. This is the case with memory chips as well as with microprocessors. The production process in these facilities usually consists of four phases: wafer fabrication, wafer probe, assembly or packaging, and final testing.

Wafer fabrication is technologically the most complex phase. Layers of metal and wafer material are built up in patterns on wafers of silicon or gallium arsenide to produce the circuitry. Each layer requires a number of operations, which typically include: (i) cleaning, (ii) oxidation, deposition and metallization, (iii) lithography, (iv) etching, (v) ion implantation, (vi) photoresist stripping, and (vii) inspection and measurement. Because it consists of various layers, each wafer has to undergo these operations several times. Thus, there is a significant amount of recirculation in the process. Wafers move through the facility in lots of 24. Some machines may require setups to prepare them for incoming jobs; the setup time often depends on the configurations of the lot just completed and the lot about to start.

The number of orders in the production process is often in the hundreds and each has its own release date and a committed shipping or due date. The scheduler's objective is to meet as many of the committed shipping dates as possible, while maximizing throughput. The latter goal is achieved by maximizing equipment utilization, especially of the bottleneck machines, requiring thus a minimization of idle times and setup times. ||

Example 1.1.3 (Gate Assignments at an Airport)

Consider an airline terminal at a major airport. There are dozens of gates and hundreds of planes arriving and departing each day. The gates are not all identical and neither are the planes. Some of the gates are in locations with a lot of space where large planes (widebodies) can be accommodated

easily. Other gates are in locations where it is difficult to bring in the planes; certain planes may actually have to be towed to their gates.

Planes arrive and depart according to a certain schedule. However, the schedule is subject to a certain amount of randomness, which may be weather related or caused by unforeseen events at other airports. During the time that a plane occupies a gate the arriving passengers have to be deplaned, the plane has to be serviced and the departing passengers have to be boarded. The scheduled departure time can be viewed as a due date and the airline's performance is measured accordingly. However, if it is known in advance that the plane cannot land at the next airport because of anticipated congestion at its scheduled arrival time, then the plane does not take off (such a policy is followed to conserve fuel). If a plane is not allowed to take off, operating policies usually prescribe that passengers remain in the terminal rather than on the plane. If boarding is postponed, a plane may remain at a gate for an extended period of time, thus preventing other planes from using that gate.

The scheduler has to assign planes to gates in such a way that the assignment is physically feasible while optimizing a number of objectives. This implies that the scheduler has to assign planes to suitable gates that are available at the respective arrival times. The objectives include minimization of work for airline personnel and minimization of airplane delays.

In this scenario the gates are the resources and the handling and servicing of the planes are the tasks. The arrival of a plane at a gate represents the starting time of a task and the departure represents its completion time. ||

Example 1.1.4 (Scheduling Tasks in a Central Processing Unit (CPU))

One of the functions of a multi-tasking computer operating system is to schedule the time that the CPU devotes to the different programs that have to be executed. The exact processing times are usually not known in advance. However, the distribution of these random processing times may be known in advance, including their means and their variances. In addition, each task usually has a certain priority level (the operating system typically allows operators and users to specify the priority level or weight of each task). In such case, the objective is to minimize the expected sum of the weighted completion times of all tasks.

To avoid the situation where relatively short tasks remain in the system for a long time waiting for much longer tasks that have a higher priority, the operating system "slices" each task into little pieces. The operating system then rotates these slices on the CPU so that in any given time interval, the CPU spends some amount of time on each task. This way, if by chance the processing time of one of the tasks is very short, the task will be able to leave the system relatively quickly.

An interruption of the processing of a task is often referred to as a *preemption*. It is clear that the optimal policy in such an environment makes heavy use of preemptions. ||

It may not be immediately clear what impact schedules may have on objectives of interest. Does it make sense to invest time and effort searching for a good schedule rather than just choosing a schedule at random? In practice, it often turns out that the choice of schedule *does* have a significant impact on the system's performance and that it *does* make sense to spend some time and effort searching for a suitable schedule.

Scheduling can be difficult from a technical as well as from an implementation point of view. The type of difficulties encountered on the technical side are similar to the difficulties encountered in other forms of combinatorial optimization and stochastic modeling. The difficulties on the implementation side are of a completely different kind. They may depend on the accuracy of the model used for the analysis of the actual scheduling problem and on the reliability of the input data that are needed.

1.2 The Scheduling Function in an Enterprise

The scheduling function in a production system or service organization must interact with many other functions. These interactions are system-dependent and may differ substantially from one situation to another. They often take place within an enterprise-wide information system.

A modern factory or service organization often has an elaborate information system in place that includes a central computer and database. Local area networks of personal computers, workstations and data entry terminals, which are connected to this central computer, may be used either to retrieve data from the database or to enter new data. The software controlling such an elaborate information system is typically referred to as an Enterprise Resource Planning (ERP) system. A number of software companies specialize in the development of such systems, including SAP, J.D. Edwards, and PeopleSoft. Such an ERP system plays the role of an information highway that traverses the enterprise with, at all organizational levels, links to decision support systems.

Scheduling is often done interactively via a decision support system that is installed on a personal computer or workstation linked to the ERP system. Terminals at key locations connected to the ERP system can give departments throughout the enterprise access to all current scheduling information. These departments, in turn, can provide the scheduling system with up-to-date information concerning the statuses of jobs and machines.

There are, of course, still environments where the communication between the scheduling function and other decision making entities occurs in meetings or through memos.

Scheduling in Manufacturing Consider the following generic manufacturing environment and the role of its scheduling. Orders that are released in a manufacturing setting have to be translated into jobs with associated due

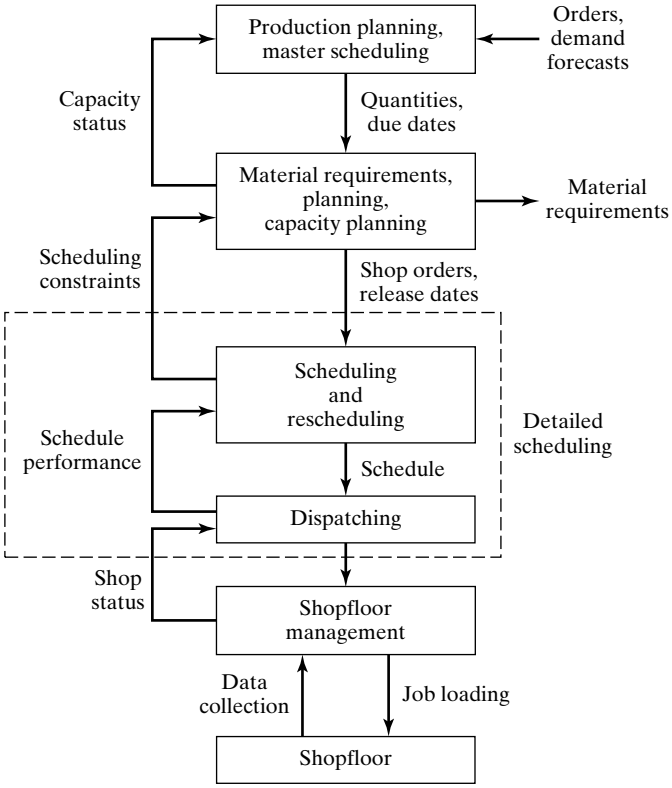


Fig. 1.1 Information flow diagram in a manufacturing system

dates. These jobs often have to be processed on the machines in a workcenter in a given order or sequence. The processing of jobs may sometimes be delayed if certain machines are busy and preemptions may occur when high priority jobs arrive at machines that are busy. Unforeseen events on the shop floor, such as machine breakdowns or longer-than-expected processing times, also have to be taken into account, since they may have a major impact on the schedules. In such an environment, the development of a detailed task schedule helps maintain efficiency and control of operations.

The shop floor is not the only part of the organization that impacts the scheduling process. It is also affected by the production planning process that handles medium- to long-term planning for the entire organization. This process attempts to optimize the firm's overall product mix and long-term resource allocation based on its inventory levels, demand forecasts and resource requirements. Decisions made at this higher planning level may impact the scheduling process directly. Figure 1.1 depicts a diagram of the information flow in a manufacturing system.

In a manufacturing environment, the scheduling function has to interact with other decision making functions. One popular system that is widely used is the Material Requirements Planning (MRP) system. After a schedule has been generated it is necessary that all raw materials and resources are available at the specified times. The ready dates of all jobs have to be determined jointly by the production planning/scheduling system and the MRP system.

MRP systems are normally fairly elaborate. Each job has a Bill Of Materials (BOM) itemizing the parts required for production. The MRP system keeps track of the inventory of each part. Furthermore, it determines the timing of the purchases of each one of the materials. In doing so, it uses techniques such as lot sizing and lot scheduling that are similar to those used in scheduling systems. There are many commercial MRP software packages available and, as a result, there are many manufacturing facilities with MRP systems. In the cases where the facility does not have a scheduling system, the MRP system may be used for production planning purposes. However, in complex settings it is not easy for an MRP system to do the detailed scheduling satisfactorily.

Scheduling in Services Describing a generic service organization and a typical scheduling system is not as easy as describing a generic manufacturing organization. The scheduling function in a service organization may face a variety of problems. It may have to deal with the reservation of resources, e.g., the assignment of planes to gates (see Example 1.1.3), or the reservation of meeting rooms or other facilities. The models used are at times somewhat different from those used in manufacturing settings. Scheduling in a service environment must be coordinated with other decision making functions, usually within elaborate information systems, much in the same way as the scheduling function in a manufacturing setting. These information systems usually rely on extensive databases that contain all the relevant information with regard to availability of resources and (potential) customers. The scheduling system interacts often with forecasting and yield management modules. Figure 1.2 depicts the information flow in a service organization such as a car rental agency. In contrast to manufacturing settings, there is usually no MRP system in a service environment.

1.3 Outline of the Book

This book focuses on both the theory and the applications of scheduling. The theoretical side deals with the detailed sequencing and scheduling of jobs. Given a collection of jobs requiring processing in a certain machine environment, the problem is to sequence these jobs, subject to given constraints, in such a way that one or more performance criteria are optimized. The scheduler may have to deal with various forms of uncertainties, such as random job processing times, machines subject to breakdowns, rush orders, and so on.

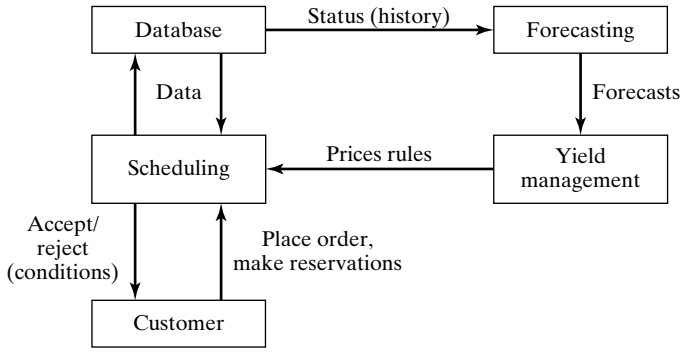


Fig. 1.2 Information flow diagram in a service system

Thousands of scheduling problems and models have been studied and analyzed in the past. Obviously, only a limited number are considered in this book; the selection is based on the insight they provide, the methodology needed for their analysis and their importance in applications.

Although the applications driving the models in this book come mainly from manufacturing and production environments, it is clear from the examples in Section 1.1 that scheduling plays a role in a wide variety of situations. The models and concepts considered in this book are applicable in other settings as well.

This book is divided into three parts. Part I (Chapters 2 to 8) deals with deterministic scheduling models. In these chapters it is assumed that there are a finite number of jobs that have to be scheduled with one or more objectives to be minimized. Emphasis is placed on the analysis of relatively simple priority or dispatching rules. Chapter 2 discusses the notation and gives an overview of the models that are considered in the subsequent chapters. Chapters 3 to 8 consider the various machine environments. Chapters 3 and 4 deal with the single machine, Chapter 5 with machines in parallel, Chapter 6 with machines in series and Chapter 7 with the more complicated job shop models. Chapter 8 focuses on open shops in which there are no restrictions on the routings of the jobs in the shop.

Part II (Chapters 9 to 13) deals with stochastic scheduling models. These chapters, in most cases, also assume that a given (finite) number of jobs have to be scheduled. The job data, such as processing times, release dates and due dates may not be exactly known in advance; only their distributions are known in advance. The actual processing times, release dates and due dates become known only at the *completion* of the processing or at the actual occurrence of the release or due date. In these models a single objective has to be minimized, usually in expectation. Again, an emphasis is placed on the analysis of relatively simple priority or dispatching rules. Chapter 9 contains preliminary material. Chapter 10 covers the single machine environment. Chapter 11 also covers the

single machine, but in this chapter it is assumed that the jobs are released at different points in time. This chapter establishes the relationship between stochastic scheduling and the theory of priority queues. Chapter 12 focuses on machines in parallel and Chapter 13 describes the more complicated flow shop, job shop, and open shop models.

Part III (Chapters 14 to 20) deals with applications and implementation issues. Algorithms are described for a number of real world scheduling problems. Design issues for scheduling systems are discussed and some examples of scheduling systems are given. Chapters 14 and 15 describe various general purpose procedures that have proven to be useful in industrial scheduling systems. Chapter 16 describes a number of real world scheduling problems and how they have been dealt with in practice. Chapter 17 focuses on the basic issues concerning the design, the development and the implementation of scheduling systems, and Chapter 18 discusses the more advanced concepts in the design and implementation of scheduling systems. Chapter 19 gives some examples of actual implementations. Chapter 20 ponders on what lies ahead in scheduling.

Appendices A, B, C, and D present short overviews of some of the basic methodologies, namely mathematical programming, dynamic programming, constraint programming, and complexity theory. Appendix E contains a complexity classification of the deterministic scheduling problems, while Appendix F presents an overview of the stochastic scheduling problems. Appendix G lists a number of scheduling systems that have been developed in industry and academia. Appendix H provides some guidelines for using the LEKIN scheduling system. The LEKIN system is included on the CD-ROM that comes with the book.

This book is designed for either a masters level course or a beginning PhD level course in Production Scheduling. When used for a senior level course, the topics most likely covered are from Parts I and III. Such a course can be given without getting into complexity theory: one can go through the chapters of Part I skipping all complexity proofs without loss of continuity. A masters level course may cover topics from Part II as well. Even though all three parts are fairly self-contained, it is helpful to go through Chapter 2 before venturing into Part II.

Prerequisite knowledge for this book is an elementary course in Operations Research on the level of Hillier and Lieberman's *Introduction to Operations Research* and an elementary course in stochastic processes on the level of Ross's *Introduction to Probability Models*.

Comments and References

During the last four decades many books have appeared that focus on sequencing and scheduling. These books range from the elementary to the more advanced.

A volume edited by Muth and Thompson (1963) contains a collection of papers focusing primarily on computational aspects of scheduling. One of the better known textbooks is the one by Conway, Maxwell and Miller (1967) (which, even though slightly out of date, is still very interesting); this book also deals with some of the stochastic aspects and with priority queues. A more recent text by Baker (1974) gives an excellent overview of the many aspects of deterministic scheduling. However, this book does not deal with computational complexity issues since it appeared just before research in computational complexity started to become popular. The book by Coffman (1976) is a compendium of papers on deterministic scheduling; it does cover computational complexity. An introductory textbook by French (1982) covers most of the techniques that are used in deterministic scheduling. The proceedings of a NATO workshop, edited by Dempster, Lenstra and Rinnooy Kan (1982), contains a number of advanced papers on deterministic as well as on stochastic scheduling. The relatively advanced book by Blazewicz, Cellary, Slowinski and Weglarz (1986) focuses mainly on resource constraints and multi-objective deterministic scheduling. The book by Blazewicz, Ecker, Schmidt and Weglarz (1993) is somewhat advanced and deals primarily with the computational aspects of deterministic scheduling models and their applications to manufacturing. The more applied text by Morton and Pentico (1993) presents a detailed analysis of a large number of scheduling heuristics that are useful for practitioners. The monograph by Dauzère-Pérès and Lasserre (1994) focuses primarily on job shop scheduling. A collection of papers, edited by Zweben and Fox (1994), describes a number of scheduling systems and their actual implementations. Another collection of papers, edited by Brown and Scherer (1995) also describe various scheduling systems and their implementation. The proceedings of a workshop edited by Chrétienne, Coffman, Lenstra and Liu (1995) contain a set of interesting papers concerning primarily deterministic scheduling. The textbook by Baker (1995) is very useful for an introductory course in sequencing and scheduling. Brucker (1995) presents, in the first edition of his book, a very detailed algorithmic analysis of the many deterministic scheduling models. Parker (1995) gives a similar overview and tends to focus on problems with precedence constraints or other graph-theoretic issues. Sule (1996) is a more applied text with a discussion of some interesting real world problems. Blazewicz, Ecker, Pesch, Schmidt and Weglarz (1996) is an extended edition of the earlier work by Blazewicz, Ecker, Schmidt and Weglarz (1993). The monograph by Ovacik and Uzsoy (1997) is entirely dedicated to decomposition methods for complex job shops. The two volumes edited by Lee and Lei (1997) contain many interesting theoretical as well as applied papers. The book by Pinedo and Chao (1999) is more application oriented and describes a number of different scheduling models for problems arising in manufacturing and in services. The monograph by Baptiste, LePape and Nuijten (2001) covers applications of constraint programming techniques to job shop scheduling. The volume edited by Nareyek (2001) contains papers on local search applied to job shop scheduling. T'kindt and Billaut (2002, 2006) provide an excellent treatise of multicriteria scheduling. Brucker (2004) is an expanded version of the orig-

inal first edition that appeared in 1995. The *Handbook of Scheduling*, edited by Leung (2004), contains numerous papers on all aspects of scheduling. The text by Pinedo (2005) is a modified and extended version of the earlier one by Pinedo and Chao (1999). Dawande, Geismar, Sethi and Sriskandarajah (2007) focus in their more advanced text on the scheduling of robotic cells; these manufacturing settings are, in a sense, extensions of flow shops.

Besides these books a number of survey articles have appeared, each one with a large number of references. The articles by Graves (1981) and Rodammer and White (1988) review production scheduling. Atabakhsh (1991) presents a survey of constraint based scheduling systems that use artificial intelligence techniques and Noronha and Sarma (1991) review knowledge-based approaches for scheduling problems. Smith (1992) focuses in his survey on the development and implementation of scheduling systems. Lawler, Lenstra, Rinnooy Kan and Shmoys (1993) give a detailed overview of deterministic sequencing and scheduling and Righter (1994) does the same for stochastic scheduling. Queyranne and Schulz (1994) provide an in depth analysis of polyhedral approaches to non-preemptive machine scheduling problems. Chen, Potts and Woeginger (1998) review computational complexity, algorithms and approximability in deterministic scheduling. Sgall (1998) and Pruhs, Sgall and Torng (2004) present surveys of an area within deterministic scheduling referred to as online scheduling. Even though online scheduling is often considered a part of deterministic scheduling, the theorems obtained may at times provide interesting new insights into certain stochastic scheduling models.

Part I

Deterministic Models

Chapter 2

Deterministic Models: Preliminaries

Over the last fifty years a considerable amount of research effort has been focused on deterministic scheduling. The number and variety of models considered is astounding. During this time a notation has evolved that succinctly captures the structure of many (but for sure not all) deterministic models that have been considered in the literature.

The first section in this chapter presents an adapted version of this notation. The second section contains a number of examples and describes some of the shortcomings of the framework and notation. The third section describes several classes of schedules. A class of schedules is typically characterized by the freedom the scheduler has in the decision-making process. The last section discusses the complexity of the scheduling problems introduced in the first section. This last section can be used, together with Appendixes D and E, to classify scheduling problems according to their complexity.

2.1 Framework and Notation

In all the scheduling problems considered the number of jobs and the number of machines are assumed to be finite. The number of jobs is denoted by n and the number of machines by m. Usually, the subscript j refers to a job while the subscript i refers to a machine. If a job requires a number of processing steps or operations, then the pair (i, j) refers to the processing step or operation of job j on machine i. The following pieces of data are associated with job j.

M.L. Pinedo, *Scheduling*, DOI: 10.1007/978-0-387-78935-4_2,

Processing time (p_{ij}) The p_{ij} represents the processing time of job j on machine i. The subscript i is omitted if the processing time of job j does not depend on the machine or if job j is only to be processed on one given machine.

Release date (r_j) The release date r_j of job j may also be referred to as the ready date. It is the time the job arrives at the system, i.e., the earliest time at which job j can start its processing.

Due date (d_j) The due date d_j of job j represents the committed shipping or completion date (i.e., the date the job is promised to the customer). Completion of a job after its due date *is* allowed, but then a penalty is incurred. When a due date *must* be met it is referred to as a *deadline* and denoted by $\bar{d}_j$.

Weight (w_j) The weight w_j of job j is basically a priority factor, denoting the importance of job j relative to the other jobs in the system. For example, this weight may represent the actual cost of keeping the job in the system. This cost could be a holding or inventory cost; it also could represent the amount of value already added to the job.

A scheduling problem is described by a triplet $\alpha \mid \beta \mid \gamma$. The α field describes the machine environment and contains just one entry. The β field provides details of processing characteristics and constraints and may contain no entry at all, a single entry, or multiple entries. The γ field describes the objective to be minimized and often contains a single entry.

The possible machine environments specified in the α field are:

Single machine (1) The case of a single machine is the simplest of all possible machine environments and is a special case of all other more complicated machine environments.

Identical machines in parallel (Pm) There are m identical machines in parallel. Job j requires a single operation and may be processed on any one of the m machines or on any one that belongs to a given subset. If job j cannot be processed on just any machine, but only on any one belonging to a specific subset M_j, then the entry M_j appears in the β field.

Machines in parallel with different speeds (Qm) There are m machines in parallel with different speeds. The speed of machine i is denoted by v_i. The time p_{ij} that job j spends on machine i is equal to p_j/v_i (assuming job j receives all its processing from machine i). This environment is referred to as *uniform* machines. If all machines have the same speed, i.e., $v_i = 1$ for all i and $p_{ij} = p_j$, then the environment is identical to the previous one.

Unrelated machines in parallel (Rm) This environment is a further generalization of the previous one. There are m different machines in parallel. Machine i can process job j at speed v_{ij}. The time p_{ij} that job j spends on machine i is equal to p_j/v_{ij} (again assuming job j receives all its processing from machine i). If the speeds of the machines are independent of the jobs, i.e., $v_{ij} = v_i$ for all i and j, then the environment is identical to the previous one.

Flow shop (Fm) There are m machines in series. Each job has to be processed on each one of the m machines. All jobs have to follow the same route, i.e., they have to be processed first on machine 1, then on machine 2, and so on. After completion on one machine a job joins the queue at the next machine. Usually, all queues are assumed to operate under the *First In First Out (FIFO)* discipline, that is, a job cannot "pass" another while waiting in a queue. If the *FIFO* discipline is in effect the flow shop is referred to as a *permutation* flow shop and the β field includes the entry *prmu*.

Flexible flow shop (FFc) A flexible flow shop is a generalization of the flow shop and the parallel machine environments. Instead of m machines in series there are c stages in series with at each stage a number of identical machines in parallel. Each job has to be processed first at stage 1, then at stage 2, and so on. A stage functions as a bank of parallel machines; at each stage job j requires processing on only one machine and any machine can do. The queues between the various stages may or may not operate according to the *First Come First Served (FCFS)* discipline. (Flexible flow shops have in the literature at times also been referred to as hybrid flow shops and as multi-processor flow shops.)

Job shop (Jm) In a job shop with m machines each job has its own predetermined route to follow. A distinction is made between job shops in which each job visits each machine at most once and job shops in which a job may visit each machine more than once. In the latter case the β-field contains the entry *rcrc* for *recirculation*.

Flexible job shop (FJc) A flexible job shop is a generalization of the job shop and the parallel machine environments. Instead of m machines in series there are c work centers with at each work center a number of identical machines in parallel. Each job has its own route to follow through the shop; job j requires processing at each work center on only one machine and any machine can do. If a job on its route through the shop may visit a work center more than once, then the β-field contains the entry *rcrc* for recirculation.

Open shop (Om) There are m machines. Each job has to be processed again on each one of the m machines. However, some of these processing times may be zero. There are no restrictions with regard to the routing of each job through the machine environment. The scheduler is allowed to determine a route for each job and different jobs may have different routes.

The processing restrictions and constraints specified in the β field may include multiple entries. Possible entries in the β field are:

Release dates (r_j) If this symbol appears in the β field, then job j cannot start its processing before its release date r_j. If r_j does not appear in the β field, the processing of job j may start at any time. In contrast to release dates, due dates are not specified in this field. The type of objective function gives sufficient indication whether or not there are due dates.

Preemptions (*prmp*) Preemptions imply that it is not necessary to keep a job on a machine, once started, until its completion. The scheduler is allowed to interrupt the processing of a job (preempt) at any point in time and put a different job on the machine instead. The amount of processing a preempted job already has received is not lost. When a preempted job is afterwards put back on the machine (or on another machine in the case of parallel machines), it only needs the machine for its *remaining* processing time. When preemptions are allowed *prmp* is included in the β field; when *prmp* is not included, preemptions are not allowed.

Precedence constraints (*prec*) Precedence constraints may appear in a single machine or in a parallel machine environment, requiring that one or more jobs may have to be completed before another job is allowed to start its processing. There are several special forms of precedence constraints: if each job has at most one predecessor and at most one successor, the constraints are referred to as *chains*. If each job has at most one successor, the constraints are referred to as an *intree*. If each job has at most one predecessor the constraints are referred to as an *outtree*. If no *prec* appears in the β field, the jobs are not subject to precedence constraints.

Sequence dependent setup times (s_{jk}) The s_{jk} represents the sequence dependent setup time that is incurred between the processing of jobs j and k; s_{0k} denotes the setup time for job k if job k is first in the sequence and s_{j0} the clean-up time after job j if job j is last in the sequence (of course, s_{0k} and s_{j0} may be zero). If the setup time between jobs j and k depends on the machine, then the subscript i is included, i.e., s_{ijk}. If no s_{jk} appears in the β field, all setup times are assumed to be 0 or sequence independent, in which case they are simply included in the processing times.

Job families (*fmls*) The n jobs belong in this case to F different job families. Jobs from the same family may have different processing times, but they can be processed on a machine one after another without requiring any setup in between. However, if the machine switches over from one family to another, say from family g to family h, then a setup is required. If this setup time depends on both families g and h and is sequence dependent, then it is denoted by s_{gh}. If this setup time depends only on the family about to start, i.e., family h, then it is denoted by s_h. If it does not depend on either family, it is denoted by s.

Batch processing (*batch*(b)) A machine may be able to process a number of jobs, say b, simultaneously; that is, it can process a batch of up to b jobs at the same time. The processing times of the jobs in a batch may not be all the same and the entire batch is finished only when the last job of the batch has been completed, implying that the completion time of the entire batch is determined by the job with the longest processing time. If $b = 1$, then the problem reduces to a conventional scheduling environment. Another special case that is of interest is $b = \infty$, i.e., there is no limit on the number of jobs the machine can handle at any time.

Breakdowns (*brkdwn*) Machine breakdowns imply that a machine may not be continuously available. The periods that a machine is not available are, in this part of the book, assumed to be fixed (e.g., due to shifts or scheduled maintenance). If there are a number of identical machines in parallel, the number of machines available at any point in time is a function of time, i.e., $m(t)$. Machine breakdowns are at times also referred to as machine availability constraints.

Machine eligibility restrictions (M_j) The M_j symbol may appear in the β field when the machine environment is m machines in parallel (Pm). When the M_j is present, not all m machines are capable of processing job j. Set M_j denotes the set of machines that can process job j. If the β field does not contain M_j, job j may be processed on any one of the m machines.

Permutation (*prmu*) A constraint that may appear in the flow shop environment is that the queues in front of each machine operate according to the *First In First Out (FIFO)* discipline. This implies that the order (or *permutation*) in which the jobs go through the first machine is maintained throughout the system.

Blocking (*block*) Blocking is a phenomenon that may occur in flow shops. If a flow shop has a limited buffer in between two successive machines, then it may happen that when the buffer is full the upstream machine is not allowed to release a completed job. Blocking implies that the completed job has to remain on the upstream machine preventing (i.e., blocking) that machine from working on the next job. The most common occurrence of blocking that is considered in this book is the case with zero buffers in between any two successive machines. In this case a job that has completed its processing on a given machine cannot leave the machine if the preceding job has not yet completed its processing on the next machine; thus, the blocked job also prevents (or blocks) the next job from starting its processing on the given machine. In the models with blocking that are considered in subsequent chapters the assumption is made that the machines operate according to *FIFO*. That is, *block* implies *prmu*.

No-wait (*nwt*) The *no-wait* requirement is another phenomenon that may occur in flow shops. Jobs are not allowed to wait between two successive machines. This implies that the starting time of a job at the first machine has to be delayed to ensure that the job can go through the flow shop without having to wait for any machine. An example of such an operation is a steel rolling mill in which a slab of steel is not allowed to wait as it would cool off during a wait. It is clear that under *no-wait* the machines also operate according to the *FIFO* discipline.

Recirculation (*rcrc*) Recirculation may occur in a job shop or flexible job shop when a job may visit a machine or work center more than once.

Any other entry that may appear in the β field is self explanatory. For example, $p_j = p$ implies that all processing times are equal and $d_j = d$ implies that all due dates are equal. As stated before, due dates, in contrast to release dates, are usually not explicitly specified in this field; the type of objective function gives sufficient indication whether or not the jobs have due dates.

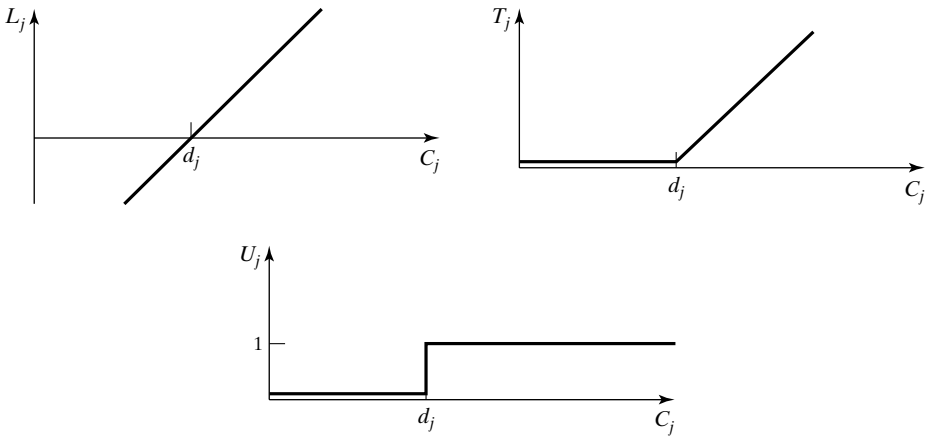


Fig. 2.1 Due date related penalty functions

The objective to be minimized is always a function of the completion times of the jobs, which, of course, depend on the schedule. The completion time of the operation of job j on machine i is denoted by C_{ij}. The time job j exits the system (that is, its completion time on the last machine on which it requires processing) is denoted by C_j. The objective may also be a function of the due dates. The *lateness* of job j is defined as

$$L_j = C_j - d_j,$$

which is positive when job j is completed late and negative when it is completed early. The *tardiness* of job j is defined as

$$T_j = \max(C_j - d_j, 0) = \max(L_j, 0).$$

The difference between the tardiness and the lateness lies in the fact that the tardiness never is negative. The *unit penalty* of job j is defined as

$$U_j = \begin{cases} 1 \text{ if } C_j > d_j \\ 0 \text{ otherwise} \end{cases}$$

The lateness, the tardiness and the unit penalty are the three basic due date related penalty functions considered in this book. The shape of these functions are depicted in Figure 2.1.

Examples of possible objective functions to be minimized are:

Makespan ($C_{\max}$) The makespan, defined as $\max(C_1, \ldots, C_n)$, is equivalent to the completion time of the last job to leave the system. A minimum makespan usually implies a good utilization of the machine(s).

Maximum Lateness ($L_{\max}$) The maximum lateness, $L_{\max}$, is defined as $\max(L_1, \ldots, L_n)$. It measures the worst violation of the due dates.

Total weighted completion time ($\sum w_j C_j$) The sum of the weighted completion times of the n jobs gives an indication of the total holding or inventory costs incurred by the schedule. The sum of the completion times is in the literature often referred to as the flow time. The total weighted completion time is then referred to as the weighted flow time.

Discounted total weighted completion time ($\sum w_j(1 - e^{-rC_j})$) This is a more general cost function than the previous one, where costs are discounted at a rate of r, $0 < r < 1$, per unit time. That is, if job j is not completed by time t an additional cost $w_j r e^{-rt} dt$ is incurred over the period $[t, t + dt]$. If job j is completed at time t the total cost incurred over the period $[\,0, t\,]$ is $w_j(1 - e^{-rt})$. The value of r is usually close to 0, say 0.1 or 10 %.

Total weighted tardiness ($\sum w_j T_j$) This is also a more general cost function than the total weighted completion time.

Weighted number of tardy jobs ($\sum w_j U_j$) The weighted number of tardy jobs is not only a measure of academic interest, it is often an objective in practice as it is a measure that can be recorded very easily.

All the objective functions above are so-called *regular* performance measures. A regular performance measure is a function that is *nondecreasing* in $C_1, \ldots, C_n$. Recently researchers have begun to study objective functions that are not regular. For example, when job j has a due date d_j, it may be subject to an earliness penalty, where the *earliness* of job j is defined as

$$E_j = \max(d_j - C_j, 0).$$

This earliness penalty is *nonincreasing* in C_j. An objective such as the total earliness plus the total tardiness, i.e.,

$$\sum_{j=1}^{n} E_j + \sum_{j=1}^{n} T_j,$$

is therefore not regular. A more general objective that is not regular is the total weighted earliness plus the total weighted tardiness, i.e.,

$$\sum_{j=1}^{n} w'_j E_j + \sum_{j=1}^{n} w''_j T_j.$$

The weight associated with the earliness of job j (w'_j) may be different from the weight associated with the tardiness of job j (w''_j).

2.2 Examples

The following examples illustrate the notation:

Example 2.2.1 (A Flexible Flow Shop)

$FFc \mid r_j \mid \sum w_j T_j$ denotes a flexible flow shop. The jobs have release dates and due dates and the objective is the minimization of the total weighted tardiness. Example 1.1.1 in Section 1.1 (the paper bag factory) can be modeled as such. Actually, the problem described in Section 1.1 has some additional characteristics including sequence dependent setup times at each of the three stages. In addition, the processing time of job j on machine i has a special structure: it depends on the number of bags and on the speed of the machine. ||

Example 2.2.2 (A Flexible Job Shop)

$FJc \mid r_j, s_{ijk}, rcrc \mid \sum w_j T_j$ refers to a flexible job shop with c work centers. The jobs have different release dates and are subject to sequence dependent setup times that are machine dependent. There is recirculation, so a job may visit a work center more than once. The objective is to minimize the total weighted tardiness. It is clear that this problem is a more general problem than the one described in the previous example. Example 1.1.2 in Section 1.1 (the semiconductor manufacturing facility) can be modeled as such. ||

Example 2.2.3 (A Parallel Machine Environment)

$Pm \mid r_j, M_j \mid \sum w_j T_j$ denotes a system with m machines in parallel. Job j arrives at release date r_j and has to leave by the due date d_j. Job j may be processed only on one of the machines belonging to the subset M_j. If job j is not completed in time a penalty $w_j T_j$ is incurred. This model can be used for the gate assignment problem described in Example 1.1.3. ||

Example 2.2.4 (A Single Machine Environment)

$1 \mid r_j, prmp \mid \sum w_j C_j$ denotes a single machine system with job j entering the system at its release date r_j. Preemptions are allowed. The objective to be minimized is the sum of the weighted completion times. This model can be used to study the deterministic counterpart of the problem described in Example 1.1.4. ||

Example 2.2.5 (Sequence Dependent Setup Times)

$1 \mid s_{jk} \mid C_{\max}$ denotes a single machine system with n jobs subject to sequence dependent setup times, where the objective is to minimize the makespan. It is well-known that this problem is equivalent to the so-called *Travelling Salesman Problem (TSP)*, where a salesman has to tour n cities in such a way that the total distance traveled is minimized (see Appendix D for a formal definition of the TSP). ||

Example 2.2.6 (A Project)

$P\infty \mid prec \mid C_{\max}$ denotes a scheduling problem with n jobs subject to precedence constraints and an unlimited number of machines (or resources) in parallel. The total time of the entire project has to be minimized. This type of problem is very common in project planning in the construction industry and has lead to techniques such as the *Critical Path Method (CPM)* and the *Project Evaluation and Review Technique (PERT).* ||

Example 2.2.7 (A Flow Shop)

$Fm \mid p_{ij} = p_j \mid \sum w_j C_j$ denotes a *proportionate* flow shop environment with m machines in series; the processing times of job j on all m machines are identical and equal to p_j (hence the term proportionate). The objective is to find the order in which the n jobs go through the system so that the sum of the weighted completion times is minimized. ||

Example 2.2.8 (A Job Shop)

$Jm \mid\mid C_{\max}$ denotes a job shop problem with m machines. There is no recirculation, so a job visits each machine at most once. The objective is to minimize the makespan. This problem is considered a classic in the scheduling literature and has received an enormous amount of attention. ||

Of course, there are many scheduling models that are not captured by this framework. One can define, for example, a more general flexible job shop in which each work center consists of a number of unrelated machines in parallel. When a job on its route through the system arrives at a bank of unrelated machines, it may be processed on any one of the machines, but its processing time now depends on the machine on which it is processed.

One can also define a model that is a mixture of a job shop and an open shop. The routes of some jobs are fixed, while the routes of other jobs are (partially) open.

The framework described in Section 2.1 has been designed primarily for models with a single objective. Most research in the past has concentrated on models with a single objective. Recently, researchers have begun studying models with multiple objectives as well.

Various other scheduling features, that are not mentioned here, have been studied and analyzed in the literature. Such features include periodic or cyclic scheduling, personnel scheduling, and resource constrained scheduling.

2.3 Classes of Schedules

In scheduling terminology a distinction is often made between a *sequence*, a *schedule* and a *scheduling policy*. A sequence usually corresponds to a permutation of the n jobs or the order in which jobs are to be processed on a given

machine. A schedule usually refers to an allocation of jobs within a more complicated setting of machines, allowing possibly for preemptions of jobs by other jobs that are released at later points in time. The concept of a scheduling policy is often used in stochastic settings: a policy prescribes an appropriate action for any one of the states the system may be in. In deterministic models usually only sequences or schedules are of importance.

Assumptions have to be made with regard to what the scheduler may and may not do when he generates a schedule. For example, it may be the case that a schedule may not have any *unforced idleness* on any machine. This class of schedules can be defined as follows.

Definition 2.3.1 (Non-Delay Schedule). *A feasible schedule is called non-delay if no machine is kept idle while an operation is waiting for processing.*

Requiring a schedule to be non-delay is equivalent to prohibiting *unforced idleness*. For many models, including those that allow preemptions and have regular objective functions, there are optimal schedules that are non-delay. For many models considered in this part of the book the goal is to find an optimal schedule that is non-delay. However, there *are* models where it may be advantageous to have periods of unforced idleness.

A smaller class of schedules, within the class of all non-delay schedules, is the class of nonpreemptive non-delay schedules. Nonpreemptive non-delay schedules may lead to some interesting and unexpected anomalies.

Example 2.3.2 (A Scheduling Anomaly)

Consider an instance of $P2 \mid prec \mid C_{\max}$ with 10 jobs and the following processing times.

jobs	1	2	3	4	5	6	7	8	9	10
p_j	8	7	7	2	3	2	2	8	8	15

The jobs are subject to the precedence constraints depicted in Figure 2.2. The makespan of the non-delay schedule depicted in Figure 2.3.a is 31 and the schedule is clearly optimal.

One would expect that, if each one of the ten processing times is reduced by one time unit, the makespan would be less than 31. However, requiring the schedule to be non-delay results in the schedule depicted in Figure 2.3.b with a makespan of 32.

Suppose that an additional machine is made available and that there are now three machines instead of two. One would again expect the makespan with the original set of processing times to be less than 31. Again, the non-delay requirement has an unexpected effect: the makespan is now 36. ||

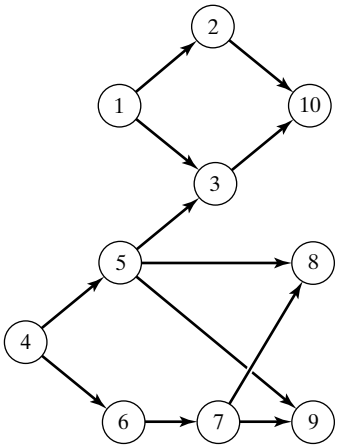


Fig. 2.2 Precedence constraints graph for Example 2.3.2.

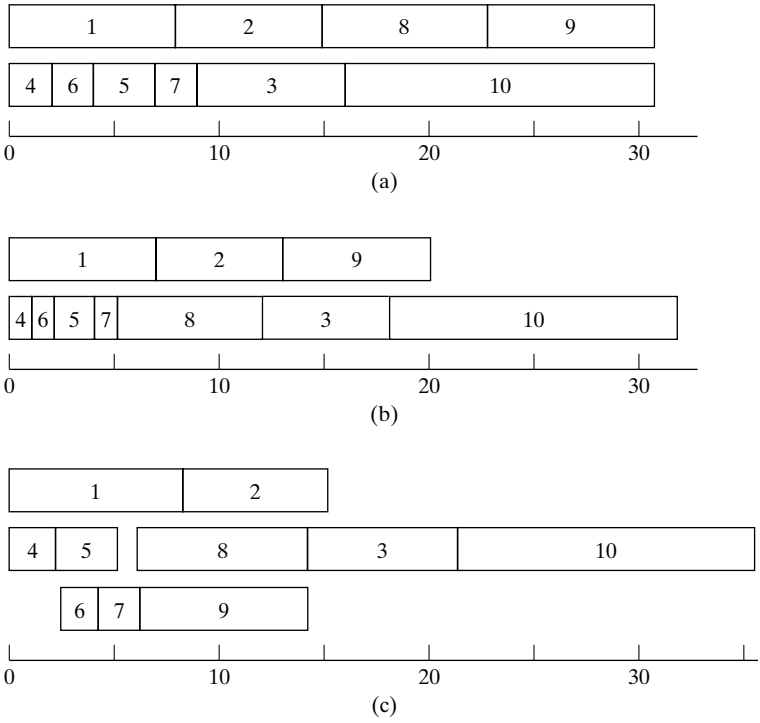


Fig. 2.3 Gantt charts of nondelay schedules: (a) Original schedule (b) Processing times one unit shorter (c) Original processing times and three machines

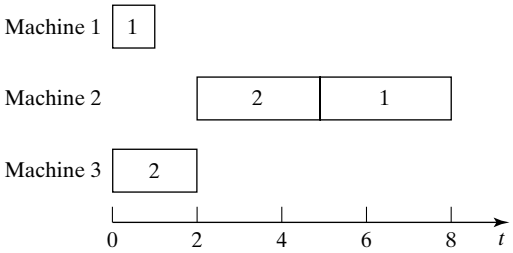


Fig. 2.4 An active schedule that is not nondelay.

Some heuristic procedures and algorithms for job shops are based on the construction of nonpreemptive schedules with certain special properties. Two classes of nonpreemptive schedules are of importance for certain algorithmic procedures for job shops.

Definition 2.3.3 (Active Schedule). *A feasible nonpreemptive schedule is called active if it is not possible to construct another schedule, through changes in the order of processing on the machines, with at least one operation finishing earlier and no operation finishing later.*

In other words, a schedule is active if no operation can be put into an empty hole earlier in the schedule while preserving feasibility. A nonpreemptive nondelay schedule has to be active but the reverse is not necessarily true. The following example describes a schedule that is active but not non-delay.

Example 2.3.4 (An Active Schedule)

Consider a job shop with three machines and two jobs. Job 1 needs one time unit on machine 1 and 3 time units on machine 2. Job 2 needs 2 time units on machine 3 and 3 time units on machine 2. Both jobs have to be processed last on machine 2. Consider the schedule which processes job 2 on machine 2 before job 1 (see Figure 2.4). It is clear that this schedule is active; reversing the sequence of the two jobs on machine 2 postpones the processing of job 2. However, the schedule is *not* non-delay. Machine 2 remains idle till time 2, while there is already a job available for processing at time 1. ||

It can be shown that, when the objective γ is regular, there exists for $Jm \mid\mid \gamma$ an optimal schedule that is active.

An even larger class of nonpreemptive schedules can be defined as follows.

Definition 2.3.5 (Semi-Active Schedule). *A feasible nonpreemptive schedule is called semi-active if no operation can be completed earlier without changing the order of processing on any one of the machines.*

It is clear that an active schedule has to be semi-active. However, the reverse is not necessarily true.

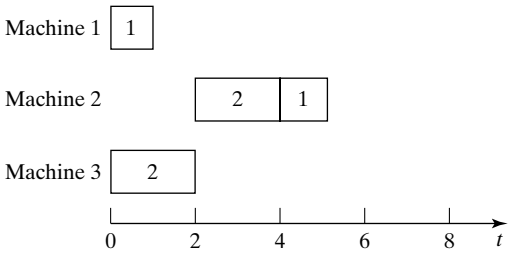


Fig. 2.5 A semi-active schedule that is not active.

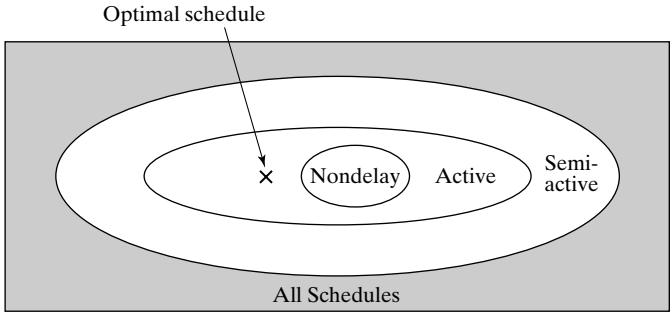


Fig. 2.6 Venn diagram of classes of nonpreemptive schedules for job shops

Example 2.3.6 (A Semi-Active Schedule)

Consider again a job shop with three machines and two jobs. The routing of the two jobs is the same as in the previous example. The processing times of job 1 on machines 1 and 2 are both equal to 1. The processing times of job 2 on machines 2 and 3 are both equal to 2. Consider the schedule under which job 2 is processed on machine 2 before job 1 (see Figure 2.5). This implies that job 2 starts its processing on machine 2 at time 2 and job 1 starts its processing on machine 2 at time 4. This schedule is semi-active. However, it is not active, as job 1 can be processed on machine 2 without delaying the processing of job 2 on machine 2.

An example of a schedule that is not even semi-active can be constructed easily. Postpone the start of the processing of job 1 on machine 2 for one time unit, i.e., machine 2 is kept idle for one unit of time between the processing of jobs 2 and 1. Clearly, this schedule is not even semi-active. ||

Figure 2.6 shows a Venn diagram of the three classes of nonpreemptive schedules: the nonpreemptive non-delay schedules, the active schedules, and the semi-active schedules.

2.4 Complexity Hierarchy

Often, an algorithm for one scheduling problem can be applied to another scheduling problem as well. For example, $1 \mid\mid \sum C_j$ is a special case of $1 \mid\mid \sum w_j C_j$ and a procedure for $1 \mid\mid \sum w_j C_j$ can, of course, also be used for $1 \mid\mid \sum C_j$. In complexity terminology it is then said that $1 \mid\mid \sum C_j$ *reduces* to $1 \mid\mid \sum w_j C_j$. This is usually denoted by

$$1 \mid\mid \sum C_j \quad \propto \quad 1 \mid\mid \sum w_j C_j.$$

Based on this concept a chain of reductions can be established. For example,

$$1 \mid\mid \sum C_j \quad \propto \quad 1 \mid\mid \sum w_j C_j \quad \propto \quad Pm \mid\mid \sum w_j C_j \quad \propto \quad Qm \mid prec \mid \sum w_j C_j.$$

Of course, there are also many problems that are not comparable with one another. For example, $Pm \mid\mid \sum w_j T_j$ is not comparable to $Jm \mid\mid C_{\max}$.

A considerable effort has been made to establish a problem hierarchy describing the relationships between the hundreds of scheduling problems. In the comparisons between the complexities of the different scheduling problems it is of interest to know how a change in a single element in the classification of a problem affects its complexity. In Figure 2.7 a number of graphs are exhibited that help determine the complexity hierarchy of deterministic scheduling problems. Most of the hierarchy depicted in these graphs is relatively straightforward. However, two of the relationships may need some explaining, namely

$$\alpha \mid \beta \mid L_{\max} \quad \propto \quad \alpha \mid \beta \mid \sum U_j$$

and

$$\alpha \mid \beta \mid L_{\max} \quad \propto \quad \alpha \mid \beta \mid \sum T_j.$$

It can, indeed, be shown that a procedure for $\alpha \mid \beta \mid \sum U_j$ and a procedure for $\alpha \mid \beta \mid \sum T_j$ can be applied to $\alpha \mid \beta \mid L_{\max}$ with only minor modifications (see Exercise 2.23).

A significant amount of research in deterministic scheduling has been devoted to finding efficient, so-called polynomial time, algorithms for scheduling problems. However, many scheduling problems do not have a polynomial time algorithm; these problems are the so-called *NP-hard* problems. Verifying that a problem is NP-hard requires a formal mathematical proof (see Appendix D).

Research in the past has focused in particular on the borderline between polynomial time solvable problems and NP-hard problems. For example, in the string of problems described above, $1 \mid\mid \sum w_j C_j$ can be solved in polynomial time, whereas $Pm \mid\mid \sum w_j C_j$ is NP-hard, which implies that $Qm \mid prec \mid \sum w_j C_j$ is also NP-hard. The following examples illustrate the borderlines between easy and hard problems within given sets of problems.

Example 2.4.1 (A Complexity Hierarchy)

Consider the problems

(i) $1 \mid\mid C_{\max}$,

(ii) $P2 \mid\mid C_{\max}$,

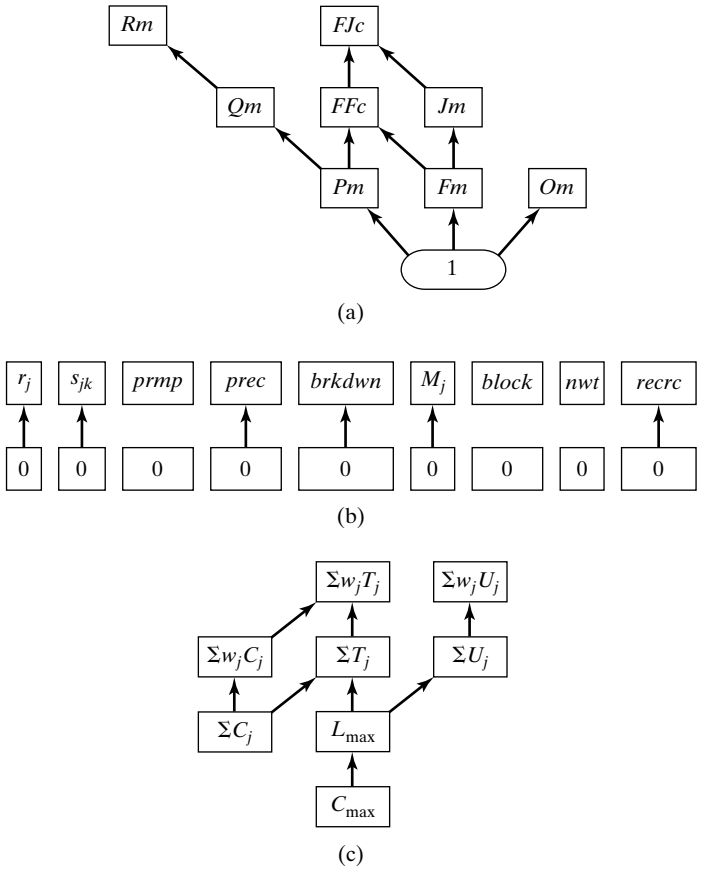


Fig. 2.7 Complexity hierarchies of deterministic scheduling problems: (a) Machine environments (b) Processing restrictions and constraints (c) Objective functions

(iii) $F2 \mid\mid C_{\max}$,
(iv) $Jm \mid\mid C_{\max}$,
(v) $FFc \mid\mid C_{\max}$.

The complexity hierarchy is depicted in Figure 2.8. ||

Example 2.4.2 (A Complexity Hierarchy)

Consider the problems

(i) $1 \mid\mid L_{\max}$,
(ii) $1 \mid prmp \mid L_{\max}$,
(iii) $1 \mid r_j \mid L_{\max}$,

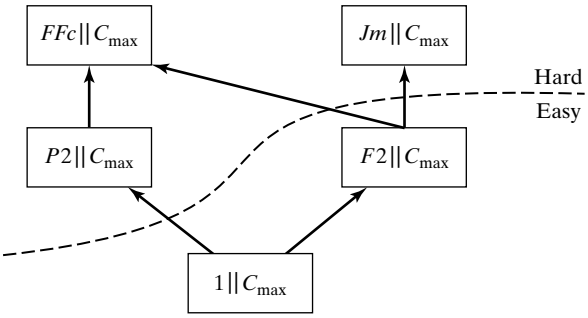


Fig. 2.8 Complexity hierarchy of problems in Example 2.4.1

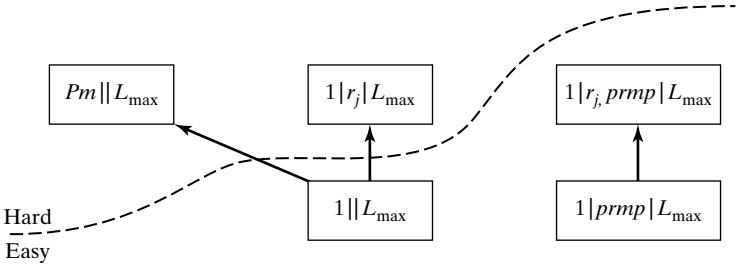


Fig. 2.9 Complexity hierarchy of problems in Example 2.4.2

(iv) $1 \mid r_j, prmp \mid L_{\max}$,
(v) $Pm \mid\mid L_{\max}$.

The complexity hierarchy is depicted in Figure 2.9. ||

Exercises (Computational)

2.1. Consider the instance of $1 \mid\mid \sum w_j C_j$ with the following processing times and weights.

jobs	1	2	3	4
w_j	6	11	9	5
p_j	3	5	7	4

(a) Find the optimal sequence and compute the value of the objective.

(b) Give an argument for positioning jobs with larger weight more towards the beginning of the sequence and jobs with smaller weight more towards the end of the sequence.

(c) Give an argument for positioning jobs with smaller processing time more towards the beginning of the sequence and jobs with larger processing time more towards the end of the sequence.

(d) Determine which one of the following two generic rules is the most suitable for the problem:

(i) sequence the jobs in decreasing order of $w_j - p_j$;

(ii) sequence the jobs in decreasing order of w_j/p_j.

2.2. Consider the instance of $1 \mid\mid L_{\max}$ with the following processing times and due dates.

jobs	1	2	3	4
p_j	5	4	3	6
d_j	3	5	11	12

(a) Find the optimal sequence and compute the value of the objective.

(b) Give an argument for positioning jobs with earlier due dates more towards the beginning of the sequence and jobs with later due dates more towards the end of the sequence.

(c) Give an argument for positioning jobs with smaller processing time more towards the beginning of the sequence and jobs with larger processing time more towards the end of the sequence.

(d) Determine which one of the following four rules is the most suitable generic rule for the problem:

(i) sequence the jobs in increasing order of $d_j + p_j$;

(ii) sequence the jobs in increasing order of $d_j p_j$;

(iii) sequence the jobs in increasing order of d_j;

(iv) sequence the jobs in increasing order of p_j.

2.3. Consider the instance of $1 \mid\mid \sum U_j$ with the following processing times and due dates.

jobs	1	2	3	4
p_j	7	6	4	8
d_j	8	9	11	14

(a) Find all optimal sequences and compute the value of the objective.

(b) Formulate a generic rule based on the due dates and processing times that yields an optimal sequence for any instance.

2.4. Consider the instance of $1 \mid\mid \sum T_j$ with the following processing times and due dates.

jobs	1	2	3	4
p_j	7	6	8	4
d_j	8	9	10	14

(a) Find all optimal sequences.

(b) Formulate a generic rule that is a function of the due dates and processing times that yields an optimal sequence for any instance.

2.5. Find the optimal sequence for $P5 \mid\mid C_{\max}$ with the following 11 jobs.

jobs	1	2	3	4	5	6	7	8	9	10	11
p_j	9	9	8	8	7	7	6	6	5	5	5

2.6. Consider the instance of $F2 \mid prmu \mid C_{\max}$ with the following processing times.

jobs	1	2	3	4
p_{1j}	8	6	4	12
p_{2j}	4	9	10	6

Find all optimal sequences and determine the makespan under an optimal sequence.

2.7. Consider the instance of $F2 \mid block \mid C_{\max}$ with the same jobs and the same processing times as in Exercise 2.6. There is no (zero) buffer between the two machines. Find all optimal sequences and compute the makespan under an optimal sequence.

2.8. Consider the instance of $F2 \mid nwt \mid C_{\max}$ with the same jobs and the same processing times as in Exercise 2.6. Find all optimal sequences and compute the makespan under an optimal sequence.

2.9. Consider the instance of $O2 \mid\mid C_{\max}$ with 4 jobs. The processing times of the four jobs on the two machines are again as in Exercise 2.6. Find all optimal schedules and compute the makespan under an optimal schedule.

2.10. Consider the instance of $J2 \mid\mid C_{\max}$ with 4 jobs. The processing times of the four jobs on the two machines are again as in Exercise 2.6. Jobs 1 and 2 have to be processed first on machine 1 and then on machine 2, while jobs 3 and 4 have to be processed first on machine 2 and then on machine 1. Find all optimal schedules and determine the makespan under an optimal schedule.

Exercises (Theory)

2.11. Explain why $\alpha \mid p_j = 1, r_j \mid \gamma$ is easier than $\alpha \mid prmp, r_j \mid \gamma$ when all processing times, release dates and due dates are integer.

2.12. Consider $1 \mid s_{jk} = a_k + b_j \mid C_{\max}$. That is, job j has two parameters associated with it, namely a_j and b_j. If job j is followed by job k, there is a setup time $s_{jk} = a_k + b_j$ required before the start of job k's processing. The setup time of the first job in the sequence, s_{0k} is a_k, while the "clean-up" time at the completion of the last job in the sequence, s_{j0}, is b_j. Show that this problem is equivalent to $1 \mid\mid C_{\max}$ and that the makespan therefore does not depend on the sequence. Find an expression for the makespan.

2.13. Show that $1 \mid s_{jk} \mid C_{\max}$ is equivalent to the following Travelling Salesman Problem: A travelling salesman starts out from city 0, visits cities $1, 2, \ldots, n$ and returns to city 0, while minimizing the total distance travelled. The distance from city 0 to city k is s_{0k}; the distance from city j to city k is s_{jk} and the distance from city j to city 0 is s_{j0}.

2.14. Show that $1 \mid brkdwn, prmp \mid \sum w_j C_j$ reduces to $1 \mid r_j, prmp \mid \sum w_j C_j$.

2.15. Show that $1 \mid p_j = 1 \mid \sum w_j T_j$ and $1 \mid p_j = 1 \mid L_{\max}$ are equivalent to the *assignment* problem (see Appendix A for a definition of the assignment problem).

2.16. Show that $Pm \mid p_j = 1 \mid \sum w_j T_j$ and $Pm \mid p_j = 1 \mid L_{\max}$ are equivalent to the *transportation* problem (see Appendix A for a definition of the transportation problem).

2.17. Consider $P \mid\mid C_{\max}$. Show that for any non-delay schedule the following inequalities hold:

$$\frac{\sum p_j}{m} \leq C_{\max} \leq 2 \times \max\left(p_1, \ldots, p_n, \frac{\sum p_j}{m}\right).$$

2.18. Show how $Pm \mid M_j \mid \gamma$ reduces to $Rm \mid\mid \gamma$.

2.19. Show that $F2 \mid block \mid C_{\max}$ is equivalent to $F2 \mid nwt \mid C_{\max}$ and show that both problems are special cases of $1 \mid s_{jk} \mid C_{\max}$ and therefore special cases of the Travelling Salesman Problem.

2.20. Consider an instance of $Om \mid \beta \mid \gamma$ and an instance of $Fm \mid \beta \mid \gamma$. The two instances have the same number of machines, the same number of jobs, and the jobs have the same processing times on the m machines. The two instances are completely identical with the exception that one instance is an open shop and the other instance a flow shop. Show that the value of the objective under the optimal sequence in the flow shop is at least as large as the value of the objective under the optimal sequence in the open shop.

2.21. Consider $O2 \mid\mid C_{\max}$. Show that

$$C_{\max} \geq \max \Big(\sum_{j=1}^{n} p_{1j}, \sum_{j=1}^{n} p_{2j} \Big).$$

Find an instance of this problem where the optimal makespan is *strictly* larger than the RHS.

2.22. Describe the complexity relationships between the problems

(i) $1 \mid\mid \sum w_j C_j$,
(ii) $1 \mid d_j = d \mid \sum w_j T_j$,
(iii) $1 \mid p_j = 1 \mid \sum w_j T_j$,
(iv) $1 \mid\mid \sum w_j T_j$,
(v) $Pm \mid p_j = 1 \mid \sum w_j T_j$,
(vi) $Pm \mid\mid \sum w_j T_j$.

2.23. Show that $\alpha \mid \beta \mid L_{\max}$ reduces to $\alpha \mid \beta \mid \sum T_j$ as well as to $\alpha \mid \beta \mid \sum U_j$. (*Hint:* Note that if the minimum $L_{\max}$ is zero, the optimal solution with regard to $\sum U_j$ and $\sum T_j$ is zero as well. It suffices to show that a polynomial time procedure for $\alpha \mid \beta \mid \sum U_j$ can be adapted easily for application to $\alpha \mid \beta \mid L_{\max}$. This can be done through a parametric analysis on the d_j, i.e., solve $\alpha \mid \beta \mid \sum U_j$ with due dates $d_j + z$ and vary z.)

Comments and References

One of the first classification schemes for scheduling problems appeared in Conway, Maxwell and Miller (1967). Lawler, Lenstra and Rinnooy Kan (1982), in their survey paper, modified and refined this scheme extensively. Herrmann, Lee and Snowdon (1993) made another round of extensions. The framework presented here is another variation of the Lawler, Lenstra and Rinnooy Kan (1982) notation, with a slightly different emphasis.

For a survey of scheduling problems subject to availability constraints (*brkdwn*), see Lee (2004) For surveys on scheduling problems with non-regular objective functions, see Raghavachari (1988) and Baker and Scudder (1990). For a survey of scheduling problems with job families and scheduling problems with batch processing, see Potts and Kovalyov (2000).

The definitions of non-delay, active, and semi-active schedules have been around for a long time; see, for example, Giffler and Thompson (1960) and French (1982) for a comprehensive overview of classes of schedules. Example 2.3.2, which illustrates some of the anomalies of non-delay schedules, is due to Graham (1966).

The complexity hierarchy of scheduling problems is motivated primarily by the work of Rinnooy Kan (1976), Lenstra (1977), Lageweg, Lawler, Lenstra and Rinnooy Kan (1981, 1982) and Lawler, Lenstra, Rinnooy Kan and Shmoys (1993). For more on reducibility in scheduling, see Timkovsky (2000).

Chapter 3

Single Machine Models (Deterministic)

Single machine models are important for various reasons. The single machine environment is very simple and a special case of all other environments. Single machine models often have properties that neither machines in parallel nor machines in series have. The results that can be obtained for single machine models not only provide insights into the single machine environment, they also provide a basis for heuristics that are applicable to more complicated machine environments. In practice, scheduling problems in more complicated machine environments are often decomposed into subproblems that deal with single machines. For example, a complicated machine environment with a single bottleneck may give rise to a single machine model.

In this chapter various single machine models are analyzed in detail. The total weighted completion time objective is considered first, followed by several due date related objectives, including the maximum lateness, the number of tardy jobs, the total tardiness and the total weighted tardiness. All objective functions considered in this chapter are regular.

In most models considered in this chapter there is no advantage in having preemptions; for these models it can be shown that the optimal schedule in the class of preemptive schedules is nonpreemptive. However, if jobs are released at different points in time, then it may be advantageous to preempt. If jobs are released at different points in time in a nonpreemptive environment, then

M.L. Pinedo, *Scheduling*, DOI: 10.1007/978-0-387-78935-4_3,

it may be advantageous to allow for unforced idleness (i.e., an optimal schedule may not be non-delay).

3.1 The Total Weighted Completion Time

The first objective to be considered is the total weighted completion time, i.e., $1 \mid\mid \sum w_j C_j$. The weight w_j of job j may be regarded as an importance factor; it may represent either a holding cost per unit time or the value already added to job j. This problem gives rise to one of the better known rules in scheduling theory, the so-called *Weighted Shortest Processing Time first (WSPT)* rule. According to this rule the jobs are ordered in decreasing order of w_j/p_j.

Theorem 3.1.1. *The WSPT rule is optimal for* $1 \mid\mid \sum w_j C_j$.

Proof. By contradiction. Suppose a schedule $\mathcal{S}$, that is not WSPT, is optimal. In this schedule there must be at least two adjacent jobs, say job j followed by job k, such that

$$\frac{w_j}{p_j} < \frac{w_k}{p_k}.$$

Assume job j starts its processing at time t. Perform a so-called *Adjacent Pairwise Interchange* on jobs j and k. Call the new schedule $\mathcal{S}'$. While under the original schedule $\mathcal{S}$ job j starts its processing at time t and is followed by job k, under the new schedule $\mathcal{S}'$ job k starts its processing at time t and is followed by job j. All other jobs remain in their original position. The total weighted completion time of the jobs processed before jobs j and k is not affected by the interchange. Neither is the total weighted completion time of the jobs processed after jobs j and k. Thus the difference in the values of the objectives under schedules $\mathcal{S}$ and $\mathcal{S}'$ is due only to jobs j and k (see Figure 3.1). Under $\mathcal{S}$ the total weighted completion time of jobs j and k is

$$(t + p_j)w_j + (t + p_j + p_k)w_k,$$

while under $\mathcal{S}'$ it is

$$(t + p_k)w_k + (t + p_k + p_j)w_j.$$

It is easily verified that if $w_j/p_j < w_k/p_k$ the sum of the two weighted completion times under $\mathcal{S}'$ is strictly less than under $\mathcal{S}$. This contradicts the optimality of $\mathcal{S}$ and completes the proof of the theorem. □

The computation time needed to order the jobs according to WSPT is the time required to sort the jobs according to the ratio of the two parameters. A simple sort can be done in $O(n\ log(n))$ time, see Example D.1.1 in Appendix D.

How is the minimization of the total weighted completion time affected by precedence constraints? Consider the simplest form of precedence constraints, i.e., precedence constraints that take the form of parallel chains (see Figure 3.2).

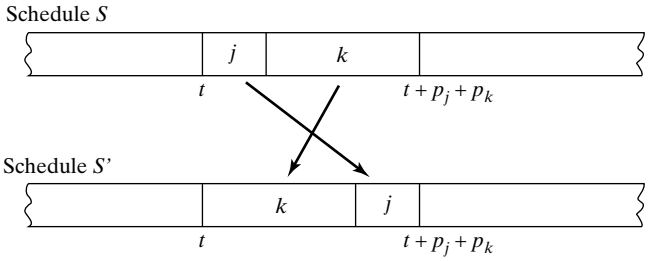


Fig. 3.1 A pairwise interchange of jobs j and k

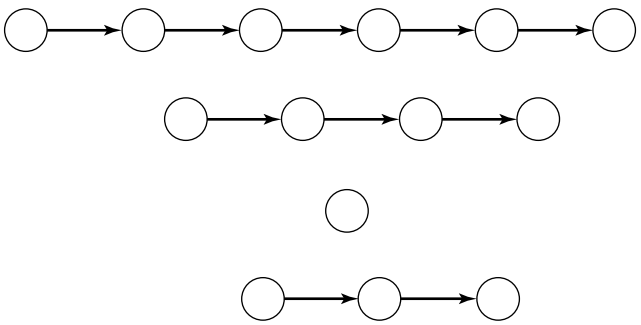


Fig. 3.2 Precedence constraints in the form of chains

This problem can still be solved by a relatively simple and very efficient (polynomial time) algorithm. This algorithm is based on some fundamental properties of scheduling with precedence constraints.

Consider two chains of jobs. One chain, say Chain I, consists of jobs $1, \ldots, k$ and the other chain, say Chain II, consists of jobs $k+1, \ldots, n$. The precedence constraints are as follows:

$$1 \to 2 \to \cdots \to k$$

and

$$k+1 \to k+2 \to \cdots \to n.$$

The next lemma is based on the assumption that if the scheduler decides to start processing jobs of one chain he has to complete the *entire* chain before he is allowed to work on jobs of the other chain. The question is: if the scheduler wishes to minimize the total weighted completion time of the n jobs, which one of the two chains should he process first?

Lemma 3.1.2. *If*

$$\frac{\sum_{j=1}^{k} w_j}{\sum_{j=1}^{k} p_j} \; > \; (<) \; \frac{\sum_{j=k+1}^{n} w_j}{\sum_{j=k+1}^{n} p_j},$$

then it is optimal to process the chain of jobs $1, \ldots, k$ *before (after) the chain of jobs* $k+1, \ldots, n$.

Proof. By contradiction. Under sequence $1, \ldots, k, k+1, \ldots, n$ the total weighted completion time is

$$w_1 p_1 + \cdots + w_k \sum_{j=1}^{k} p_j + w_{k+1} \sum_{j=1}^{k+1} p_j + \cdots + w_n \sum_{j=1}^{n} p_j,$$

while under sequence $k+1, \ldots, n, 1, \ldots, k$ it is

$$w_{k+1} p_{k+1} + \cdots + w_n \sum_{j=k+1}^{n} p_j + w_1 \Big(\sum_{j=k+1}^{n} p_j + p_1 \Big) + \cdots + w_k \sum_{j=1}^{n} p_j.$$

The total weighted completion time of the first sequence is less than the total weighted completion time of the second sequence if

$$\frac{\sum_{j=1}^{k} w_j}{\sum_{j=1}^{k} p_j} > \frac{\sum_{j=k+1}^{n} w_j}{\sum_{j=k+1}^{n} p_j}$$

The result follows. □

An interchange between two adjacent chains of jobs is usually referred to as an *Adjacent Sequence Interchange*. Such an interchange is a generalization of an Adjacent Pairwise Interchange.

An important characteristic of chain

$$1 \to 2 \to \cdots \to k$$

is defined as follows: let l^* satisfy

$$\frac{\sum_{j=1}^{l^*} w_j}{\sum_{j=1}^{l^*} p_j} = \max_{1 \le l \le k} \Big(\frac{\sum_{j=1}^{l} w_j}{\sum_{j=1}^{l} p_j} \Big).$$

The ratio on the left-hand side is called the ρ-factor of chain $1, \ldots, k$ and is denoted by $\rho(1, \ldots, k)$. Job l^* is referred to as the job that determines the ρ-factor of the chain.

Suppose now that the scheduler does not have to complete all the jobs in a chain before he is allowed to work on another chain. He may process some jobs of one chain (while adhering to the precedence constraints), switch over to another chain, and, at some later point in time, return to the first chain. If, in the case of multiple chains, the total weighted completion time is the objective function, then the following result holds.

Lemma 3.1.3. *If job l^* determines $\rho(1,\ldots,k)$, then there exists an optimal sequence that processes jobs $1,\ldots,l^*$ one after another without any interruption by jobs from other chains.*

Proof. By contradiction. Suppose that under the optimal sequence the processing of the subsequence $1,\ldots,l^*$ is interrupted by a job, say job v, from another chain. That is, the optimal sequence contains the subsequence $1,\ldots,u,v,u+1,\ldots,l^*$, say subsequence $\mathcal{S}$. It suffices to show that either with subsequence $v,1,\ldots,l^*$, say $\mathcal{S}'$, or with subsequence $1,\ldots,l^*,v$, say $\mathcal{S}''$, the total weighted completion time is less than with subsequence $\mathcal{S}$. If it is not less with the first subsequence, then it has to be less with the second and vice versa. From Lemma 3.1.2 it follows that if the total weighted completion time with $\mathcal{S}$ is less than with $\mathcal{S}'$ then

$$\frac{w_v}{p_v} < \frac{w_1+w_2+\ldots+w_u}{p_1+p_2+\ldots+p_u}.$$

From Lemma 3.1.2 it also follows that if the total weighted completion time with $\mathcal{S}$ is less than with $\mathcal{S}''$ then

$$\frac{w_v}{p_v} > \frac{w_{u+1}+w_{u+2}+\ldots+w_{l^*}}{p_{u+1}+p_{u+2}+\ldots+p_{l^*}}.$$

If job l^* is the job that determines the ρ-factor of chain $1,\ldots,k$, then

$$\frac{w_{u+1}+w_{u+2}+\ldots+w_{l^*}}{p_{u+1}+p_{u+2}+\ldots+p_{l^*}} > \frac{w_1+w_2+\ldots+w_u}{p_1+p_2+\ldots+p_u}.$$

If $\mathcal{S}$ is better than $\mathcal{S}''$, then

$$\frac{w_v}{p_v} > \frac{w_{u+1}+w_{u+2}+\ldots+w_{l^*}}{p_{u+1}+p_{u+2}+\ldots+p_{l^*}} > \frac{w_1+w_2+\ldots+w_u}{p_1+p_2+\ldots+p_u}.$$

So $\mathcal{S}'$ is therefore better than $\mathcal{S}$. The same argument goes through if the interruption of the chain is caused by more than one job. □

The result in Lemma 3.1.3 is intuitive. The condition of the lemma implies that the ratios of the weight divided by the processing time of the jobs in the string $1,\ldots,l^*$ must be increasing in some sense. If one had already decided to start processing a string of jobs, it makes sense to continue processing the string until job l^* is completed without processing any other job in between.

The two previous lemmas contain the basis for a simple algorithm that minimizes the total weighted completion time when the precedence constraints take the form of chains.

Algorithm 3.1.4 (Total Weighted Completion Time and Chains)
Whenever the machine is freed, select among the remaining chains the one with the highest ρ-factor. Process this chain without interruption up to and including the job that determines its ρ-factor. ||

The following example illustrates the use of the algorithm.

Example 3.1.5 (Total Weighted Completion Time and Chains)
Consider the following two chains:

$$1 \rightarrow 2 \rightarrow 3 \rightarrow 4$$

and

$$5 \rightarrow 6 \rightarrow 7$$

The weights and processing times of the jobs are given in the table below.

jobs	1	2	3	4	5	6	7
w_j	6	18	12	8	8	17	18
p_j	3	6	6	5	4	8	10

The ρ-factor of the first chain is (6+18)/(3+6) and is determined by job 2. The ρ-factor of the second chain is (8+17)/(4+8) and is determined by job 6. As 24/9 is larger than 25/12 jobs 1 and 2 are processed first. The ρ-factor of the remaining part of the first chain is 12/6 and determined by job 3. As 25/12 is larger than 12/6 jobs 5 and 6 follow jobs 1 and 2. The ρ-factor of the remaining part of the second chain is 18/10 and is determined by job 7; so job 3 follows job 6. As the w_j/p_j ratio of job 7 is higher than the ratio of job 4, job 7 follows job 3 and job 4 goes last. ||

Polynomial time algorithms have been obtained for $1 \mid prec \mid \sum w_j C_j$ with more general precedence constraints than the parallel chains considered above. However, with *arbitrary* precedence constraints, the problem is strongly NP-hard.

Up to now all jobs were assumed to be available at time zero. Consider the problem where jobs are released at different points in time and the scheduler is allowed to preempt, i.e., $1 \mid r_j, prmp \mid \sum w_j C_j$. The first question that comes to mind is whether a preemptive version of the WSPT rule is optimal. A preemptive version of the WSPT rule can be formulated as follows: At any point in time the available job with the highest ratio of weight to *remaining* processing time is selected for processing. The priority level of a job thus increases while being processed and a job can therefore not be preempted by another job that already was available at the start of its processing. However, a job may be preempted by a newly released job with a higher priority factor. Although this rule may appear a logical extension of the nonpreemptive WSPT rule, it does not necessarily lead to an optimal schedule since the problem is strongly NP-hard (see Appendix E).

If all the weights are equal, then the $1 \mid r_j, prmp \mid \sum C_j$ problem is easy (see Exercise 3.15). On the other hand, the nonpreemptive version of this problem, i.e., $1 \mid r_j \mid \sum C_j$, is strongly NP-hard.

In Chapter 2 the total weighted discounted completion time $\sum w_j(1-e^{-rC_j})$, with r being the discount factor, is described as an objective that is, in a way, a generalization of the total weighted (undiscounted) completion time. The problem $1 \mid\mid \sum w_j(1-e^{-rC_j})$ gives rise to a different priority rule, namely the rule that schedules the jobs in decreasing order of

$$\frac{w_j e^{-rp_j}}{1-e^{-rp_j}}.$$

In what follows this rule is referred to as the *Weighted Discounted Shortest Processing Time first (WDSPT)* rule.

Theorem 3.1.6. *For $1 \mid\mid \sum w_j(1-e^{-rC_j})$ the WDSPT rule is optimal.*

Proof. By contradiction. Again, assume that a different schedule, say schedule $\mathcal{S}$, is optimal. Under this schedule there have to be two jobs j and k, job j followed by job k, such that

$$\frac{w_j e^{-rp_j}}{1-e^{-rp_j}} < \frac{w_k e^{-rp_k}}{1-e^{-rp_k}}.$$

Assume job j starts its processing at time t. An Adjacent Pairwise Interchange between these two jobs results in a schedule $\mathcal{S}'$. It is clear that the only difference in the objective is due to jobs j and k. Under $\mathcal{S}$ the contribution of jobs j and k to the objective function equals

$$w_j\Big(1-e^{-r(t+p_j)}\Big) + w_k\Big(1-e^{-r(t+p_j+p_k)}\Big).$$

The contribution of jobs j and k to the objective under $\mathcal{S}'$ is obtained by interchanging the j's and k's in this expression. Elementary algebra then shows that the value of objective function under $\mathcal{S}'$ is less than under $\mathcal{S}$. This leads to the contradiction that completes the proof. □

As discussed in Chapter 2, the total undiscounted weighted completion time is basically a limiting case of the total discounted weighted completion time $\sum w_j(1-e^{-rC_j})$. The WDSPT rule results in the same sequence as the WSPT rule if r is sufficiently close to zero (note that the WDSPT rule is not properly defined for $r = 0$).

Both $\sum w_j C_j$ and $\sum w_j(1-e^{-rC_j})$ are special cases of the more general objective function $\sum w_j h(C_j)$. It has been shown that only the functions $h(C_j) = C_j$ and $h(C_j) = 1-e^{-rC_j}$ lead to simple priority rules that order the jobs in decreasing order of some function $g(w_j, p_j)$. No such priority function g, that guarantees optimality, exists for any other cost function h. However, the objective $\sum h_j(C_j)$ can be dealt with via Dynamic Programming (see Appendix B).

In a similar way as Lemma 3.1.2 generalizes the Adjacent Pairwise Interchange argument for WSPT there exists an Adjacent Sequence Interchange

result that generalizes the Adjacent Pairwise Interchange argument used in the optimality proof for the WDSPT rule (see Exercise 3.21).

3.2 The Maximum Lateness

The objectives considered in the next four sections are due date related. The first due date related model is of a rather general nature, namely the problem $1 \mid prec \mid h_{\max}$, where

$$h_{\max} = \max\Big(h_1(C_1), \ldots, h_n(C_n)\Big)$$

with $h_j,\ j = 1, \ldots, n$, being nondecreasing cost functions. This objective is clearly due date related as the functions h_j may take any one of the forms depicted in Figure 2.1. This problem allows for an efficient *backward* dynamic programming algorithm even when the jobs are subject to arbitrary precedence constraints.

It is clear that the completion of the last job occurs at the makespan $C_{\max} = \sum p_j$, which is independent of the schedule. Let J denote the set of jobs already scheduled, which are processed during the time interval

$$[C_{\max} - \sum_{j \in J} p_j, \quad C_{\max}].$$

The complement of set J, set J^c, denotes the set of jobs still to be scheduled and the subset J' of J^c denotes the set of jobs that can be scheduled immediately before set J, i.e., the set of jobs all of whose successors are in J. Set J' is referred to as the set of *schedulable* jobs. The following backward algorithm yields an optimal schedule.

Algorithm 3.2.1 (Minimizing Maximum Cost)

Step 1.

Set $J = \emptyset$, $J^c = \{1, \ldots, n\}$
and J' *the set of all jobs with no successors.*

Step 2.

Let j^* *be such that*

$$h_{j^*}(\sum_{k \in J^c} p_k) = \min_{j \in J'} \Big(h_j(\sum_{k \in J^c} p_k)\Big)$$

Add j^* *to* J
Delete j^* *from* J^c
Modify J' *to represent the new set of schedulable jobs.*

Step 3.

If $J^c = \emptyset$ *STOP, otherwise go to Step 2.* ||

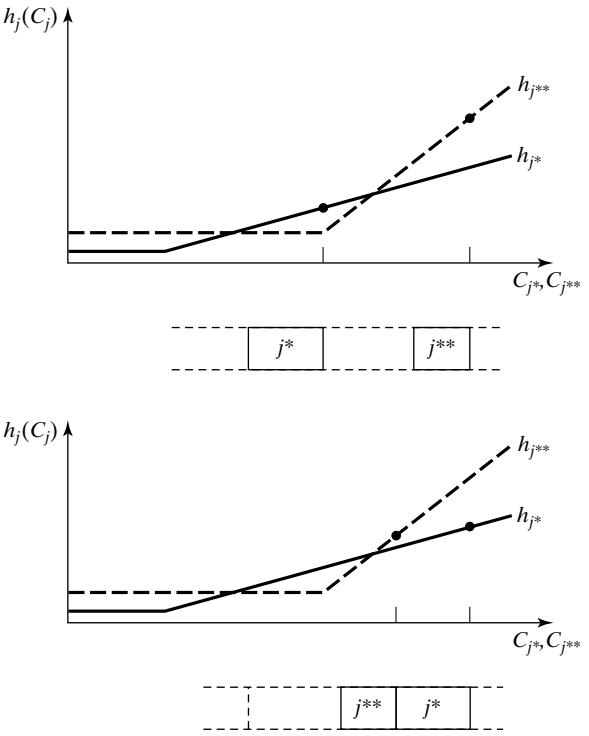


Fig. 3.3 Proof of optimality of Theorem 3.2.2

Theorem 3.2.2. *Algorithm 3.2.1 yields an optimal schedule for* $1 \mid prec \mid h_{\max}$.

Proof. By contradiction. Suppose in a given iteration job j^{**}, selected from J', does not have the minimum completion cost

$$h_{j^*}\Big(\sum_{k \in J^c} p_k\Big)$$

among the jobs in J'. The minimum cost job j^* must then be scheduled in a later iteration, implying that job j^* has to appear in the sequence before job j^{**}. A number of jobs may even appear between jobs j^* and j^{**} (see Figure 3.3).

To show that this sequence cannot be optimal, take job j^* and insert it in the schedule immediately following job j^{**}. All jobs in the original schedule between jobs j^* and j^{**}, including job j^{**} itself, are now completed earlier. The only job whose completion cost increases is job j^*. However, its completion cost now is, by definition, smaller than the completion cost of job j^{**} under the original schedule, so the maximum completion cost decreases after the insertion of job j^*. This completes the proof. □

The worst case computation time required by this algorithm can be established as follows. There are n steps needed to schedule the n jobs. In each step at most n jobs have to be considered. The overall running time of the algorithm is therefore bounded by $O(n^2)$.

The following example illustrates the application of this algorithm.

Example 3.2.3 (Minimizing Maximum Cost)

Consider the following three jobs.

jobs	1	2	3
p_j	2	3	5
$h_j(C_j)$	$1 + C_1$	$1.2\, C_2$	10

The makespan $C_{\max} = 10$ and $h_3(10) < h_1(10) < h_2(10)$ (as $10 < 11 < 12$). Job 3 is therefore scheduled last and has to start its processing at time 5. To determine which job is to be processed before job 3, $h_2(5)$ has to be compared with $h_1(5)$. Either job 1 or job 2 may be processed before job 3 in an optimal schedule as $h_1(5) = h_2(5) = 6$. So two schedules are optimal: $1, 2, 3$ and $2, 1, 3$. ||

The problem $1 \,||\, L_{\max}$ is the best known special case of $1 \mid prec \mid h_{\max}$. The function h_j is then defined as $C_j - d_j$ and the algorithm yields the schedule that orders the job in increasing order of their due dates, i.e., *Earliest Due Date (EDD) first.*

A generalization of $1 \,||\, L_{\max}$ is the problem $1 \mid r_j \mid L_{\max}$ with the jobs released at different points in time. This generalization, which does not allow preemption, is significantly harder than the problem with all jobs available at time 0. The optimal schedule is not necessarily a non-delay schedule. It may be advantageous to keep the machine idle just before the release of a new job.

Theorem 3.2.4. *The problem* $1 \mid r_j \mid L_{\max}$ *is strongly NP-hard.*

Proof. The proof is based on the fact that 3-PARTITION reduces to $1 \mid r_j \mid L_{\max}$. Given integers $a_1, \ldots, a_{3t}, b$, such that $b/4 < a_j < b/2$ and $\sum_{j=1}^{3t} a_j = tb$, the following instance of $1 \mid r_j \mid L_{\max}$ can be constructed. The number of jobs, n, is equal to $4t - 1$ and

$$
\begin{array}{llll}
r_j = jb + (j-1), & p_j = 1, & d_j = jb + j, & j = 1, \ldots, t-1, \\
r_j = 0, & p_j = a_{j-t+1}, & d_j = tb + (t-1), & j = t, \ldots, 4t-1.
\end{array}
$$

Let $z = 0$. A schedule with $L_{\max} \leq 0$ exists if and only if every job j, $j = 1, \ldots, t-1$, can be processed between r_j and $d_j = r_j + p_j$. This can be done if and only if the remaining jobs can be partitioned over the t intervals

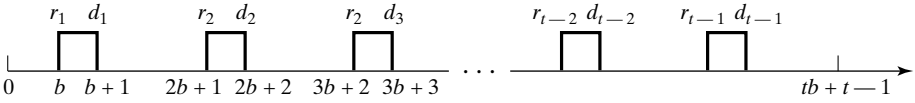


Fig. 3.4 $1 \mid r_j \mid L_{\max}$ is strongly NP-hard

of length b, which can be done if and only if 3-PARTITION has a solution (see Figure 3.4). □

The $1 \mid r_j \mid L_{\max}$ problem is important because it appears often as a subproblem in heuristic procedures for flow shop and job shop problems. It has received a considerable amount of attention that has resulted in a number of reasonably effective enumerative branch-and-bound procedures. Branch-and-bound procedures are basically enumeration schemes where certain schedules or classes of schedules are discarded by showing that the values of the objective obtained with schedules from this class are larger than a provable lower bound; this lower bound is greater than or equal to the value of the objective of a schedule obtained earlier.

A branch-and-bound procedure for $1 \mid r_j \mid L_{\max}$ can be constructed as follows. The branching process may be based on the fact that schedules are developed starting from the beginning of the schedule. There is a single node at level 0 which is the top of the tree. At this node no job has been put yet into any position in the sequence. There are n branches going down to n nodes at level 1. Each node at this level has a specific job put into the first position in the schedule. So, at each one of these nodes there are still $n-1$ jobs whose position in the schedule has not yet been determined. There are $n-1$ arcs emanating from each node at level 1 to level 2. There are therefore $(n-1) \times (n-2)$ nodes at level 2. At each node at level 2, the jobs in the first two positions are specified; at level k, the jobs in the first k positions are specified. Actually, it is not necessary to consider every remaining job as a candidate for the next position. If at a node at level $k-1$ jobs $j_1, \ldots, j_{k-1}$ are scheduled as the first $k-1$ jobs, then job j_k only has to be considered if

$$r_{j_k} < \min_{l \in J} \Big(\max(t, r_l) + p_l \Big),$$

where J denotes the set of jobs not yet scheduled and t denotes the time job j_k is supposed to start. The reason for this condition is clear: if job j_k does not satisfy this inequality, then selecting the job that minimizes the right-hand side instead of j_k does not increase the value of $L_{\max}$. The branching rule is thus fairly easy.

There are several ways in which bounds for nodes can be obtained. An easy lower bound for a node at level $k-1$ can be established by scheduling the remaining jobs J according to the *preemptive* EDD rule. The preemptive EDD rule is known to be optimal for $1 \mid r_j, prmp \mid L_{\max}$ (see Exercise 3.24) and

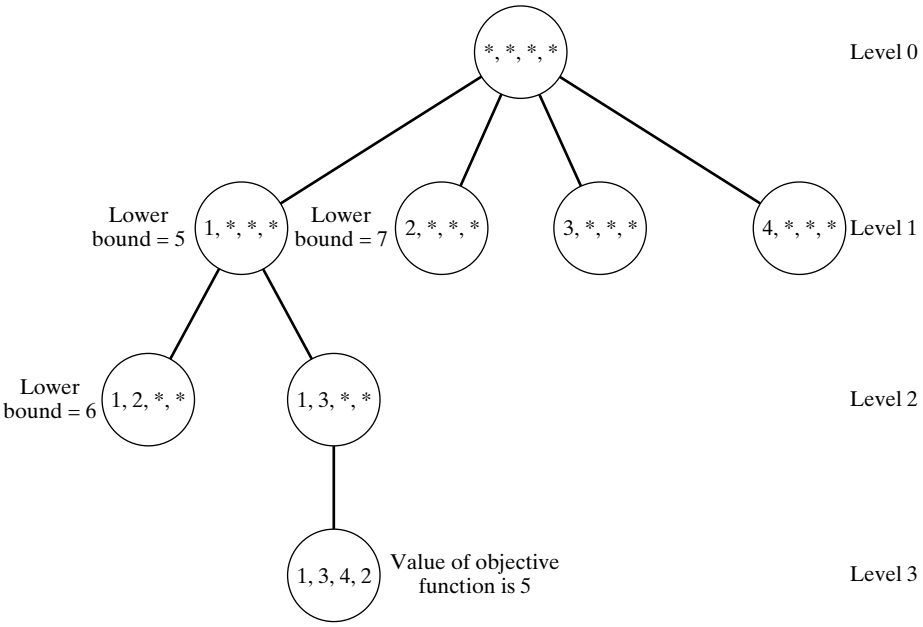


Fig. 3.5 Branch-and-bound procedure for Example 3.2.5

thus provides a lower bound for the problem at hand. If a preemptive EDD rule results in a nonpreemptive schedule, then all nodes with a higher lower bound may be disregarded.

Example 3.2.5 (Branch-and-Bound for Minimizing Maximum Lateness)

Consider the following 4 jobs.

$jobs$	1	2	3	4
p_j	4	2	6	5
r_j	0	1	3	5
d_j	8	12	11	10

At level 1 of the search tree there are four nodes: $(1, *, *, *)$, $(2, *, *, *)$, $(3, *, *, *)$ and $(4, *, *, *)$. It is easy to see that nodes $(3, *, *, *)$ and $(4, *, *, *)$ may be disregarded immediately. Job 3 is released at time 3; if job 2 would start its processing at time 1, job 3 still can start at time 3. Job 4 is released at time 5; if job 1 would start its processing at time 0, job 4 still can start at time 5 (see Figure 3.5).

Computing a lower bound for node $(1,*,*,*)$ according to the preemptive EDD rule results in a schedule where job 3 is processed during the time interval [4,5], job 4 during the time interval [5,10], job 3 (again) during interval [10,15] and job 2 during interval [15,17]. The $L_{\max}$ of this schedule, which provides a lower bound for node $(1,*,*,*)$, is 5. In a similar way a lower bound can be obtained for node $(2,*,*,*)$. The value of this lower bound is 7.

Consider node $(1,2,*,*)$ at level 2. The lower bound for this node is 6 and is determined by the (nonpreemptive) schedule $1,2,4,3$. Proceed with node $(1,3,*,*)$ at level 2. The lower bound is 5 and determined by the (nonpreemptive) schedule 1,3,4,2. From the fact that the lower bound for node $(1,*,*,*)$ is 5 and the lower bound for node $(2,*,*,*)$ is larger than 5 it follows that schedule $1,3,4,2$ has to be optimal. ||

The problem $1 \mid r_j, prec \mid L_{\max}$ can be handled in a similar way. This problem, from an enumeration point of view, is easier than the problem without precedence constraints, since the precedence constraints allow certain schedules to be ruled out immediately.

3.3 The Number of Tardy Jobs

Another due date related objective is $\sum U_j$. This objective may at first appear somewhat artificial and of no practical interest. However, in the real world it is a performance measure that is often monitored and according to which managers are being measured. It is equivalent to the percentage of on time shipments.

An optimal schedule for $1 \mid\mid \sum U_j$ takes the form of one set of jobs that will meet their due dates and that are scheduled first followed by the set of remaining jobs that will not meet their due dates and that are scheduled last. It follows from the results in the previous section that the first set of jobs have to be scheduled according to EDD in order to make sure that $L_{\max}$ is negative; the order in which the second set of jobs is scheduled is immaterial.

The problem $1 \mid\mid \sum U_j$ can be solved easily using a *forward* algorithm. Reorder the jobs in such a way that $d_1 \leq d_2 \leq \cdots \leq d_n$. The algorithm goes through n iterations. In iteration k of the algorithm jobs $1,2,\ldots,k$ are taken into consideration. Of these k jobs, the subset J refers to jobs that, in an optimal schedule, may be completed before their due dates and the subset J^d refers to jobs that already have been discarded and will not meet their due dates in the optimal schedule. In iteration k the set J^c refers to jobs $k+1, k+2, \ldots, n$.

Algorithm 3.3.1 (Minimizing Number of Tardy Jobs)

Step 1.

Set $J = \emptyset$, $J^c = \{1,\ldots,n\}$, *and* $J^d = \emptyset$.
Set the counter $k = 1$.

Step 2.

Add job k to J.
Delete job k from J^c.
Go to Step 3.

Step 3.

If

$$\sum_{j \in J} p_j \leq d_k,$$

go to Step 4.
Otherwise, let ℓ denote the job that satisfies

$$p_\ell = \max_{j \in J} (p_j).$$

Delete job ℓ from J.
Add job ℓ to J^d.

Step 4.

Set $J_k = J$.
If $k = n$ STOP,
otherwise set $k = k + 1$ and go to Step 2. ||

In words the algorithm can be described as follows. Jobs are added to the set of on-time jobs in increasing order of their due dates. If including job k to the set of scheduled jobs implies that job k would be completed late, then the scheduled job with the longest processing time, say job ℓ, is marked late and discarded. Since the algorithm basically orders the jobs according to their due dates, the worst case computation time is that of a simple sort, i.e., $O(n\ log(n))$.

Note that the algorithm creates in its last step n job sets $J_1, \ldots, J_n$. Set J_k is a subset of jobs $\{1, \ldots, k\}$, consisting of those jobs that are candidates for meeting their due dates in the final optimal schedule. Set J_n consists of all jobs that meet their due dates in the optimal schedule generated.

Theorem 3.3.2. *Algorithm 3.3.1 yields an optimal schedule for* $1 \,||\, \sum U_j$.

Proof. The proof uses the following notation and terminology. A job set J is called feasible if the jobs, scheduled according to EDD, all meet their due dates. A job set J is called $l-$optimal if it is a feasible subset of jobs $1, \ldots, l$ and if it has, among all feasible subsets of jobs $1, \ldots, l$, the maximum number of jobs.

The proof consists of three steps. The first step of the proof shows that the job sets $J_1, \ldots, J_n$ created in Step 4 of the algorithm are all feasible. This can be shown by induction (see Exercise 3.27).

The second step of the proof shows that for $l > k$, there exists an $l-$optimal set that consists of a subset of jobs J_k and a subset of jobs $k+1, \ldots, l$. To show this, assume it is true for $k-1$, i.e., there exists an $l-$optimal set J' that consists of a subset of jobs from set J_{k-1} and a subset of jobs $k, k+1, \ldots, l$.

It can be shown that an $l-$optimal set J'' can be created from J_k and jobs $k+1,\ldots,l$ by considering three cases:

Case 1: Set J_k consists of set J_{k-1} plus job k. In order to create set J'', just take set J'.

Case 2: Set J_k consists of set J_{k-1} plus job k minus some job q which is not an element of set J'. Again, in order to create set J'', just take set J'.

Case 3: Set J_k is equal to set J_{k-1} plus job k minus some job q which is an element of set J'. The argument is now a little bit more complicated. Since J_{k-1} plus k is not a feasible set, there must exist in the set that comprises J_{k-1} and k a job r that is not an element of J'. Take any such r. Now, to create set J'', take set J', include job r and delete job q. Clearly, set J'' is a subset of set J_k and jobs $k+1,\ldots,n$. Since the number of jobs in J'' is the same as the number of jobs in J', it only remains to be shown that J'' is feasible. Since J'' differs from J' only in its intersection with jobs $\{1,\ldots,k\}$, it suffices to verify two properties, namely that the job set which is the intersection of set J'' and set $\{1,\ldots,k\}$ is feasible and that the total processing time of the jobs in the intersection of J'' and $\{1,\ldots,k\}$ is less than or equal to the total processing time of the jobs in the intersection of J' and $\{1,\ldots,k\}$. The feasibility of the intersection of J'' and set $\{1,\ldots,k\}$ follows from the fact that it is a subset of J_k, which is feasible because of the first step of the proof. The second property follows from the fact that $p_r \leq p_q$.

The third and final step of the proof shows that set J_k is $k-$optimal for $k=1,\ldots,n$. It is clearly true for $k=0$ and $k=1$. Suppose it is true for $k-1$. From the previous step it follows that the set that comprises J_{k-1} and k must contain a $k-$optimal set. If set J_k contains the entire set J_{k-1} plus job k then it must clearly be $k-$optimal since J_{k-1} is $(k-1)$-optimal. If set J_{k-1} combined with job k is not feasible, then the $k-$optimal set must be a smaller set within the set that contains J_{k-1} and k; however, it must contain at least as many jobs as set J_{k-1}. Set J_k clearly satisfies this condition. □

Example 3.3.3 (Minimizing Number of Tardy Jobs)

Consider the following 5 jobs.

jobs	1	2	3	4	5
p_j	7	8	4	6	6
d_j	9	17	18	19	21

Jobs 1 and 2 can be positioned first and second in the sequence with both jobs being completed on time. Putting job 3 into the third position causes problems. Its completion time would be 19 while its due date is 18. Algorithm 3.3.1 prescribes the deletion of the job with the longest processing time among the first three jobs. Job 2 is therefore deleted and jobs 1 and 3 remain in the first two positions. If now job 4 follows job 3, it is completed on time at

17; however, if job 5 follows job 4, it is completed late. The algorithm then prescribes to delete the job with the longest processing time among those already scheduled, which is job 1. So the optimal schedule is $3, 4, 5, 1, 2$ with $\sum U_j = 2$. ||

Note that Algorithm 3.3.1 is an algorithm that goes forward in time. For this problem there is not any algorithm that goes backward in time. Note also that there may be many optimal schedules; characterizing the class of all optimal schedules seems to be a very difficult problem.

The generalization of this problem with weights, i.e., $1 \,||\, \sum w_j U_j$ is known to be NP-hard (see Appendix D). The special case with all due dates being equal is equivalent to the so-called *knapsack* problem. The due date is equivalent to the size of the knapsack, the processing times of the jobs are equivalent to the sizes of the items and the weights are equivalent to the benefits obtained by putting the items into the knapsack. A popular heuristic for this problem is the WSPT rule which sequences the jobs in decreasing order of w_j/p_j. A worst case analysis shows that this heuristic may perform arbitrarily badly, i.e., that the ratio

$$\frac{\sum w_j U_j(WSPT)}{\sum w_j U_j(OPT)}$$

may be arbitrarily large.

Example 3.3.4 (The WSPT Rule and a Knapsack)
Consider the following three jobs.

jobs	1	2	3
p_j	11	9	90
w_j	12	9	89
d_j	100	100	100

Scheduling the jobs according to WSPT results in the schedule $1, 2, 3$. The third job is completed late and $\sum w_j U_j(WSPT)$ is 89. Scheduling the jobs according to $2, 3, 1$ results in $\sum w_j U_j(OPT)$ being equal to 12. ||

3.4 The Total Tardiness - Dynamic Programming

The objective $\sum T_j$ is one that is important in practice as well. Minimizing the number of tardy jobs, $\sum U_j$, in practice cannot be the only objective to measure how due dates are being met. Some jobs may have to wait for an unacceptably long time if the number of late jobs is minimized. If instead the sum of the tardinesses is minimized it is less likely that the wait of any given job will be unacceptably long.

The model $1 \mid\mid \sum T_j$ has received an enormous amount of attention in the literature. For many years its computational complexity remained open, until its NP-hardness was established in 1990. As $1 \mid\mid \sum T_j$ is NP-hard in the ordinary sense it allows for a pseudo-polynomial time algorithm based on dynamic programming (see Appendix D). The algorithm is based on two preliminary results.

Lemma 3.4.1. *If $p_j \leq p_k$ and $d_j \leq d_k$, then there exists an optimal sequence in which job j is scheduled before job k.*

Proof. The proof of this result is left as an exercise. □

This type of result is useful when an algorithm has to be developed for a problem that is NP-hard. Such a result, often referred to as a *Dominance Result* or *Elimination Criterion*, often allows one to disregard a fairly large number of sequences. Such a dominance result may also be thought of as a set of precedence constraints on the jobs. The more precedence constraints created through such dominance results, the easier the problem becomes.

In the following lemma the sensitivity of an optimal sequence to the due dates is considered. Two problem instances are considered, both of which have n jobs with processing times $p_1, \ldots, p_n$. The first instance has due dates $d_1, \ldots, d_n$. Let C'_k be the latest possible completion time of job k in any optimal sequence, say $\mathcal{S}'$, for this instance. The second instance has due dates $d_1, \ldots, d_{k-1}, \max(d_k, C'_k), d_{k+1}, \ldots, d_n$. Let $\mathcal{S}''$ denote the optimal sequence with respect to this second set of due dates and C''_j the completion of job j under this second sequence.

Lemma 3.4.2. *Any sequence that is optimal for the second instance is also optimal for the first instance.*

Proof. Let $V'(\mathcal{S})$ denote the total tardiness under an arbitrary sequence $\mathcal{S}$ with respect to the first set of due dates and let $V''(\mathcal{S})$ denote the total tardiness under sequence $\mathcal{S}$ with respect to the second set of due dates. Now

$$V'(\mathcal{S}') = V''(\mathcal{S}') + A_k$$

and

$$V'(\mathcal{S}'') = V''(\mathcal{S}'') + B_k,$$

where, if $C'_k \leq d_k$ the two sets of due dates are the same and the sequence that is optimal for the second set is therefore also optimal for the first set. If $C'_k \geq d_k$, then

$$A_k = C'_k - d_k$$

and

$$B_k = \max(0, \min(C''_k, C'_k) - d_k)$$

It is clear that $A_k \geq B_k$. As $\mathcal{S}''$ is optimal for the second instance $V''(\mathcal{S}') \geq V''(\mathcal{S}'')$. Therefore $V'(\mathcal{S}') \geq V'(\mathcal{S}'')$ which completes the proof. □

In the remainder of this section it is assumed for purposes of exposition that (without loss of generality) all processing times are different, if necessary after an infinitesimal perturbation. Assume that $d_1 \leq \cdots \leq d_n$ and $p_k = \max(p_1, \ldots, p_n)$. That is, the job with the kth smallest due date has the longest processing time. From Lemma 3.4.1 it follows that there exists an optimal sequence in which jobs $\{1, \ldots, k-1\}$ all appear, in some order, before job k. Of the remaining $n-k$ jobs, i.e., jobs $\{k+1, \ldots, n\}$, some may appear before job k and some may appear after job k. The subsequent lemma focuses on these $n-k$ jobs.

Lemma 3.4.3. *There exists an integer δ, $0 \leq \delta \leq n-k$, such that there is an optimal sequence $\mathcal{S}$ in which job k is preceded by all jobs j with $j \leq k+\delta$ and followed by all jobs j with $j > k+\delta$.*

Proof. Let C'_k denote the latest possible completion time of job k in any sequence that is optimal with respect to the given due dates $d_1, \ldots, d_n$. Let $\mathcal{S}''$ be a sequence that is optimal with respect to the due dates $d_1, \ldots, d_{k-1}, \max(C'_k, d_k)$, $d_{k+1}, \ldots, d_n$ and that satisfies the condition stated in Lemma 3.4.1. Let C''_k denote the completion time of job k under this sequence. By Lemma 3.4.2 sequence $\mathcal{S}''$ is also optimal with respect to the original due dates. This implies that $C''_k \leq \max(C'_k, d_k)$. One can assume that job k is not preceded in $\mathcal{S}''$ by a job with a due date later than $\max(C'_k, d_k)$ (if this would have been the case this job would be on time and repositioning this job by inserting it immediately after job k would not increase the objective function). Also, job k has to be preceded by all jobs with a due date earlier than $\max(C'_k, d_k)$ (otherwise Lemma 3.4.1 would be violated). So δ can be chosen to be the largest integer such that $d_{k+\delta} \leq \max(C'_k, d_k)$. This completes the proof. □

In the dynamic programming algorithm a subroutine is required that generates an optimal schedule for the set of jobs $1, \ldots, l$ starting with the processing of this set at time t. Let k be the job with the longest processing time among these l jobs. From Lemma 3.4.3 it follows that for some δ $(0 \leq \delta \leq l-k)$ there exists an optimal sequence starting at t which may be regarded as a concatenation of three subsets of jobs, namely

(i) jobs $1, 2, \ldots, k-1, k+1, \ldots, k+\delta$ in some order, followed by
(ii) job k, followed by
(iii) jobs $k+\delta+1, k+\delta+2, \ldots, l$ in some order.

The completion time of job k, $C_k(\delta)$, is given by

$$C_k(\delta) = \sum_{j \leq k+\delta} p_j.$$

It is clear that for the entire sequence to be optimal the first and third subsets must be optimally sequenced within themselves. This suggests a dynamic programming procedure that determines an optimal sequence for a larger set of

jobs after having determined optimal sequences for proper subsets of the larger set. The subsets J used in this recursive procedure are of a very special type. A subset consists of all the jobs in a set $\{j, j+1, \ldots, l-1, l\}$ with processing times smaller than the processing time p_k of job k. Such a subset is denoted by $J(j,l,k)$. Let $V(J(j,l,k),t)$ denote the total tardiness of this subset under an optimal sequence, assuming that this subset starts at time t. The dynamic programming procedure can now be stated as follows.

Algorithm 3.4.4 (Minimizing Total Tardiness)
Initial Conditions

$$V(\emptyset, t) = 0,$$
$$V(\{j\}, t) = \max(0, t + p_j - d_j).$$

Recursive Relation

$$V(J(j,l,k),t) = \min_{\delta} \Big(V(J(j,k'+\delta,k'),t) \; + \; \max(0, C_{k'}(\delta) - d_{k'}) \; + \; V(J(k'+\delta+1,l,k'), C_{k'}(\delta)) \Big)$$

where k' is such that

$$p_{k'} = \max \Big(p_{j'} \mid j' \in J(j,l,k) \Big).$$

Optimal Value Function

$$V(\{1,\ldots,n\},0). \qquad ||$$

The optimal $\sum T_j$ value is given by $V(\{1,\ldots,n\},0)$. The worst case computation time required by this algorithm can be established as follows. There are at most $O(n^3)$ subsets $J(j,l,k)$ and $\sum p_j$ points in time t. There are therefore at most $O(n^3 \sum p_j)$ recursive equations to be solved in the dynamic programming algorithm. As each recursive equation takes $O(n)$ time, the overall running time of the algorithm is bounded by $O(n^4 \sum p_j)$, which is clearly polynomial in n. However, because of the term $\sum p_j$ it qualifies only as a pseudopolynomial time algorithm.

Example 3.4.5 (Minimizing Total Tardiness)
Consider the following 5 jobs.

jobs	1	2	3	4	5
p_j	121	79	147	83	130
d_j	260	266	266	336	337

The job with the largest processing time is job 3. So $0 \leq \delta \leq 2$. The recursive equation yields:

$$V(\{1,2,\ldots,5\},0) = \min \begin{cases} V(J(1,3,3),0) + 81 + V(J(4,5,3),347) \\ V(J(1,4,3),0) + 164 + V(J(5,5,3),430) \\ V(J(1,5,3),0) + 294 + V(\emptyset,560) \end{cases}$$

The optimal sequences of the smaller sets can be determined easily. Clearly, $V(J(1,3,3),0)$ is zero and there are two sequences that yield zero: $1,2$ and $2,1$. The value of

$$V(J(4,5,3),347) = 94 + 223 = 317$$

and this is achieved with sequence $4,5$. Also

$$V(J(1,4,3),0) = 0.$$

This value is achieved with the sequences $1,2,4$ and $2,1,4$. The value of $V(J(5,5,3),430)$ is equal to 560 minus 337 which is 223. Finally,

$$V(J(1,5,3),0) = 76.$$

This value is achieved with sequences $1,2,4,5$ and $2,1,4,5$.

$$V(\{1,2,\ldots,5\},0) = \min \left\{ \begin{matrix} 0 + 81 + 317 \\ 0 + 164 + 223 \\ 76 + 294 + 0 \end{matrix} \right\} = 370.$$

Two optimal sequences are $1,2,4,5,3$ and $2,1,4,5,3$. ||

The $1 \mid\mid \sum T_j$ problem can also be solved with a branch-and-bound procedure. As this branch-and-bound procedure can also be applied to the more general problem with arbitrary weights, it is presented in Section 3.6.

3.5 The Total Tardiness - An Approximation Scheme

Since $1 \mid\mid \sum T_j$ is NP-hard, neither branch-and-bound nor dynamic programming can yield an optimal solution in polynomial time. It may therefore be of interest to have an algorithm that yields, in polynomial time, a solution that is close to optimal.

An approximation scheme A is called fully polynomial if the value of the objective it achieves, say $\sum T_j(A)$, satisfies

$$\sum T_j(A) \;\leq\; (1+\epsilon) \sum T_j(OPT),$$

where $\sum T_j(OPT)$ is the value of the objective under an optimal schedule. Moreover, for the approximation scheme to be fully polynomial its worst case running time has to be bounded by a polynomial of a fixed degree in n and in $1/\epsilon$. The remainder of this section discusses how the dynamic programming algorithm described in the previous section can be used to construct a Fully Polynomial Time Approximation Scheme (FPTAS).

It can be shown that a given set of n jobs can only be scheduled with zero total tardiness if and only if the EDD schedule results in a zero total tardiness. Let $\sum T_j(EDD)$ denote the total tardiness under the EDD sequence and $T_{\max}(EDD)$ the maximum tardiness, i.e., $\max(T_1, \ldots, T_n)$, under the EDD sequence. Clearly,

$$T_{\max}(EDD) \;\leq\; \sum T_j(OPT) \;\leq\; \sum T_j(EDD) \;\leq\; nT_{\max}(EDD).$$

Let $V(J,t)$ denote the minimum total tardiness of the subset of jobs J, which starts processing at time t. For any given subset J, a time t^* can be computed such that $V(J,t) = 0$ for $t \leq t^*$, and $V(J,t) > 0$ for $t > t^*$. Moreover, it can be shown easily that

$$V(J, t^* + \delta) \geq \delta,$$

for $\delta \geq 0$. So in executing the pseudopolynomial dynamic programming algorithm described before, one only has to compute $V(J,t)$ for

$$t^* \;\leq\; t \;\leq\; n\, T_{\max}(EDD).$$

Substituting $\sum p_j$ in the overall running time of the dynamic programming algorithm by $nT_{\max}(EDD)$ yields a new running time bound of $O(n^5 T_{\max}(EDD))$.

Now replace the given processing times p_j by the rescaled processing times

$$p'_j = \lfloor p_j/K \rfloor,$$

where K is a suitable chosen scaling factor. (This implies that p'_j is the largest integer that is smaller than or equal to p_j/K.) Replace the due dates d_j by new due dates

$$d'_j = d_j/K$$

(but without rounding). Consider an optimal sequence with respect to the rescaled processing times and the rescaled due dates and call this sequence $\mathcal{S}$. This sequence can be obtained within the time bound $O(n^5 T_{\max}(EDD)/K)$.

Let $\sum T^*_j(\mathcal{S})$ denote the total tardiness under sequence $\mathcal{S}$ with respect to the processing times Kp'_j and the original due dates and let $\sum T_j(\mathcal{S})$ denote the total tardiness with respect to the original processing times p_j (which may be slightly larger than Kp'_j) and the original due dates. From the fact that

$$Kp'_j \leq p_j < K(p'_j + 1),$$

it follows that

$$\sum T_j^*(\mathcal{S}) \;\le\; \sum T_j(OPT) \;\le\; \sum T_j(\mathcal{S}) \;<\; \sum T_j^*(\mathcal{S}) + K\Big(\frac{n(n+1)}{2}\Big).$$

From this chain of inequalities it follows that

$$\sum T_j(\mathcal{S}) - \sum T_j(OPT) < K\Big(\frac{n(n+1)}{2}\Big).$$

Recall that the goal is for $\mathcal{S}$ to satisfy

$$\sum T_j(\mathcal{S}) - \sum T_j(OPT) \le \epsilon \sum T_j(OPT).$$

If K is chosen such that

$$K = \Big(\frac{2\epsilon}{n(n+1)}\Big) T_{\max}(EDD),$$

then the stronger result

$$\sum T_j(\mathcal{S}) - \sum T_j(OPT) \le \epsilon\, T_{\max}(EDD)$$

is obtained. Moreover, for this choice of K the time bound $O(n^5 T_{\max}(EDD)/K)$ becomes $O(n^7/\epsilon)$, making the approximation scheme fully polynomial.

This Fully Polynomial Time Approximation Scheme can be summarized as follows:

Algorithm 3.5.1 (FPTAS for Minimizing Total Tardiness)

Step 1.

Apply EDD and determine $T_{\max}$.
If $T_{\max} = 0$, *then* $\sum T_j = 0$ *and EDD is optimal; STOP.*
Otherwise set

$$K = \Big(\frac{2\epsilon}{n(n+1)}\Big) T_{\max}(EDD).$$

Step 2.

Rescale processing times and due dates as follows:

$$p'_j = \lfloor\, p_j/K \rfloor,$$
$$d'_j = d_j/K.$$

Step 3.

Apply Algorithm 3.4.4 to the rescaled data. ||

The sequence generated by this algorithm, say sequence $\mathcal{S}$, satisfies

$$\sum T_j(\mathcal{S}) \le (1+\epsilon) \sum T_j(OPT).$$

The following example illustrates the approximation scheme.

Example 3.5.2. (FPTAS Minimizing Total Tardiness)
Consider a single machine and 5 jobs.

jobs	1	2	3	4	5
p_j	1210	790	1470	830	1300
d_j	1996	2000	2660	3360	3370

It can be verified (via dynamic programming) that the optimal sequence is $1, 2, 4, 5, 3$, and that the total tardiness under this optimal sequence is 3700.

Applying EDD yields $T_{\max}(EDD) = 2230$. If ϵ is chosen 0.02, then $K = 2.973$. The rescaled data are:

jobs	1	2	3	4	5
p_j	406	265	494	279	437
d_j	671.38	672.72	894.72	1130.17	1133.54

Solving this instance using the dynamic programming procedure described in Section 3.4 yields two optimal sequences: 1,2,4,5,3 and 2,1,4,5,3. If sequence 2,1,4,5,3 is applied to the original data set, then the total tardiness is 3704. Clearly,

$$\sum T_j(2, 1, 4, 5, 3) \leq (1.02) \sum T_j(1, 2, 4, 5, 3). \qquad ||$$

3.6 The Total Weighted Tardiness

The problem $1 \,||\, \sum w_j T_j$ is an important generalization of the $1 \,||\, \sum T_j$ problem discussed in the previous sections. Dozens of researchers have worked on this problem and have experimented with many different approaches. The approaches range from very sophisticated computer intensive techniques to fairly crude heuristics designed primarily for implementation in practice.

The dynamic programming algorithm for $1 \,||\, \sum T_j$ described in the previous section can also deal with agreeable weights, that is, $p_j \geq p_k \Longrightarrow w_j \leq w_k$. Lemma 3.4.1 can be generalized to this case as follows:

Lemma 3.6.1. *If there are two jobs j and k with $d_j \leq d_k$, $p_j \leq p_k$ and $w_j \geq w_k$, then there is an optimal sequence in which job j appears before job k.*

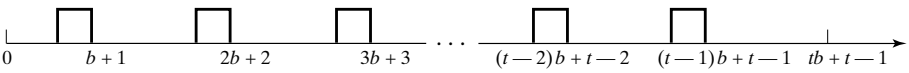


Fig. 3.6 3-PARTITION reduces to $1 \mid\mid \sum w_j T_j$

Proof. The proof is based on a (not necessarily adjacent) pairwise interchange argument. □

Unfortunately, no efficient algorithm can be obtained for $1 \mid\mid \sum w_j T_j$ with arbitrary weights.

Theorem 3.6.2. *The problem* $1 \mid\mid \sum w_j T_j$ *is strongly NP-hard.*

Proof. The proof is done again by reducing 3-PARTITION to $1 \mid\mid \sum w_j T_j$. The reduction is based on the following transformation. Again, the number of jobs, n, is chosen to be equal to $4t - 1$ and

$$\begin{array}{llll} d_j = 0, & p_j = a_j, \ w_j = a_j, & j = 1, \ldots, 3t, \\ d_j = (j - 3t)(b + 1), & p_j = 1, \quad w_j = 2, & j = 3t + 1, \ldots, 4t - 1. \end{array}$$

Let

$$z = \sum_{1 \le j \le k \le 3t} a_j a_k + \frac{1}{2}(t - 1)tb.$$

It can be shown that there exists a schedule with an objective value z if and only if there exists a solution for the 3-PARTITION problem. The first $3t$ jobs have a w_j/p_j ratio equal to 1 and are due at time 0. There are $t - 1$ jobs with w_j/p_j ratio equal to 2 and their due dates are at $b + 1$, $2b + 2$, and so on. A solution with value z can be obtained if these $t-1$ jobs can be processed exactly during the intervals

$$[b,\ b + 1]\ , \quad [2b + 1,\ 2b + 2]\ , \quad \ldots, \quad [(t - 1)b + t - 2,\ (t - 1)b + t - 1]$$

(see Figure 3.6). In order to fit these $t - 1$ jobs in these $t - 1$ intervals, the first $3t$ jobs have to be partitioned into t subsets of three jobs each with the sum of the three processing times in each subset being equal to b. It can be verified that in this case the sum of the weighted tardinesses is equal to z.

If such a partition is not possible, then there is at least one subset of which the sum of the three processing times is larger than b and one other subset of which the sum of the three processing times is smaller than b. It can be verified that in this case the sum of the weighted tardinesses is larger than z. □

Usually a branch-and-bound approach is used for $1 \mid\mid \sum w_j T_j$. Most often, schedules are constructed starting from the end, i.e., backwards in time. At the jth level of the search tree, jobs are put into the $(n - j + 1)$th position. So from each node at level $j - 1$ there are $n - j + 1$ branches going to level j. It may not

be necessary to evaluate *all* possible nodes. Dominance results such as the one described in Lemma 3.6.1 may eliminate a number of nodes. The upper bound on the number of nodes at level j is $n!/(n-j)!$ The argument for constructing the sequence backwards is that the larger terms in the objective function are likely to correspond to jobs that are positioned more towards the end of the schedule. It appears to be advantageous to schedule these ones first.

There are many different bounding techniques. One of the more elementary bounding techniques is based on a *relaxation* of the problem to a transportation problem. In this procedure each job j with (integer) processing time p_j is divided into p_j jobs, each with unit processing time. The decision variables x_{jk} is 1 if one unit of job j is processed during the time interval $[k-1, k]$ and 0 otherwise. These decision variables x_{jk} must satisfy two sets of constraints:

$$\sum_{k=1}^{C_{\max}} x_{jk} = p_j, \qquad j = 1, \ldots, n$$

$$\sum_{j=1}^{n} x_{jk} = 1, \qquad k = 1, \ldots, C_{\max}.$$

Clearly, a solution satisfying these constraints does not guarantee a feasible schedule without preemptions. Define cost coefficients c_{jk} that satisfy

$$\sum_{k=l-p_j+1}^{l} c_{jk} \leq w_j \max(l - d_j, 0)$$

for $j = 1, \ldots, n; \quad l = 1, \ldots, C_{\max}$. Then the minimum cost solution provides a lower bound, since for any solution of the transportation problem with $x_{jk} = 1$ for $k = C_j - p_j + 1, \ldots, C_j$ the following holds

$$\sum_{k=1}^{C_{\max}} c_{jk} x_{jk} = \sum_{k=C_j-p_j+1}^{C_j} c_{jk} \leq w_j \max(C_j - d_j, 0).$$

It is fairly easy to find cost functions that satisfy this relationship. For example, set

$$c_{jk} = \begin{cases} 0, & \text{for } k \leq d_j \\ w_j, & \text{for } k > d_j. \end{cases}$$

The solution of the transportation problem provides a lower bound for $1 \mid\mid \sum w_j T_j$. This bounding technique is applied to the set of unscheduled jobs at each node of the tree. If the lower bound is larger than the solution of any known schedule, then the node may be eliminated.

Example 3.6.3 (Minimizing Total Weighted Tardiness)
Consider the following 4 jobs.

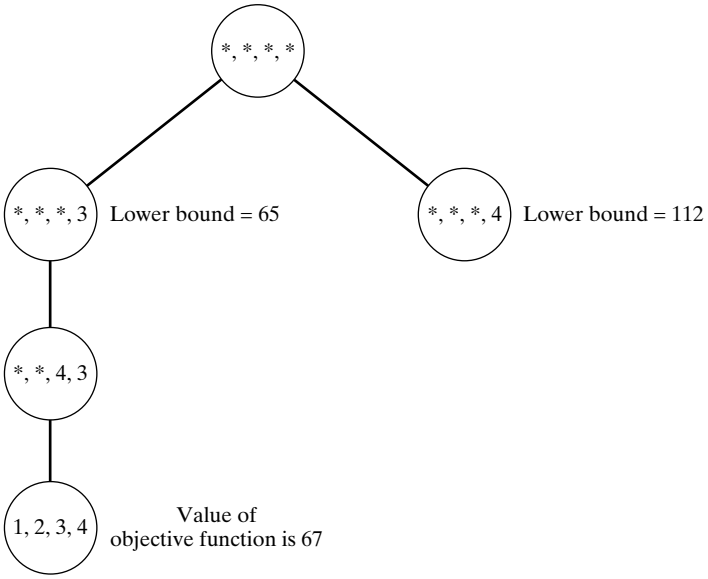


Fig. 3.7 Branch-and-bound procedure for Example 3.5.3

jobs	1	2	3	4
w_j	4	5	3	5
p_j	12	8	15	9
d_j	16	26	25	27

From Lemma 3.6.1 it immediately follows that in an optimal sequence job 4 follows job 2 and job 3 follows job 1. The branch-and-bound tree is constructed backwards in time. Only two jobs have to be considered as candidates for the last position, namely jobs 3 and 4. The nodes of the branch-and-bound tree that need to be investigated are depicted in Figure 3.7. To select a branch to search first, bounds are determined for both nodes at level 1.

A lower bound for an optimal sequence among the offspring of node $(*, *, *, 4)$ can be obtained by considering the transportation problem described before applied to jobs 1, 2 and 3. The cost functions are chosen as follows

$$\begin{aligned} c_{1k} &= 0, && k = 1, \ldots, 16 \\ c_{1k} &= 4, && k = 17, \ldots, 35 \\ c_{2k} &= 0, && k = 1, \ldots, 26 \end{aligned}$$

$$c_{2k} = 5, \qquad k = 27, \ldots, 35$$
$$c_{3k} = 0, \qquad k = 1, \ldots, 25$$
$$c_{3k} = 3, \qquad k = 26, \ldots, 35$$

The optimal allocation of job segments to time slots puts job 1 in the first 12 slots, job 2 into slots 19 to 26 and job 3 in slots 13 to 18 and 27 to 35 (this optimal solution can be found by solving a transportation problem but can, of course, also be found by trial and error). The cost of this allocation of the three jobs is 3×9 (the cost of allocating job 3 to slots 27 to 35). In order to obtain a lower bound for the node the tardiness of job 4 has to be added; this results in the lower bound $27 + 80$ which equals 107.

In a similar fashion a lower bound can be obtained for node $(*, *, *, 3)$. A lower bound for an optimal schedule for jobs 1, 2 and 4 yields 8, while the tardiness of job 3 is 54 resulting in a bound of 62.

As node $(*, *, *, 3)$ appears to be the more promising node, the offspring of this node is considered first. It turns out that the best schedule reachable from this node is $1, 2, 4, 3$ with an objective value of 64.

From the fact that the lower bound for $(*, *, *, 4)$ is 107 it follows that $1, 2, 4, 3$ is the best overall schedule. ||

There are many heuristic procedures for this problem. Chapter 14 describes a composite dispatching rule, the so-called *Apparent Tardiness Cost (ATC)* rule, in detail.

3.7 Discussion

All the models considered in this chapter have regular objective functions. This is one of the reasons why most of the models are relatively easy.

Some are solvable via simple priority (dispatching) rules, e.g., WSPT, EDD. Most of the models that are not solvable via simple priority rules, are still solvable either in polynomial time or in pseudo-polynomial time. The models that are solvable in polynomial time are usually dealt with through dynamic programming, e.g., $1 \mid prec \mid h_{\max}, \quad 1 \mid\mid \sum T_j$.

One of the strongly NP-hard problems considered in this chapter is $1 \mid\mid \sum w_j T_j$. This problem has received an enormous amount of attention in the literature. There are two approaches for obtaining optimal solutions, namely branch-and-bound, and dynamic programming. Section 3.6 presents a branch-and-bound approach, while Appendix B describes a dynamic programming approach that can be applied to the more general problem $1 \mid\mid \sum h_j(C_j)$.

This chapter has also shown an application of a Fully Polynomial Time Approximation Scheme (FPTAS) for a single machine scheduling problem. Over the last decade Polynomial Time Approximation Schemes (PTAS) and Fully Polynomial Time Approximation Schemes (FPTAS) have received an enormous amount of attention. Most of this attention has focused on NP-hard problems

that are close to the boundaries separating NP-hard problems from polynomial time problems, e.g., $1 \mid r_j \mid \sum C_j$.

Most of the problems described in this chapter can be formulated as Mixed Integer Programs (MIPs). Mixed Integer Programming formulations of several single machine scheduling problems are presented in Appendix A. This appendix gives also an overview of the techniques that can be applied to MIPs.

This chapter does not exhibit all the possible procedures and techniques that can be brought to bear on single machine scheduling problems. One important class of solution procedures is often referred to as constraint programming. Appendix C gives a detailed description of this class of procedures and Chapter 15 provides an example of a constraint programming procedure that can be applied to $1 \mid r_j \mid \sum w_j U_j$.

Many heuristic procedures have been developed that can be applied to single machine scheduling problems. These procedures include the so-called composite dispatching rules as well as local search techniques. Chapter 14 provides an in-depth overview of these techniques and their applications to single machine problems.

The next chapter considers more general and more complicated single machine problems. It focuses on problems with non-regular objective functions and on problems with multiple objective functions.

Exercises (Computational)

3.1. Consider $1 \mid\mid \sum w_j C_j$ with the following weights and processing times.

jobs	1	2	3	4	5	6	7
w_j	0	18	12	8	8	17	16
p_j	3	6	6	5	4	8	9

(a) Find *all* optimal sequences.

(b) Determine the effect of a change in p_2 from 6 to 7 on the optimal sequence(s).

(c) Determine the effect of the change under (b) on the value of the objective.

3.2. Consider $1 \mid \mathit{chains} \mid \sum w_j C_j$ with the same set of jobs as in Exercise 3.1.(a). The jobs are now subject to precedence constraints which take the form of chains:

$$1 \rightarrow 2$$

$$3 \rightarrow 4 \rightarrow 5$$

$$6 \rightarrow 7$$

Find all optimal sequences.

3.3. Consider $1 \mid\mid \sum w_j(1-e^{-rC_j})$ with the same set of jobs as in Exercise 3.1.

(a) Assume the discount rate r is 0.05. Find the optimal sequence. Is it unique?

(b) Assume the discount rate r is 0.5. Does the optimal sequence change?

3.4. Find all optimal sequences for the instance of $1 \mid\mid h_{\max}$ with the following jobs.

jobs	1	2	3	4	5	6	7
p_j	4	8	12	7	6	9	9
$h_j(C_j)$	$3\,C_1$	77	C_3^2	$1.5\,C_4$	$70+\sqrt{C_5}$	$1.6\,C_6$	$1.4\,C_7$

3.5. Consider $1 \mid prec \mid h_{\max}$ with the same set of jobs as in Exercise 3.4 and the following precedence constraints.

$$1 \rightarrow 7 \rightarrow 6$$

$$5 \rightarrow 7$$

$$5 \rightarrow 4$$

Find the optimal sequence.

3.6. Solve by branch-and-bound the following instance of the $1 \mid r_j \mid L_{\max}$ problem.

jobs	1	2	3	4	5	6	7
p_j	6	18	12	10	10	17	16
r_j	0	0	0	14	25	25	50
d_j	8	42	44	24	90	85	68

3.7. Consider the same problem as in the previous exercise. However, now the jobs are subject to the following precedence constraints.

$$2 \rightarrow 1 \rightarrow 4$$

$$6 \rightarrow 7$$

Find the optimal sequence.

3.8. Find the optimal sequence for the following instance of the $1 \mid\mid \sum T_j$ problem.

jobs	1	2	3	4	5	6	7	8
p_j	6	18	12	10	10	11	5	7
d_j	8	42	44	24	26	26	70	75

Hint: Before applying the dynamic programming algorithm, consider first the elimination criterion in Lemma 3.4.1.

3.9. Consider a single machine and 6 jobs.

jobs	1	2	3	4	5	6
p_j	1190	810	1565	719	1290	482
d_j	1996	2000	2660	3360	3370	3375

Apply the FPTAS described in Section 3.5 to this instance with $\epsilon = 0.02$. Are all sequences that are optimal for the rescaled data set also optimal for the original data set?

3.10. Find the optimal sequence for the following instance of the $1 \mid\mid \sum w_j T_j$ problem.

jobs	1	2	3	4	5	6	7
p_j	6	18	12	10	10	17	16
w_j	1	5	2	4	1	4	2
d_j	8	42	44	24	90	85	68

Exercises (Theory)

3.11. Consider $1 \mid\mid \sum w_j(1-e^{-rC_j})$. Assume that $w_j/p_j \neq w_k/p_k$ for all j and k. Show that for r sufficiently close to zero the optimal sequence is WSPT.

3.12. Show that if all jobs have equal weights, i.e., $w_j = 1$ for all j, the WDSPT rule is equivalent to the *Shortest Processing Time first* (*SPT*) rule for any r, $0 < r < 1$.

3.13. Consider the problem $1 \mid prmp \mid \sum h_j(C_j)$. Show that if the functions h_j are *nondecreasing* there exists an optimal schedule that is nonpreemptive. Does the result continue to hold for arbitrary functions h_j?

3.14. Consider the problem $1 \mid r_j \mid \sum C_j$.

(a) Show through a counterexample that the nonpreemptive rule that selects, whenever a machine is freed, the shortest job among those available for processing is not always optimal. In part (b) and (c) this rule is referred to as SPT*.

(b) Perform a worst case analysis of the SPT* rule, i.e., determine the maximum possible value of the ratio $\sum C_j(SPT^*)/\sum C_j(OPT)$.

(c) Design a heuristic for $1 \mid r_j \mid C_j$ that performs better than SPT*.

3.15. Consider the problem $1 \mid r_j, prmp \mid \sum C_j$. Show that the preemptive *Shortest Remaining Processing Time first (SRPT)* rule is optimal.

3.16. Consider the problem $1 \mid prmp \mid \sum C_j$ with the additional restriction that job j *has to* be completed by a hard deadline $\bar{d}_j$. Assuming that there are feasible schedules, give an algorithm that minimizes the total completion time and prove that it leads to optimality.

3.17. Consider the following preemptive version of the WSPT rule: if $p_j(t)$ denotes the *remaining* processing time of job j at time t, then a preemptive version of the WSPT rule puts at every point in time the job with the highest $w_j/p_j(t)$ ratio on the machine. Show, through a counterexample, that this rule is not necessarily optimal for $1 \mid r_j, prmp \mid \sum w_j C_j$.

3.18. Give an algorithm for $1 \mid intree \mid \sum w_j C_j$ and prove that it leads to an optimal schedule (recall that in an intree each job has at most one successor).

3.19. Give an algorithm for $1 \mid outtree \mid \sum w_j C_j$ and show that it leads to an optimal schedule (recall that in an outtree each job has at most one predecessor).

3.20. Consider the problem $1 \mid\mid L_{\max}$. The *Minimum Slack first (MS)* rule selects at time t, when a machine is freed, among the remaining jobs the job with the minimum slack $\max(d_j - p_j - t, 0)$. Show through a counterexample that this rule is not necessarily optimal.

3.21. Perform an Adjacent Sequence Interchange for the weighted discounted flow time cost function. That is, state and prove a result similar to Lemma 3.1.2.

3.22. Consider the problem $1 \mid chains \mid \sum w_j(1 - e^{-rC_j})$. Describe the algorithm that solves this problem and prove that it results in an optimal sequence.

3.23. Consider the problem $1 \mid prec \mid \max(h_1(S_1), \ldots, h_n(S_n))$, where S_j denotes the starting time of job j. The cost function h_j, $j = 1, \ldots, n$ is *decreasing*. Unforced idleness of the machine is *not* allowed. Describe a dynamic programming type algorithm for this problem similar to the one in Section 3.2. Why does one have to use here forward dynamic programming instead of backward dynamic programming?

3.24. Consider the problem $1 \mid r_j, prmp \mid L_{\max}$. Determine the optimal schedule and prove its optimality.

3.25. Show that

(a) SPT is optimal for $1 \mid brkdwn \mid \sum C_j$,
(b) Algorithm 3.3.1 is optimal for $1 \mid brkdwn \mid \sum U_j$,
(c) WSPT is not necessarily optimal for $1 \mid brkdwn \mid \sum w_j C_j$.

3.26. Consider $1 \mid\mid \sum w_j T_j$. Prove or disprove the following statement: If

$$w_j/p_j > w_k/p_k,$$

$$p_j < p_k,$$

and

$$d_j < d_k,$$

then there exists an optimal sequence in which job j appears before job k.

3.27. Complete the first step of the proof of Theorem 3.3.2.

Comments and References

The optimality of the WSPT rule for $1 \mid\mid \sum w_j C_j$ appears in the seminal paper by W.E. Smith (1956). Lawler (1978), Monma and Sidney (1979, 1987), Möhring and Radermacher (1985a) and Sidney and Steiner (1986) all present very elegant results for $1 \mid prec \mid \sum w_j C_j$; the classes of precedence constraints considered in these papers is fairly general and includes chains as well as intrees and outtrees. The $1 \mid r_j, prmp \mid \sum C_j$ problem has been analyzed by Schrage (1968). The complexity proof for $1 \mid r_j, prmp \mid \sum w_j C_j$ is due to Labetoulle, Lawler, Lenstra and Rinnooy Kan (1984). Rothkopf (1966a, 1966b) and Rothkopf and Smith (1984) analyze $1 \mid\mid \sum w_j(1 - e^{-rC_j})$.

The EDD rule is due to Jackson (1955) and the algorithm for $1 \mid prec \mid h_{\max}$ is due to Lawler (1973). The complexity proof for $1 \mid r_j \mid L_{\max}$ appears in Lenstra, Rinnooy Kan and Brucker (1977). Many researchers have worked on branch-and-bound methods for $1 \mid r_j \mid L_{\max}$; see, for example, McMahon and Florian (1975), Carlier (1982) and Nowicki and Zdrzalka (1986). Potts (1980) analyzes a heuristic for $1 \mid r_j \mid L_{\max}$.

Algorithm 3.3.1, which minimizes the number of late jobs, is from Moore (1968). Kise, Ibaraki and Mine (1978) consider the $1 \mid r_j \mid \sum U_j$ problem. The NP-hardness of $1 \mid\mid \sum w_j U_j$ (i.e., the knapsack problem) is established in the classic paper by Karp (1972) on computational complexity. Sahni (1976) presents a pseudopolynomial time algorithm for this problem and Gens and Levner (1981) and Ibarra and Kim (1978) provide fast approximation algorithms. Potts and Van Wassenhove (1988) give a very efficient algorithm for a Linear Programming relaxation of the Knapsack problem. Van den Akker and Hoogeveen (2004) give an in-depth overview of scheduling problems with the $\sum w_j U_j$ objective. (A problem related to the knapsack problem is the so-called due date assignment problem. This problem has received a lot of attention as well; see Panwalkar, Smith and Seidmann (1982) and Cheng and Gupta (1989).)

The dominance condition in Lemma 3.4.1 is due to Emmons (1969) and the pseudo-polynomial time Algorithm 3.4.4 is from Lawler (1977). The NP-hardness of $1 \mid\mid \sum T_j$ is shown by Du and Leung (1990). For additional work

on dynamic programming and other approaches for this problem, see Potts and van Wassenhove (1982, 1987).

An enormous amount of work has been done on Polynomial Time Approximation Schemes (PTAS) and on Fully Polynomial Time Approximation Schemes (FPTAS). The algorithm described in Section 3.5 is one of the very first schemes developed for scheduling problems. This section is based entirely on the paper by Lawler (1982). A significant amount of work has been done on approximation algorithms for $1 \mid r_j \mid \sum C_j$; see, for example, Chekuri, Motwani, Natarajan, and Stein (1997). There are many other interesting applications of PTAS to scheduling problems; see, for example, Hochbaum and Shmoys (1987), Sevastianov and Woeginger (1998), and Alon, Azar, Woeginger and Yadid (1998). Schuurman and Woeginger (1999) present in their paper ten open problems concerning PTAS; their paper also contains an extensive reference list of PTAS papers. For a general overview of approximation techniques (covering more than just PTAS and FPTAS), see Chen, Potts and Woeginger (1998).

The complexity of $1 \mid\mid \sum w_j T_j$ is established in the Lawler (1977) paper as well as in the Lenstra, Rinnooy Kan and Brucker (1977) paper. Branch-and-bound methods using bounding techniques based on relaxations to transportation problems are discussed in Gelders and Kleindorfer (1974, 1975). Many other approaches have been suggested for $1 \mid\mid \sum w_j T_j$, see for example Fisher (1976, 1981), Potts and van Wassenhove (1985) and Rachamadugu (1987).

Chapter 4

Advanced Single Machine Models (Deterministic)

This chapter covers several more advanced topics in single machine scheduling. Some of these topics are important because of the theoretical insights they provide, others are important because of their applications in practice.

The first section considers a generalization of the total tardiness problem. In addition to tardiness costs, there are now also earliness costs; the objective functions are nonregular. The second section focuses on problems with a primary objective and a secondary objective. The goal is to first determine the set of all schedules that are optimal with respect to the primary objective; within this set of schedules a schedule has to be found then that is optimal with respect to the secondary objective. The third section also focuses on problems with two objectives. However, now the two objectives have to be considered simultaneously with the weights of the objectives being arbitrary. The overall objective is to minimize the weighted sum of the two objectives. The next section considers the makespan when there are sequence dependent setup times. There are two reasons for not having considered the makespan before. First, in most single machine environments the makespan does not depend on the sequence and is therefore not that important. Second, when there are sequence dependent setup times, the algorithms for minimizing the makespan tend to be complicated. The fifth section also considers sequence dependent setup times. However, now the jobs belong to a fixed number of different families. If in a

M.L. Pinedo, *Scheduling*, DOI: 10.1007/978-0-387-78935-4_4,

schedule a job is followed by a job from a different family, then a sequence dependent setup time is incurred; if a job is followed by another job from the same family, then no setup is incurred. A number of dynamic programming approaches are described for various different objective functions. The sixth section focuses on batch processing. The machine can process now a number of jobs (a batch) simultaneously. The jobs processed in a batch may have different processing times and the time to process the batch is determined by the longest processing time. Various different objective functions are considered.

4.1 The Total Earliness and Tardiness

All objective functions considered in Chapter 3 are regular performance measures (i.e., nondecreasing in C_j for all j). In practice, it may occur that if job j is completed before its due date d_j an earliness penalty is incurred. The earliness of job j is defined as

$$E_j = \max(d_j - C_j, 0).$$

The objective function in this section is a generalization of the total tardiness objective. It is the sum of the total earliness and the total tardiness, i.e.,

$$\sum_{j=1}^{n} E_j + \sum_{j=1}^{n} T_j.$$

Since this problem is harder than the total tardiness problem it makes sense to first analyze special cases that are tractable. Consider the special case with all jobs having the same due date, i.e., $d_j = d$ for all j.

An optimal schedule for this special case has a number of useful properties. For example, it can be shown easily that after the first job is started, the n jobs have to be processed without interruption, i.e., there should be no unforced idleness in between the processing of any two consecutive jobs (see Exercise 4.11). However, it is possible that an optimal schedule does not start processing the jobs immediately at time 0; it may wait for some time before it starts with its first job.

A second property concerns the actual sequence of the jobs. Any sequence can be partitioned into two disjoint sets of jobs and possibly one additional job. One set contains the jobs that are completed early, i.e., $C_j \leq d$, and the other set contains the jobs that are started late. The first set of jobs is called J_1 and the second set of jobs J_2. In addition to these two sets of jobs, there may be one more job that is started early and completed late.

Lemma 4.1.1. *In an optimal schedule the jobs in set J_1 are scheduled first according to LPT and the jobs in set J_2 are scheduled last according to SPT. In between these two sets of jobs there may be one job that is started early and completed late.*

Proof. The proof is easy and left as an exercise (see Exercise 4.12). □

Because of the property described in Lemma 4.1.1, it is often said that the optimal schedule has a V shape.

Consider an instance with the property that no optimal schedule starts processing its first job at $t = 0$, i.e., the due date d is somewhat loose and the machine remains idle for some time before it starts processing its first job. If this is the case, then the following property holds.

Lemma 4.1.2. *There exists an optimal schedule in which one job is completed exactly at time d.*

Proof. The proof is by contradiction. Suppose there is no such schedule. Then there is always one job that starts its processing before d and completes its processing after d. Call this job j^*. Let $|J_1|$ denote the number of jobs that are early and $|J_2|$ the number of jobs that are late. If $|J_1| < |J_2|$, then shift the entire schedule to the left in such a way that job j^* completes its processing exactly at time d. This implies that the total tardiness decreases by $|J_2|$ times the length of the shift, while the total earliness increases by $|J_1|$ times the shift. So, clearly, the total earliness plus the total tardiness is reduced. The case $|J_1| > |J_2|$ can be treated in a similar way.

The case $|J_1| = |J_2|$ is somewhat special. In this case there are many optimal schedules, of which only two satisfy the property stated in the lemma. □

For an instance in which all optimal schedules start processing the first job some time after $t = 0$, the following algorithm yields the optimal allocations of jobs to sets J_1 and J_2. Assume $p_1 \geq p_2 \geq \cdots \geq p_n$.

Algorithm 4.1.3 (Minimizing Total Earliness and Tardiness with Loose Due Date)

Step 1.

Assign job 1 to Set J_1.
Initialize $k = 2$.

Step 2.

Assign job k to Set J_1 and job $k + 1$ to Set J_2 or vice versa.

Step 3.

If $k + 2 \leq n - 1$, increase k by 2 and go to Step 2.
If $k + 2 = n$, assign job n to either Set J_1 or Set J_2 and STOP.
If $k + 2 = n + 1$, then all jobs have been assigned; STOP. ||

This algorithm is somewhat flexible in its assignment of jobs to sets J_1 and J_2. It can be implemented in such a way that in the optimal assignment the total processing time of the jobs assigned to J_1 is minimized. Given the total processing time of the jobs in J_1 and the due date d, it can be verified easily

whether the machine indeed must remain idle before it starts processing its first job.

If the due date d is tight and it is necessary to start processing a job immediately at time zero, then the problem is NP-hard. However, the following heuristic, which assigns the n jobs to the n positions in the sequence, is very effective. Assume again $p_1 \geq p_2 \geq \cdots \geq p_n$.

Algorithm 4.1.4 (Minimizing Total Earliness and Tardiness with Tight Due Date)

Step 1.

Initialize $\tau_1 = d$ *and* $\tau_2 = \sum p_j - d$.
Initialize $k = 1$.

Step 2.

If $\tau_1 > \tau_2$, *assign job* k *to the first unfilled position in the sequence and decrease* τ_1 *by* p_k.
If $\tau_1 < \tau_2$, *assign job* k *to the last unfilled position in the sequence and decrease* τ_2 *by* p_k.

Step 3.

If $k < n$, *increase* k *by 1 and go to Step 2.*
If $k = n$, *STOP.* ||

Example 4.1.5 (Minimizing Total Earliness and Tardiness with Tight Due Date)

Consider the following example with 6 jobs and $d = 180$.

jobs	1	2	3	4	5	6
p_j	106	100	96	22	20	2

Applying the heuristic yields the following results.

τ_1	τ_2	*Assignment*	*Sequence*
180	166	Job 1 Placed First	1,*,*,*,*,*
74	166	Job 2 Placed Last	1,*,*,*,*,2
74	66	Job 3 Placed First	1,3,*,*,*,2
-22	66	Job 4 Placed Last	1,3,*,*,4,2
-22	44	Job 5 Placed Last	1,3,*,5,4,2
-22	12	Job 6 Placed Last	1,3,6,5,4,2

||

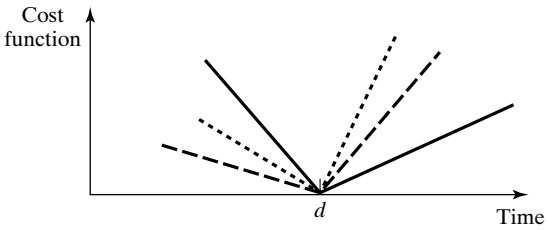


Fig. 4.1 Cost functions with common due date and different shapes

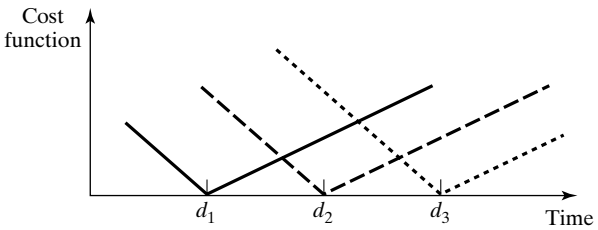


Fig. 4.2 Cost functions with different due dates and similar shapes

Consider now the objective $\sum w'E_j + \sum w''T_j$ and assume again that all the due dates are the same, i.e., $d_j = d$, for all j. All jobs have exactly the same cost function, but the earliness penalty w' and the tardiness penalty w'' are not the same. All previous properties and algorithms can be generalized relatively easily to take the difference between w' and w'' into account (see Exercises 4.13 and 4.14).

Consider the even more general objective $\sum w'_j E_j + \sum w''_j T_j$, with $d_j = d$ for all j. So all jobs have the same due date, but the shapes of their cost functions are different, see Figure 4.1. The LPT-SPT sequence of Lemma 4.1.1 is in this case not necessarily optimal. The first part of the sequence must now be ordered in increasing order of w_j/p_j, i.e., according to Weighted Longest Processing Time first (WLPT) rule, and the last part of the sequence must be ordered according to the Weighted Shortest Processing Time first (WSPT) rule.

Consider the model with the objective function $\sum w'E_j + \sum w''T_j$ and with each job having a different due date (see Figure 4.2). It is clear that this problem is NP-hard, since it is a more general model than the one considered in Section 3.4. This problem has an additional level of complexity. Because of the different due dates, it may not necessarily be optimal to process the jobs one after another without interruption; it may be necessary to have idle times between the processing of consecutive jobs. This problem has therefore two aspects: one aspect concerns the search for an optimal order in which to sequence the jobs and the other aspect concerns the computation of the optimal starting

times and completion times of the jobs. These two optimization problems are clearly not independent. Determining the optimal schedule is therefore a very hard problem. Approaches for dealing with this problem are typically based on dynamic programming or branch-and-bound. However, given a predetermined and fixed sequence, the timing of the processing of the jobs (and therefore also the idle times) can be determined via a relatively simple polynomial time algorithm. This polynomial time algorithm is also applicable in a more general setting that is described next.

The most general setting has as objective $\sum w'_j E_j + \sum w''_j T_j$, where the jobs have different due dates and different weights. This problem is clearly strongly NP-hard, since it is harder than the total weighted tardiness problem considered in Section 3.6. But, given a predetermined ordering of the jobs, the timings of the processings and the idle times can be computed in polynomial time. Some preliminary results are necessary to describe the algorithm that inserts the idle times in a given sequence. Assume that the job sequence $1, \ldots, n$ is fixed.

Lemma 4.1.6. *If $d_{j+1} - d_j \leq p_{j+1}$, then there is no idle time between jobs j and $j+1$.*

Proof. The proof is by contradiction. Consider three cases: Job j is early ($C_j < d_j$), job j is completed exactly at its due date ($C_j = d_j$), and job j is late ($C_j > d_j$).

Case 1: If job j is completed early and there is an idle time between jobs j and $j+1$, then the objective can be reduced by postponing the processing of job j and reducing the idle time. The schedule with the idle time can therefore not be optimal.

Case 2: If job j is completed at its due date and there is an idle time, then job $j+1$ is completed late. Processing job $j+1$ earlier and eliminating the idle time, reduces the total objective. So the original schedule cannot be optimal.

Case 3: If job j is completed late and there is an idle time, then job $j+1$ is also completed late. Processing job $j+1$ earlier reduces the objective. □

Subsequence $u, \ldots, v$ is called a job cluster if for each pair of adjacent jobs j and $j+1$ the inequality

$$d_{j+1} - d_j \leq p_{j+1}$$

holds and if for $j = u-1$ and $j = v$ the inequality does not hold. A cluster of jobs must therefore be processed without interruptions.

Lemma 4.1.7. *In each cluster in a schedule the early jobs precede the tardy jobs. Moreover, if jobs j and $j+1$ belong to the same cluster and are both early, then $E_j \geq E_{j+1}$. If jobs j and $j+1$ are both late then $T_j \leq T_{j+1}$.*

Proof. Assume jobs j and $j+1$ belong to the same cluster. Let t denote the optimal start time of job j. Subtracting $t + p_j$ from both sides of

$$d_{j+1} - d_j \leq p_{j+1}$$

and rearranging yields

$$d_{j+1} - t - p_j - p_{j+1} \leq d_j - t - p_j.$$

This last inequality can be rewritten as

$$d_j - C_j \geq d_{j+1} - C_{j+1},$$

which implies the lemma. □

The given job sequence $1, \ldots, n$ can be decomposed into a set of m clusters $\sigma_1, \sigma_2, \ldots, \sigma_m$ with each cluster representing a subsequence. The algorithm that inserts the idle times starts out with the given sequence $1, \ldots, n$ without idle times. That is, the completion time of the kth job in the sequence is

$$C_k = \sum_{j=1}^{k} p_j.$$

Since the completion times in the original schedule are the earliest possible completion times, the algorithm has to determine how much to increase (or shift) the completion time of each job. In fact, it is sufficient to compute the optimal shift for each cluster since all its jobs are shifted by the same amount.

Consider a cluster σ_r that consists of jobs $k, k+1, \ldots, \ell$. Let

$$\Delta_j = \sum_{l=k}^{j} w'_l - \sum_{l=j+1}^{\ell} w''_l, \qquad j = k, \ldots, \ell.$$

These numbers can be computed recursively by setting

$$\Delta_{k-1} = -\sum_{l=k}^{\ell} w''_l,$$

and

$$\Delta_j = \Delta_{j-1} + w'_j + w''_j, \qquad j = k, \ldots, \ell.$$

Define a block as a sequence of clusters that are processed without interruption. Consider block $\sigma_s, \sigma_{s+1}, \ldots, \sigma_m$. Such a block may have one or more clusters preceding it, but no clusters following it. Let j_r be the last job in cluster σ_r that is early, i.e., the job with the smallest earliness. Let

$$E(r) = E_{j_r} = d_{j_r} - C_{j_r}.$$

Clearly

$$E(r) = \min_{k \leq j \leq j_r} (d_j - C_j).$$

Let

$$\Delta(r) = \Delta_{j_r} = \max_{k \le j \le j_r} \Delta_j.$$

If none of the jobs in cluster σ_r is early, then $E(r) = \infty$ and $\Delta(r) = -\sum_{l=k}^{\ell} w_l''$. If $d_{j_r} - C_{j_r} \geq 1$ for the last early job in every cluster σ_r, $r = s, s+1, \ldots, m$, then a shift of the entire block by one time unit to the right decreases the total cost by

$$\sum_{r=s}^{m} \Delta(r).$$

The idea behind the algorithm is to find the first block of clusters that cannot be shifted. If such a block is found, then this block stays in place and the procedure is repeated for the set of remaining clusters. If no such block is found, then all remaining clusters are shifted by an amount that is equal to the smallest $E(r)$; in one of the shifted clusters the last early job becomes an on-time job. All the completion times are updated and all the non-early jobs are removed from the list of each cluster. The procedure is then repeated. The algorithm terminates once a block involving the last cluster cannot be shifted. The algorithm can be summarized as follows.

Algorithm 4.1.8 (Optimizing the Timings Given a Sequence)

Step 1.

Identify the clusters and compute $\Delta(r)$ for each cluster.

Step 2.

Find the smallest s such that $\sum_{r=1}^{s} \Delta(r) \leq 0$.
Fix the current C_k for each job in the first s clusters.
If $s = m$ then STOP, otherwise go to Step 3.
If no such s exists, then go to Step 4.

Step 3.

Remove the first s clusters from the list.
Reindex all remaining clusters and jobs.
Go to Step 2 to consider the set of remaining clusters.

Step 4.

Find $\min(E(1), \ldots, E(m))$.
Increase all C_k by $\min(E(1), \ldots, E(m))$.
Eliminate all early jobs that are no longer early.
Update $E(r)$ and $\Delta(r)$.
Go to Step 2. ||

Example 4.1.9 (Optimizing the Timings Given a Sequence)

Consider seven jobs. The given sequence is $1, \ldots, 7$.

jobs	1	2	3	4	5	6	7
p_j	3	2	7	3	6	2	8
d_j	12	4	26	18	16	25	30
w'_j	10	20	18	9	10	16	11
w''_j	12	25	38	12	12	18	15

The set of jobs can be decomposed into the three clusters $\sigma_1, \sigma_2, \sigma_3$, where $\sigma_1 = 1, 2$, $\sigma_2 = 3, 4, 5$, and $\sigma_3 = 6, 7$. The initial schedule has job completion times

$$3, 5, \quad 12, 15, 21, \quad 23, 31.$$

The sets of early jobs in clusters σ_1, σ_2, and σ_3 are, respectively, jobs (1), (3,4), and (6). For each one of these jobs the values of $d_j - C_j$ can be computed. Applying Step 1 of the algorithm results in the table presented below.

clusters	1	2	3
$E(r)$	9	3	2
$\Delta(r)$	-15	15	1

Step 2 of the Algorithm yields $s = 1$ and $\Delta(1) < 0$. So cluster σ_1 is not shifted and $C_1 = 3$ and $C_2 = 5$. Step 3 eliminates σ_1. Going back to Step 2 yields $\Delta(2) > 0$, $\Delta(2) + \Delta(3) > 0$ and no s exists. Going to Step 4 results in

$$\min(E(2), E(3)) = \min(3, 2) = 2.$$

Increase all completion times in the second and third cluster by 2 time units and eliminate job 6 from the list of early jobs. Update $d_k - C_k$. The new values of $E(r)$ and $\Delta(r)$ are presented in the table below.

clusters	2	3
$E(r)$	1	∞
$\Delta(r)$	15	-33

Returning to Step 2 yields $\Delta(2) > 0$ and $\Delta(2) + \Delta(3) < 0$. It follows that $s = 3 = m$. So the second and third cluster should not be shifted and the algorithm stops. The optimal completion times are

$$3, 5, 14, 17, 23, 25, 33. \qquad ||$$

As stated earlier, finding at the same time an optimal job sequence as well as the optimal starting times and completion times of the jobs is strongly NP-hard.

A branch-and-bound procedure for this problem is more complicated than the one for the total weighted tardiness problem described in Section 3.6. The branching tree can be constructed in a manner that is similar to the one for the $1 \mid\mid \sum w_j T_j$ problem. However, finding good lower bounds for $1 \mid\mid \sum w'_j E_j + \sum w''_j T_j$ is considerably harder. One type of lower bound can be established by first setting $w'_j = 0$ for all j and then applying the lower bound described in Section 3.6 to the given instance by taking only the tardiness penalties into account. This lower bound may not be that good, since it is based on two simplifications.

It is possible to establish certain dominance conditions. For example, if the due dates of two adjacent jobs both occur before the starting time of the first one of the two jobs, then the job with the higher w''_j/p_j ratio has to go first. Similarly, if the due dates of two adjacent jobs both occur after the completion time of the last one of the two jobs, then the job with the lower w'_j/p_j ratio has to go first.

Many heuristic procedures have been developed for this problem. These procedures are often based on a combination of decomposition and local search. The problem lends itself well to time-based decomposition procedures, since it may be possible to tailor the decomposition process to the clusters and the blocks.

4.2 Primary and Secondary Objectives

In practice a scheduler is often concerned with more than one objective. For example, he may want to minimize inventory costs and meet due dates. It would then be of interest to find, for example, a schedule that minimizes a combination of $\sum C_j$ and $L_{\max}$.

Often, more than one schedule minimizes a given objective. A decision-maker may then wish to consider the set of all schedules that are optimal with respect to such an objective (say, the primary objective), and then search within this set of schedules for the schedule that is best with regard to a secondary objective. If the primary objective is denoted by γ_1 and the secondary by γ_2, then such a problem can be referred to as $\alpha \mid \beta \mid \gamma_1^{(1)}, \gamma_2^{(2)}$.

Consider the following simple example. The primary objective is the total completion time $\sum C_j$ and the secondary objective is the maximum lateness $L_{\max}$, that is, $1 \mid\mid \sum C_j^{(1)}, L_{\max}^{(2)}$. If there are no jobs with identical processing times, then there is exactly one schedule that minimizes the total completion time; so there is no freedom remaining to minimize $L_{\max}$. If there are jobs with identical processing times, then there are multiple schedules that minimize the total completion time. A set of jobs with identical processing times is preceded by a job with a strictly shorter processing time and followed by a job with a strictly longer processing time. Jobs with identical processing times have to be

processed one after another; but, they may be done in any order. The decision-maker now must find among all the schedules that minimize the total completion time the one that minimizes $L_{\max}$. So, in an optimal schedule a set of jobs with identical processing times has to be sequenced according to the EDD rule. The decision-maker has to do so for each set of jobs with identical processing times. This rule may be referred to as SPT/EDD, since the jobs are first scheduled according to SPT and ties are broken according to EDD (see Exercise 4.16 for a generalization of this rule).

Consider now the same two objectives with reversed priorities, that is, $1 \mid\mid L_{\max}^{(1)}, \sum C_j^{(2)}$. In Chapter 3 it was shown that the EDD rule minimizes $L_{\max}$. Applying the EDD rule yields also the value of the minimum $L_{\max}$. Assume that the value of this minimum $L_{\max}$ is z. The original problem can be transformed into another problem that is equivalent. Create a new set of due dates $\bar{d}_j = d_j + z$. These new due dates are now deadlines. The problem is to find a schedule that minimizes $\sum C_j$ subject to the constraint that every job must be completed by its deadline, i.e., the maximum lateness with respect to the new due dates has to be zero or, equivalently, all the jobs have to be completed on time.

The algorithm for finding the optimal schedule is based on the following result.

Lemma 4.2.1. *For the single machine problem with n jobs subject to the constraint that all due dates have to be met, there exists a schedule that minimizes $\sum C_j$ in which job k is scheduled last, if and only if*

(i) $\bar{d}_k \geq \sum_{j=1}^{n} p_j$,

(ii) $p_k \geq p_\ell$, *for all* ℓ *such that* $\bar{d}_\ell \geq \sum_{j=1}^{n} p_j$.

Proof. By contradiction. Suppose that job k is not scheduled last. There is a set of jobs that is scheduled after job k and job ℓ is the one scheduled last. Condition (i) must hold for job ℓ otherwise job ℓ would not meet its due date. Assume that condition (ii) does not hold and that $p_\ell < p_k$. Perform a (nonadjacent) pairwise interchange between jobs k and ℓ. Clearly, the sum of the completion times of jobs k and ℓ decreases and the sum of the completion times of all jobs scheduled in between jobs k and ℓ goes down as well. So the original schedule that positioned job ℓ last could not have minimized $\sum C_j$. □

In the next algorithm J^c denotes the set of jobs that remain to be scheduled.

Algorithm 4.2.2 (Minimizing Total Completion Time with Deadlines)

Step 1.

Set $k = n$, $\tau = \sum_{j=1}^{n} p_j$, $J^c = \{1, \ldots, n\}$.

Step 2.

Find k^* *in* J^c *such that* $\bar{d}_{k^*} \geq \tau$ *and* $p_{k^*} \geq p_\ell$,
for all jobs ℓ *in* J^c *such that* $\bar{d}_\ell \geq \tau$.
Put job k^* *in position* k *of the sequence.*

Step 3.

Decrease k by 1.
Decrease τ by p_{k^*}.
Delete job k^* from J^c.

Step 4.

If $k \geq 1$ go to Step 2, otherwise STOP. ||

This algorithm, similar to the algorithms in Sections 3.2 and 3.6, is a backward algorithm. The following example illustrates the use of this algorithm.

Example 4.2.3 (Minimizing the Total Completion Time with Deadlines)

Consider the following instance with 5 jobs.

jobs	1	2	3	4	5
p_j	4	6	2	4	2
d_j	10	12	14	18	18

Starting out, $\tau = 18$. Two jobs have a deadline larger than or equal to τ, namely jobs 4 and 5. Job 4 has the longer processing time and should therefore go last. For the second iteration the value of τ is reduced to 14. There are two jobs that have a deadline greater than or equal to 14, namely 3 and 5. So either job can occupy the second last position. For the third iteration the value of τ is reduced further down to 12. Again, there are two jobs that have a deadline greater than or equal to 12; either jobs 2 and 3 or jobs 2 and 5. Clearly, job 2 (with a processing time of 6) should go in the third position. Proceeding in this manner yields two optimal schedules, namely schedules $5, 1, 2, 3, 4$ and $3, 1, 2, 5, 4$. ||

It can be shown that even when preemptions are allowed, the optimal schedules are nonpreemptive.

A fairly large number of problems of the type $1 \mid \beta \mid \gamma_1^{(1)}, \gamma_2^{(2)}$ have been studied in the literature. Very few can be solved in polynomial time. However, problems of the type $1 \mid \beta \mid \sum w_j C_j^{(1)}, \gamma_2^{(2)}$ tend to be easy (see Exercises 4.16 and 4.17).

4.3 Multiple Objectives: A Parametric Analysis

Suppose there are two objectives γ_1 and γ_2. If the overall objective is $\theta_1\gamma_1 + \theta_2\gamma_2$, where θ_1 and θ_2 are the weights of the two objectives, then a scheduling problem can be denoted by $1 \mid \beta \mid \theta_1\gamma_1 + \theta_2\gamma_2$. Since multiplying both weights by the

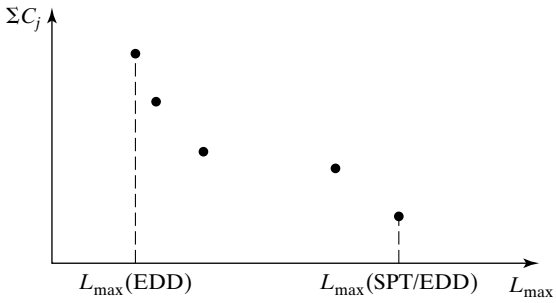


Fig. 4.3 Trade-offs between total completion time and maximum lateness

same constant does not change the problem, it is in what follows assumed that the weights add up to 1, i.e., $\theta_1 + \theta_2 = 1$. The remaining part of this section focuses on a specific class of schedules.

Definition 4.3.1 (Pareto-Optimal Schedule). *A schedule is called Pareto-optimal if it is not possible to decrease the value of one objective without increasing the value of the other.*

All Pareto-optimal solutions can be represented by a set of points in the (γ_1, γ_2) plane. This set of points illustrates the trade-offs between the two objectives. Consider the two objectives analyzed in the previous section, i.e., $\sum C_j$ and $L_{\max}$. The two cases considered in the previous section are the two extreme points of the trade-off curve. If $\theta_1 \to 0$ and $\theta_2 \to 1$, then

$$1 \mid \beta \mid \theta_1\gamma_1 + \theta_2\gamma_2 \quad \to \quad 1 \mid \beta \mid \gamma_2^{(1)}, \gamma_1^{(2)}.$$

If $\theta_2 \to 0$ and $\theta_1 \to 1$, then

$$1 \mid \beta \mid \theta_1\gamma_1 + \theta_2\gamma_2 \quad \to \quad 1 \mid \beta \mid \gamma_1^{(1)}, \gamma_2^{(2)}.$$

So the two extreme points of the trade-off curve in Figure 4.3 correspond to the problems discussed in the previous section. At one of the extreme points the total completion time is minimized by the SPT rule and ties are broken according to EDD; the $L_{\max}$ of this schedule can be computed easily and is denoted by $L_{\max}(SPT/EDD)$. At the other extreme point the schedule is generated according to a more complicated backward procedure. The $L_{\max}$ is equal to $L_{\max}(EDD)$ and the total completion time of this schedule can be computed also. Clearly,

$$L_{\max}(EDD) \leq L_{\max}(SPT/EDD).$$

The algorithm that generates all Pareto-optimal solutions in the trade-off curve contains two loops. One series of steps in the algorithm (the inner loop) is an adaptation of Algorithm 4.2.2. These steps determine, in addition to the

optimal schedule with a maximum allowable $L_{\max}$, also the minimum increment δ in the $L_{\max}$ that would allow for a decrease in the minimum $\sum C_j$. The second (outer) loop of the algorithm contains the structure that generates all the Pareto optimal points. The outer loop calls the inner loop at each Pareto-optimal point to generate a schedule at that point and also to determine how to move to the next efficient point. The algorithm starts out with the EDD schedule that generates the first Pareto-optimal point in the upper left part of the trade-off curve. It determines the minimum increment in the $L_{\max}$ needed to achieve a reduction in $\sum C_j$. Given this new value of $L_{\max}$, it uses the algorithm in the previous section to determine the schedule that minimizes $\sum C_j$, and proceeds to determine the next increment. This goes on until the algorithm reaches $L_{\max}(SPT/EDD)$.

Algorithm 4.3.2 (Determining Trade-Offs between Total Completion Time and Maximum Lateness)

Step 1.

Set $r = 1$.
Set $L_{\max} = L_{\max}(EDD)$ *and* $\bar{d}_j = d_j + L_{\max}$.

Step 2.

Set $k = n$ *and* $J^c = \{1, \ldots, n\}$.
Set $\tau = \sum_{j=1}^{n} p_j$ *and* $\delta = \tau$.

Step 3.

Find j^* *in* J^c *such that* $\bar{d}_{j^*} \geq \tau$ *and* $p_{j^*} \geq p_l$,
for all jobs l *in* J^c *such that* $\bar{d}_l \geq \tau$.
Put job j^* *in position* k *of the sequence.*

Step 4.

If there is no job ℓ *such that* $\bar{d}_\ell < \tau$ *and* $p_\ell > p_{j^*}$, *go to Step 5.*
Otherwise find j^{**} *such that*

$$\tau - \bar{d}_{j^{**}} = \min_{\ell}(\tau - \bar{d}_\ell)$$

for all ℓ *such that* $\bar{d}_\ell < \tau$ *and* $p_\ell > p_{j^*}$.
Set $\delta^{**} = \tau - d_{j^{**}}$.
If $\delta^{**} < \delta$, *then* $\delta = \delta^{**}$.

Step 5.

Decrease k *by 1.*
Decrease τ *by* p_{j^*}.
Delete job j^* *from* J^c.
If $k \geq 1$ *go to Step 3, otherwise go to Step 6.*

Step 6.

Set $L_{\max} = L_{\max} + \delta$.
If $L_{\max} > L_{\max}(SPT/EDD)$, *then STOP.*
Otherwise set $r = r + 1$, $\bar{d}_j = \bar{d}_j + \delta$, *and go to Step 2.* ||

The outer loop consists of Steps 1 and 6 while the inner loop consists of Steps 2, 3, 4, and 5. Steps 2, 3 and 5 represent an adaptation of Algorithm 4.2.2 and Step 4 computes the minimum increment in the $L_{\max}$ needed to achieve a subsequent reduction in $\sum C_j$.

It can be shown that the maximum number of Pareto-optimal solutions is $n(n-1)/2$, which is $O(n^2)$ (see Exercise 4.19). Generating one Pareto-optimal schedule can be done in $O(n\ log(n))$. The total computation time of Algorithm 4.3.2 is therefore $O(n^3\ log(n))$.

Example 4.3.3 (Determining Trade-Offs between Total Completion Time and Maximum Lateness)

Consider the following set of jobs.

jobs	1	2	3	4	5
p_j	1	3	6	7	9
d_j	30	27	20	15	12

The EDD sequence is 5, 4, 3, 2, 1 and $L_{\max}(EDD) = 2$. The SPT/EDD sequence is 1, 2, 3, 4, 5 and $L_{\max}(SPT/EDD) = 14$. Application of Algorithm 4.3.2 results in the following iterations.

Iteration	$(\sum C_j, L_{\max})$	Pareto-optimal schedule	current $\tau + \delta$	δ
1	96, 2	5, 4, 3, 1, 2	32 29 22 17 14	1
2	77, 3	1, 5, 4, 3, 2	33 30 23 18 15	2
3	75, 5	1, 4, 5, 3, 2	35 32 25 20 17	1
4	64, 6	1, 2, 5, 4, 3	36 33 26 21 18	2
5	62, 8	1, 2, 4, 5, 3	38 35 28 23 20	3
6	60, 11	1, 2, 3, 5, 4	41 38 31 26 23	3
7	58, 14	1, 2, 3, 4, 5	44 41 34 29 26	STOP

However, when one would consider the objective $\theta_1 L_{\max} + \theta_2 \sum C_j$, then certain Pareto-optimal schedules never may be optimal, no matter what the weights are (see Exercise 4.8). ||

Consider the generalization $1\ ||\ \theta_1 \sum w_j C_j + \theta_2 L_{\max}$. It is clear that the two extreme points of the trade-off curve can be determined in polynomial time (using WSPT/EDD and EDD). However, even though the two end-points of the trade-off curve can be analyzed in polynomial time, the problem with arbitrary weights θ_1 and θ_2 is NP-hard.

The trade-off curve that corresponds to the example in this section has the shape of a staircase. This shape is fairly common in a single machine environment with multiple objectives, especially when preemptions are not allowed.

However, in other machine environments, e.g., parallel machines, smoother curves may occur, especially when preemptions are allowed (see Chapter 15).

4.4 The Makespan with Sequence Dependent Setup Times

For single machine scheduling problems with all $r_j = 0$ and no sequence dependent setup times the makespan is independent of the sequence and equal to the sum of the processing times. When there are sequence dependent setup times the makespan *does* depend on the schedule. In Appendix D it is shown that $1 \mid s_{jk} \mid C_{\max}$ is strongly NP-hard.

However, the NP-hardness of $1 \mid s_{jk} \mid C_{\max}$ in the case of arbitrary setup times does not rule out the existence of efficient solution procedures when the setup times have a special form. And in practice setup times often do have a special structure.

Consider the following structure. Two parameters are associated with job j, say a_j and b_j, and $s_{jk} = \mid a_k - b_j \mid$. This setup time structure can be described as follows: after the completion of job j the machine is left in state b_j and to be able to start job k the machine has to be brought into state a_k. The total setup time necessary for bringing the machine from state b_j to state a_k is proportional to the absolute difference between the two states. This state variable could be, for example, temperature (in the case of an oven) or a measure of some other setting of the machine. In what follows it is assumed that at time zero the state is b_0 and that after completing the last job the machine has to be left in state a_0 (this implies that an additional "clean-up" time is needed after the last job is completed).

This particular setup time structure *does* allow for a polynomial time algorithm. The description of the algorithm is actually easier in the context of the Travelling Salesman Problem (TSP). The algorithm is therefore presented here in the context of a TSP with $n+1$ cities; the additional city being called city 0 with parameters a_0 and b_0. Without loss of generality it may be assumed that $b_0 \leq b_1 \leq \cdots \leq b_n$. The travelling salesman leaving city j for city k (or, equivalently, job k following job j) is denoted by $k = \phi(j)$. The sequence of cities in a tour is denoted by Φ, which is a vector that maps each element of $\{0, 1, 2, \ldots, n\}$ onto a unique element of $\{0, 1, 2, \ldots, n\}$ by relations $k = \phi(j)$ indicating that the salesman visits city k after city j (or, equivalently, job k follows job j). Such mappings are called permutation mappings. Note that not all possible permutation mappings of $\{0, 1, 2, \ldots, n\}$ constitute feasible TSP tours. For example, $\{0, 1, 2, 3\}$ mapped onto $\{2, 3, 1, 0\}$ represents a feasible TSP. However, $\{0, 1, 2, 3\}$ mapped onto $\{2, 1, 3, 0\}$ does not represent a feasible tour, since it represents two disjoint sub-tours, namely subtour $0 \rightarrow 2 \rightarrow 3 \rightarrow 0$ and the subtour $1 \rightarrow 1$ which consists of a single city (see Figure 4.4). Define $\phi(k) = k$ to mean a redundant tour that starts and ends at k.

For the special cost structure of going from city j to k it is clear that this cost is equal to the vertical height of the arrow connecting b_j with a_k in Figure 4.5.

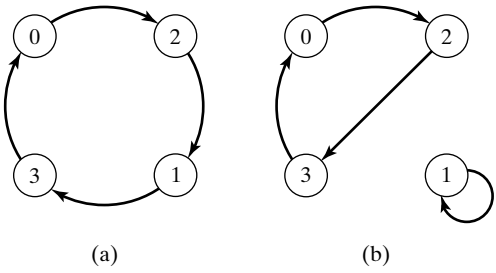


Fig. 4.4 Permutation mappings: (a) $\{0, 1, 2, 3\} \rightarrow \{2, 3, 1, 0\}$ (b) $\{0, 1, 2, 3\} \rightarrow \{2, 1, 3, 0\}$

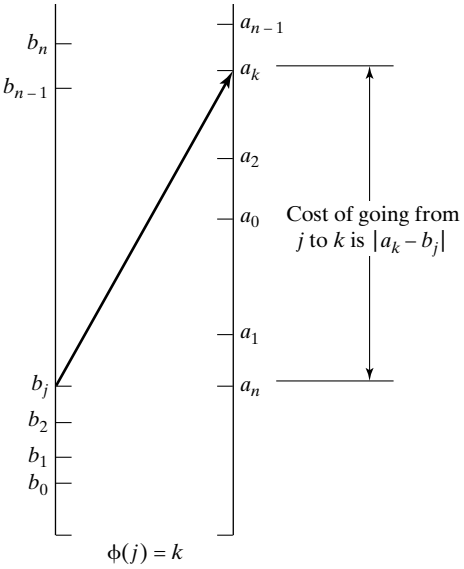


Fig. 4.5 Cost of going from j to k

Define the cost of a redundant sub-tour, i.e., $\phi(k) = k$, as the vertical height of an arrow from b_k to a_k.

Thus any permutation mapping (which might possibly consist of subtours) can be represented as a set of arrows connecting $b_j,\ j = 0, \ldots, n$ to $a_k,\ k = 0, \ldots, n$ and the cost associated with such a mapping is simply the sum of the vertical heights of the $n + 1$ arrows.

Define now a *swap* $I(j, k)$ as that procedure which when applied to a permutation mapping Φ produces another permutation mapping Φ' by affecting only the assignments of j and k and leaving the others unchanged. More precisely, the new assignment $\Phi' = \Phi I(j, k)$ is defined as:

$$\phi'(k) = \phi(j),$$
$$\phi'(j) = \phi(k),$$

and

$$\phi'(l) = \phi(l)$$

for all l not equal to j or k. This transformation may also be denoted by $\phi'(j) = \phi(j)I(j,k)$. Note that this is *not* equivalent to an adjacent pairwise interchange within a sequence, since a permutation mapping Φ does not always represent a sequence (a feasible TSP tour) to begin with. More intuitively, it only represents a swap of the arrows emanating from b_j and b_k leaving all other arrows unchanged. In particular, if these arrows crossed each other before they will uncross now and vice versa. The implication of such a swap in terms of the actual tour and subtours is quite surprising though. It can be easily verified that the swap $I(j,k)$ has the effect of creating two subtours out of one if j and k belong to the same subtour in Φ. Conversely, it combines two subtours to which j and k belong otherwise.

The following lemma quantifies the cost of the interchange $I(j,k)$ applied to the sequence Φ; the cost of this interchange is denoted by $c_\Phi I(j,k)$. In the lemma, the interval of the *unordered* pair $[a,b]$ refers to an interval on the real line and

$$|| \, [a,b] \, || = \begin{cases} 2(b-a) & \text{if } b \geq a \\ 2(a-b) & \text{if } b < a \end{cases}$$

Lemma 4.4.1. *If the swap $I(j,k)$ causes two arrows that did not cross earlier to cross, then the cost of the tour increases and vice versa. The magnitude of this increase or decrease is given by*

$$c_\Phi I(j,k) = || \, [b_j, b_k] \cap [a_{\phi(j)}, a_{\phi(k)}] \, ||$$

So the change in cost is equal to the length of vertical overlap of the intervals $[b_j, b_k]$ and $[a_{\phi(j)}, a_{\phi(k)}]$.

Proof. The proof can be divided into several cases and is fairly straightforward since the swap does not affect arrows other than the two considered. Hence it is left as an exercise (see Figure 4.6). □

The lemma is significant since it gives a visual cue to reducing costs by uncrossing the arrows that cross and helps quantify the cost savings in terms of amount of overlap of certain intervals. Such a visual interpretation immediately leads to the following result for optimal permutation mappings.

Lemma 4.4.2. *An optimal permutation mapping Φ^* is obtained if $b_j \leq b_k \Longrightarrow a_{\phi^*(j)} \leq a_{\phi^*(k)}$.*

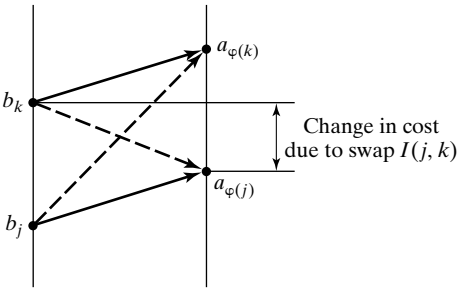


Fig. 4.6 Change in cost due to swap $I(j, k)$

Proof. The statement of the theorem is equivalent to no lines crossing in the diagram. Suppose two of the lines did cross. Performing a swap which uncrosses the lines leads to a solution as good or better than the previous. □

As mentioned before, this is simply an optimal *permutation mapping* and not necessarily a feasible tour. It does, however, provide a lower bound for the optimal cost of any TSP. This optimal Φ^* may consist of p distinct subtours, say, $TR_1, \ldots, TR_p$. As seen before, performing a swap $I(j, k)$ such that j and k belong to distinct subtours will cause these subtours to coalesce into one and the cost will increase (since now two previously uncrossed lines do cross) by an amount $c_{\Phi^*} I(j, k)$. It is desirable to select j and k from different subtours in such a way that this cost of coalescing $c_{\Phi^*} I(j, k)$ is, in some way, minimized.

To determine these swaps, instead of considering the *directed* graph which represents the subtours of the travelling salesman, consider the undirected version of the same graph. The subtours represent distinct cycles and redundant subtours are simply independent nodes. To connect the disjoint elements (i.e., the cycles corresponding to the subtours) and construct a connected graph, additional arcs have to be inserted in this undirected graph. The costs of the arcs between cities belonging to different subtours in this undirected graph are chosen to be equal to the cost of performing the corresponding swaps in the tour of the travelling salesman in the directed graph. The cost of such a swap can be computed easily by Lemma 4.4.1. The arcs used to connect the disjoint subtours are selected according to the *Greedy Algorithm*: select the cheapest arc which connects two of the p subtours in the undirected graph; select among the remaining unused arcs the cheapest arc connecting two of the $p - 1$ remaining subtours, and so on. The arcs selected then satisfy the following property.

Lemma 4.4.3. *The collection of arcs that connect the undirected graph with the least cost contain only arcs that connect city j to city $j + 1$.*

Proof. The cost of the arcs $(c_{\Phi^*} I(j, k))$ needed to connect the distinct cycles of the undirected graph are computed from the optimal permutation mapping defined in Lemma 4.4.2 in which no two arrows cross. It is shown below that

the cost of swapping two non-adjacent arrows is at least equal to the cost of swapping all arrows between them. This is easy to see if the cost is regarded as the intersection of two intervals given by Lemma 4.4.1. In particular, if $k > j+1$,

$$\begin{aligned} c_{\Phi^*} I(j,k) &= || \, [b_j, b_k] \cap [a_{\phi^*(j)}, a_{\phi^*(k)}] \, || \\ &\geq \sum_{i=j}^{k-1} || \, [b_i, b_{i+1}] \cap [a_{\phi^*(i)}, a_{\phi^*(i+1)}] \, || \\ &= \sum_{i=j}^{k-1} c_{\Phi^*} I(i, i+1) \end{aligned}$$

So the arc (j,k) can be replaced by the sequence of arcs $(i, i+1)$, $i = j, \ldots, k-1$ to connect the two subtours to which j and k belong at as low or lower cost. □

It is important to note that in the construction of the undirected graph, the costs assigned to the arcs connecting the subtours were computed under the assumption that the swaps are performed on Φ^* in which no arrows cross. However, as swaps are performed to connect the subtours this condition no longer remains valid. However, it can be shown that if the order in which the swaps are performed is determined with care, the costs of swaps are not affected by previous swaps. The following example shows that the sequence in which the swaps are performed can have an impact on the final cost.

Example 4.4.4 (Sequencing of Swaps)

Consider the situation depicted in Figure 4.7. The swap costs are $c_\Phi I(1,2) = 1$ and $c_\Phi I(2,3) = 1$. If the swap $I(2,3)$ is performed followed by the swap $I(1,2)$ the overlapping intervals which determine the costs of the interchange remain unchanged. However, if the sequence of swaps is reversed, i.e., first swap $I(1,2)$ is performed followed by swap $I(2,3)$, then the costs do change: the cost of the first swap remains, of course, the same but the cost of the second swap, $c_\Phi I(2,3)$ now has become 2 instead of 1.

The key point here is that the two swaps under consideration have an arrow in common, i.e., $b_2 \rightarrow a_{\phi(2)}$. This arrow points "up" and any swap that keeps it pointing "up" will not affect the cost of the swap below it as the overlap of intervals does not change. ||

The example suggests that if a sequence of swaps needs to be performed, the swaps whose lower arcs point "up" can be performed starting from the top going down without changing the costs of swaps below them. A somewhat similar statement can be made with respect to swaps whose lower arrows point "down"

In order to make this notion of "up" and "down" more rigorous, classify the nodes into two types. A node is said to be of Type I if $a_j \leq b_{\phi(j)}$, i.e., the arrow points "up", and it is of Type II if $a_j > b_{\phi(j)}$. A swap is of Type I if its

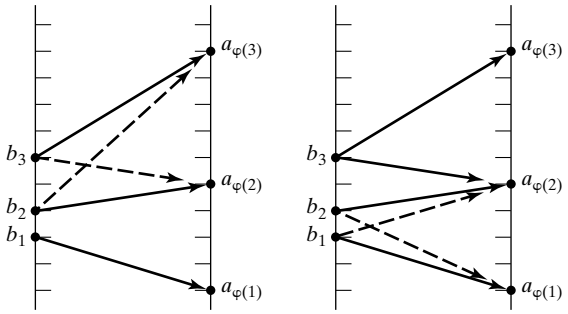


Fig. 4.7 Situation in Example 4.4.4

lower node is of Type I and of Type II if its lower node is of Type II. From the previous arguments it is easy to deduce that if the swaps $I(j, j+1)$ of Type I are performed in decreasing order of the node indices, followed by swaps of Type II in increasing order of the node indices, a single tour is obtained without changing any $c_{\Phi^*} I(j, j+1)$ involved in the swaps. The following algorithm sums up the entire procedure in detail.

Algorithm 4.4.5 (Finding Optimal Tour for TSP)

Step 1.

Arrange the b_j in order of size and renumber the jobs so that

$$b_0 \leq b_1 \leq \cdots \leq b_n.$$

The permutation mapping Φ^* is defined by

$$\phi^*(j) = k,$$

k being such that a_k is the $(j+1)$th smallest of the a_j.

Step 2.

Form an undirected graph with $n+1$ nodes and undirected arcs $A_{j,\phi^*(j)}$ connecting the jth and $\phi^*(j)$th nodes.
If the current graph has only one component, then STOP.
Otherwise go to Step 3.

Step 3.

Compute the interchange costs $c_{\Phi^*} I(j, j+1)$ for $j = 0, \ldots, n-1$:

$$c_{\Phi^*} I(j, j+1) = 2 \max \Big(\min(b_{j+1}, a_{\phi^*(j+1)}) - \max(b_j, a_{\phi^*(j)}), 0 \Big)$$

Step 4.

Select the smallest $c_{\Phi^*} I(j, j+1)$ such that j is in one component and

$j+1$ in another (break ties arbitrarily).
Insert the undirected arc $A_{j,j+1}$ into the graph and
repeat this step until all components in the undirected graph are connected.

Step 5.

Divide the arcs selected in Step 4 into two groups.
Those $A_{j,j+1}$ for which $a_{\phi^*(j)} \geq b_j$ go in Group 1;
the remaining go in Group 2.

Step 6.

Find the largest index j_1 such that A_{j_1,j_1+1} is in Group 1.
Find the second largest j_2, and so on.
Find the smallest index k_1 such that A_{k_1,k_1+1} is in Group 2.
Find the second smallest k_2, and so on.

Step 7.

The optimal tour Φ^{**} is constructed by applying the following
sequence of interchanges to the permutation Φ^*:

$$\Phi^{**} = \Phi^* I(j_1, j_1{+}1) I(j_2, j_2{+}1) \ldots I(j_l, j_l{+}1) I(k_1, k_1{+}1) I(k_2, k_2{+}1) \ldots I(k_m, k_m{+}1).$$

||

The total cost of the resulting tour may be viewed as consisting of two components. One is the cost of the unrestricted permutation mapping Φ^* before the interchanges are performed. The other is the additional cost caused by the interchanges.

That this algorithm actually leads to the optimal tour can be shown in two steps. First, a lower bound is established for the total cost of an arbitrary permutation mapping. Second, it is shown that this lower bound, in case the permutation mapping represents an actual tour, is greater than or equal to the total cost of the tour constructed in the algorithm. These two steps then prove the optimality of the tour of the algorithm. As this proof is somewhat intricate the reader is referred to the literature for its details.

A careful analysis of the algorithm establishes that the overall running time is bounded by $O(n^2)$.

Example 4.4.6 (Finding Optimal Tour for TSP)

Consider 7 cities with the parameters given below.

cities	0	1	2	3	4	5	6
b_j		1	15	26	40	3	19	31
a_j		7	16	22	18	4	45	34

Step 1. Reordering the cities in such a way that $b_j \leq b_{j+1}$ results in the ordering below and the $\phi^*(j)$ below:

cities	0	1	2	3	4	5	6
b_j		1	3	15	19	26	31	40
$a_{\phi^*(j)}$	4	7	16	18	22	34	45
$\phi^*(j)$	1	0	2	6	4	5	3

Step 2. Form the undirected graph with j connected $\phi^*(j)$. Nodes 0 and 1 have to be connected with one another; nodes 3 and 6 have to be connected with one another; nodes 2, 4 and 5 are independent (each one of these three nodes is connected with itself).

Step 3. Computation of the interchange costs $c_{\Phi^*} I(j, j+1)$ gives

$$\begin{aligned} c_{\Phi^*} I(0,1) &= 0 \\ c_{\Phi^*} I(1,2) &= 2(15-7) = 16 \\ c_{\Phi^*} I(2,3) &= 2(18-16) = 4 \\ c_{\Phi^*} I(3,4) &= 2(22-19) = 6 \\ c_{\Phi^*} I(4,5) &= 2(31-26) = 10 \\ c_{\Phi^*} I(5,6) &= 2(40-34) = 12 \end{aligned}$$

Step 4. The undirected arcs $A_{1,2}$, $A_{2,3}$, $A_{3,4}$ and $A_{4,5}$ are inserted into the graph.

Step 5. The four arcs have to be partitioned into the two groups. In order to determine this, each b_j has to be compared to the corresponding $a_{\phi^*(j)}$.

arcs	b_j	$a_{\phi^*(j)}$	Group
$A_{1,2}$	$b_1 = 3$	$a_{\phi^*(1)} = a_0 = 7$	1
$A_{2,3}$	$b_2 = 15$	$a_{\phi^*(2)} = a_2 = 16$	1
$A_{3,4}$	$b_3 = 19$	$a_{\phi^*(3)} = a_6 = 18$	2
$A_{4,5}$	$b_4 = 26$	$a_{\phi^*(4)} = a_4 = 22$	2

Step 6. $j_1 = 2$, $j_2 = 1$, $k_1 = 3$ and $k_2 = 4$.

Step 7. The optimal tour is obtained after the following interchanges.

$$\Phi^{**} = \Phi^* I(2,3) I(1,2) I(3,4) I(4,5).$$

So the optimal tour is

$$0 \to 1 \to 6 \to 3 \to 4 \to 5 \to 2 \to 0.$$

The cost of this tour is

$$3 + 15 + 5 + 3 + 8 + 15 + 8 = 57 \qquad ||$$

In practice, when the setup times have an arbitrary structure, the myopic *Shortest Setup Time (SST) first* rule is often used. This rule implies that whenever a job is completed, the job with the smallest setup time is selected to go next. This SST rule is equivalent to the *Nearest Neighbour* rule for the TSP. Applying the SST rule to the instance in Example 4.4.6 results in the tour

$$0 \rightarrow 1 \rightarrow 2 \rightarrow 6 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 0.$$

The associated cost is

$$3 + 13 + 3 + 5 + 3 + 8 + 24 = 59.$$

This tour is not optimal.

Even though the SST rule usually leads to reasonable schedules, there are instances where the ratio

$$\frac{C_{\max}(SST) - \sum_{j=1}^{n} p_j}{C_{\max}(OPT) - \sum_{j=1}^{n} p_j}$$

is quite large. Nevertheless, the SST rule is often used as an integral component within more elaborate dispatching rules (see Section 14.2).

4.5 Job Families with Setup Times

Consider n jobs that belong to F different job families. Jobs from the same family may have different processing times, but they can be processed one after another without requiring any setup in between. However, if the machine switches over from one family to another, say from family g to family h, then a setup is required. If the setup time is sequence dependent, it is denoted by s_{gh}. If the setup time depends only on the family that is about to start, it is denoted by s_h. If it does not depend on either family, it is denoted by s. In what follows, sequence dependent setup times satisfy the so-called triangle inequality, i.e., $s_{fg} + s_{gh} \geq s_{fh}$ for any three families f, g, and h. (The reverse inequality would not make sense, since one always would do then two setups instead of the single longer setup.) If the very first job to be processed in the sequence belongs to family h, then the setup at time 0 is s_{0h}.

This section does not consider the makespan objective, since it has already been discussed in a fair amount of detail in the previous section. This section does cover the total weighted completion time objective, the maximum lateness objective, and the number of tardy jobs objective.

Consider the problem $1 \mid fmls, s_{gh} \mid \sum w_j C_j$. Before describing a backward dynamic programming procedure for this problem it is necessary to establish some properties of optimal schedules. Any schedule consists of F subsequences of jobs that are intertwined with one another and each subsequence corresponds to one family. The next result focuses on the subsequences in optimal schedules.

Lemma 4.5.1. *In an optimal schedule for* $1 \mid fmls, s_{gh} \mid \sum w_j C_j$ *jobs from the same family are ordered according to WSPT.*

Proof. Consider an optimal sequence $\sigma^* = \sigma_1, j, \sigma_2, k, \sigma_3$, where jobs j and k are from the same family and σ_1, σ_2 and σ_3 are arbitrary partial sequences. The partial sequence σ_2 does not contain any job from the family that jobs j and k belong to. It suffices to show that if $w_j/p_j < w_k/p_k$, then either sequence $\sigma' = \sigma_1, k, j, \sigma_2, \sigma_3$ or sequence $\sigma'' = \sigma_1, \sigma_2, k, j, \sigma_3$ has a smaller total weighted completion time. It can be shown that, because of the triangle inequality that applies to setup times, the interchanges yielding sequences σ' and σ'' reduce the total setup time (subsequence σ_3 starts in sequence σ' as well as in sequence σ'' earlier than in sequence σ^*). The setup times can therefore be ignored.

It is possible to replace two consecutive jobs u and v in a sequence by a single composite job r with processing time $p_r = p_u + p_v$ and weight $w_r = w_u + w_v$. This increases the total weighted completion time of the sequence by an amount $w_u p_v$. Replacing the partial sequence σ_2 in σ^*, σ' and σ'' by an accordingly defined composite job ℓ changes all three objective values by the same constant (independently from the position of σ_2 in the sequence). So, substituting a partial sequence with a composite job changes the overall cost only by a constant; comparisons of the schedules based on the values of the objective function will yield the same result.

Now it is easy to see (through a standard adjacent pairwise interchange argument), that since job k has a higher priority than job j, either the composite job ℓ has a higher priority than job j implying that σ'' is better than σ^* or the composite job ℓ has a lower priority than job k implying that σ' is better than σ^*. □

In order to describe a backward dynamic programming procedure, some notation has to be introduced. Let n_g denote the number of jobs from family g. Let (j, g) refer to job j from family g, $j = 1, \ldots, n_g$; it has a processing time p_{jg} and a weight w_{jg}. Without loss of generality one can assume that

$$\frac{w_{1g}}{p_{1g}} \geq \frac{w_{2g}}{p_{2g}} \geq \cdots \geq \frac{w_{n_g,g}}{p_{n_g,g}}$$

for all $g = 1, \ldots, F$.

Let $V(q_1, \ldots, q_F, h)$ denote the minimum total weighted completion time of schedules that contain jobs $(q_g, g), \ldots, (n_g, g)$ for $g = 1, \ldots, F$ where job (q_h, h) from family h is the first one to be processed starting at time 0. In other words, $V(q_1, \ldots, q_F, h)$ defines the minimum total weighted completion time among all schedules that contain for each family g, $g = 1, \ldots, F$, all $n_g - q_g + 1$ lowest

priority jobs $(q_g, g), \ldots, (n_g, g)$ and that starts at time 0 with a batch of jobs from family h. Note that $q_h \leq n_h$, and that the setup for the batch of jobs of family h at the start of the schedule is not included.

A backward dynamic programming algorithm can now be described as follows.

Algorithm 4.5.2 (Minimizing the Total Weighted Completion Time)
Initial Condition:

$$V(n_1 + 1, \ldots, n_F + 1, g) = 0, \qquad \text{for } g = 1, \ldots, F.$$

Recursive Relation:

$$V(q_1, \ldots, q_F, h) = \min_{h'=1,\ldots,F} \left(V(q'_1, \ldots, q'_F, h') + (p_{q_h,h} + s_{hh'}) \sum_{g=1}^{F} \sum_{j=q_g}^{n_g} w_{jg} - s_{hh'} w_{q_h,h} \right),$$

$$\text{where } q'_h = q_h + 1 \text{ and } q'_g = q_g \text{ if } g \neq h, \text{ and } s_{hh'} = 0 \text{ if } h = h';$$
$$\text{for } q_g = n_g + 1, n_g, \ldots, 1, \;\; g = 1, \ldots, F, \;\; h = 1, \ldots, F.$$

Optimal Value Function:

$$\min_{h=1,\ldots,F} \left(V(1, \ldots, 1, h) + s_{0h} \sum_{g=1}^{F} \sum_{j=1}^{n_g} w_{jg} \right) \qquad ||$$

In words, this algorithm can be described as follows. The minimization selects a previous schedule to which job (q_h, h) is appended at the beginning. If the first job of the previous schedule is also from family h, i.e., $h' = h$, then this previous schedule is only delayed by $p_{q_h,h}$. On the other hand, if the first job of the previous schedule is from family h', where $h' \neq h$, then the delay is $p_{q_h,h} + s_{hh'}$, because the first job of the previous schedule starts a new batch requiring a setup between the job from family h and the job from family h'.

It is easy to obtain an upper bound for the makespan of any schedule by taking the sum of all the processing times plus n times the maximum setup time. Let $\mathcal{U}$ denote this upper bound. The number of states for which the value function has to be computed recursively is then $O(n^F F\, \mathcal{U})$. The value of each state can be computed in $O(F)$ (since the minimum is taken over F values). So the algorithm operates in $O(F^2 n^F\, \mathcal{U})$.

Example 4.5.3 (Dynamic Programming and the Total Weighted Completion Time)

Consider two families, i.e., $F = 2$. The sequence dependent setup times between the families are $s_{12} = s_{21} = 2$ and $s_{01} = s_{02} = 0$. There is one job in family 1 and two jobs in family 2, i.e., $n_1 = 1$ and $n_2 = 2$. The processing times are in the table below:

jobs	(1,1)	(1,2)	(2,2)
p_{jg}	3	1	1
w_{jg}	27	30	1

Applying the WSPT rule to the two jobs of family 2 indicates that job (1,2) should appear in the schedule before job (2,2).

Applying the dynamic procedure results in the following computations. The initial conditions are: $V(2,3,1) = V(2,3,2) = 0$. These initial conditions basically represent empty schedules.

The first recursive relation computes an optimal value function by appending job (1,1) to the empty schedule and then computes an optimal value function by appending job (2,2) to the empty schedule.

$$\begin{aligned} V(1,3,1) &= \min\Big(V(2,3,1) + (p_{11}+s_{11})w_{11} - s_{11}w_{11}, \\ &\qquad V(2,3,2) + (p_{11}+s_{12})w_{11} - s_{12}w_{11}\Big) \\ &= \min\ \Big(0 + (3+0)27 - 0\times 27,\ \ 0 + (3+2)27 - 2\times 27\Big)\ =\ 81 \end{aligned}$$

and

$$\begin{aligned} V(2,2,2) &= \min\Big(V(2,3,2) + (p_{22}+s_{22})w_{22} - s_{22}w_{22}, \\ &\qquad V(2,3,1) + (p_{22}+s_{21})w_{22} - s_{21}w_{22}\Big) \\ &= \min\ \Big(0 + (1+0)1 - 0\times 1,\ \ 0 + (1+2)1 - 2\times 1\Big)\ =\ 1 \end{aligned}$$

The next value functions to be computed are $V(1,2,1)$ and $V(1,2,2)$.

$$\begin{aligned} V(1,2,1) &= V(2,2,2) + (p_{11}+s_{12})(w_{11}+w_{22}) - s_{12}w_{11} \\ &= 1 + (3+2)(27+1) - 2\times 27\ =\ 87 \end{aligned}$$

(Note that it was not necessary here to consider $V(2,2,1)$ on the RHS of the expression above, since state $(2,2,1)$ is not a feasible state.) Similarly,

$$\begin{aligned} V(1,2,2) &= V(1,3,1) + (p_{22}+s_{21})(w_{11}+w_{22}) - s_{21}w_{22} \\ &= 81 + (1+2)(27+1) - 2\times 1\ =\ 163 \end{aligned}$$

(Again, it is not necessary to consider here $V(1,3,2)$ since state $(1,3,2)$ is not feasible.)

Proceeding in a similar fashion yields

$$V(2,1,2) = V(2,2,2) + (p_{12})(w_{12} + w_{22}) = 1 + 1 \times (30 + 1) = 32.$$

Finally,

$$\begin{aligned} V(1,1,1) &= V(2,1,2) + (p_{11} + s_{12})(w_{11} + w_{12} + w_{22}) - s_{12}w_{11} \\ &= 32 + (3+2)(27 + 30 + 1) - 2 \times 27 \;=\; 268 \end{aligned}$$

and

$$\begin{aligned} V(1,1,2) = \min \Big(& V(1,2,1) + (p_{12} + s_{21})(w_{12} + w_{22} + w_{11}) - s_{21}w_{12}, \\ & V(1,2,2) + (p_{12})(w_{12} + w_{22} + w_{11}) \Big) \\ = \min \Big(& 87 + 3 \times (27 + 30 + 1) - 2 \times 30, \;\; 163 + 1 \times (27 + 30 + 1) \Big) \\ = \min(201, 221) \;&=\; 201 \end{aligned}$$

The optimal value function is

$$\min \Big(V(1,1,1), V(1,1,2) \Big) = \min(268, 201) = 201.$$

Backtracking yields the optimal schedule $(1,2),(1,1),(2,2)$ with a total weighted completion time of 201. ||

The next objective to be considered is $L_{\max}$, i.e., the problem $1 \mid fmls, s_{gh} \mid L_{\max}$. Let d_{jg} denote the due date of job (j,g), $j = 1, \ldots, n_g$, $g = 1, \ldots, F$. Before describing the dynamic programming procedure for this problem it is again necessary to establish some properties pertaining to optimal schedules. Again, a schedule can be regarded as a combination of F subsequences that are intertwined with one another, each subsequence corresponding to one family. The following lemma focuses on these subsequences.

Lemma 4.5.4. *There exists an optimal schedule for* $1 \mid fmls, s_{gh} \mid L_{\max}$ *with the jobs from any given family sequenced according to EDD.*

Proof. The proof can be constructed in a manner that is similar to the proof of Lemma 4.5.1 and is left as an exercise. □

In order to formulate a dynamic programming procedure, first assume that

$$d_{1g} \leq d_{2g} \leq \cdots \leq d_{n_g,g},$$

for $g = 1, \ldots, F$. Let $V(q_1, \ldots, q_F, h)$ denote the minimum value of the maximum lateness for schedules containing jobs $(q_g, g), \ldots, (n_g, g)$ for $g = 1, \ldots, F$ where job (q_h, h) is processed first starting at time zero, and the setup for the

batch of jobs of family h at the start of the schedule is not included. The following backward dynamic programming procedure can now be applied to this problem.

Algorithm 4.5.5 (Minimizing the Maximum Lateness)
Initial Condition:

$$V(n_1+1,\ldots,n_F+1,g) = -\infty, \qquad \textit{for } g = 1,\ldots,F.$$

Recursive Relation:

$$V(q_1,\ldots,q_F,h) = \min_{h'=1,\ldots,F} \Big(\max \Big(V(q'_1,\ldots,q'_F,h') + p_{q_h,h} + s_{hh'},\ \ p_{q_h,h} - d_{q_h,h} \Big) \Big)$$

where $q'_h = q_h + 1$, $q'_g = q_g$ *if* $g \neq h$, *and* $s_{hh'} = 0$ *if* $h = h'$;
for $q_g = n_g + 1, n_g, \ldots, 1, \quad g = 1,\ldots,F, \quad h = 1,\ldots,F.$

Optimal Value Function:

$$\min_{h=1,\ldots,F} \Big(V(1,\ldots,1,h) + s_{0h} \Big) \qquad ||$$

In words, the minimization in the recursive relationship assumes that if job (q_h, h) (with processing time $p_{q_h,h}$) is appended at the beginning of a schedule that contains jobs $(q'_1, 1), \ldots, (q'_F, F)$, then the maximum lateness of these jobs is increased by $p_{q_h,h} + s_{hh'}$, while the lateness of job (q_h, h) itself is $p_{q_h,h} - d_{q_h,h}$. In the recursive relationship the maximum of these latenesses has to be minimized. The time complexity of this algorithm can be determined in the same way as the time complexity of the algorithm for the total weighted completion time; it operates also in $O(F^2 n^F \mathcal{U})$.

Consider now the problem $1 \mid \textit{fmls}, s_{gh} \mid \sum U_j$. Recall that the algorithm that yields an optimal solution for $1 \mid\mid \sum U_j$ operates forward in time. This already may suggest that it probably would not be that easy to find a backward dynamic programming algorithm for $1 \mid \textit{fmls}, s_{gh} \mid \sum U_j$; this problem requires, indeed, a forward dynamic programming algorithm.

Before describing the dynamic programming algorithm that solves this problem it is again necessary to establish some properties of optimal schedules. An optimal schedule can again be regarded as a combination of F subsequences that are intertwined, with each subsequence corresponding to one family. A subsequence from a family contains jobs that are completed early as well as jobs that are completed late. The early jobs appear in the subsequence before the late jobs. The following lemma focuses on the structure of such a subsequence.

Lemma 4.5.6. *There exists an optimal schedule for* $1 \mid \textit{fmls}, s_{gh} \mid \sum U_j$ *that has all the on-time jobs from any given family sequenced according to EDD. In such an optimal schedule the jobs from any given family that are finished late are processed after all on-time jobs from that family have been completed.*

Proof. The proof is easy and is left as an exercise. □

Assume again that the jobs within each family g are indexed so that $d_{1g} \leq \cdots \leq d_{n_g,g}$. In order to formulate a dynamic program for $1 \mid fmls, s_{gh} \mid \sum U_j$ let $V(q_1, \ldots, q_F, u, h)$ denote the minimum makespan for schedules that contain early jobs from the sets $\{(1,g), \ldots, (q_g, g)\}$, $g = 1, \ldots, F$, with u being the total number of late jobs from these sets and a family h job being the last job in the schedule that is completed early. Note that $q_h \geq 1$. The following forward dynamic programming algorithm solves the problem.

Algorithm 4.5.7 (Minimizing the Number of Tardy Jobs)
Initial Condition:

$$V(q_1, \ldots, q_F, u, 0) = \begin{cases} 0, & \text{for } u = \sum_{g=1}^{F} q_g, \\ \infty, & \text{otherwise} \end{cases}$$

$$\text{for} \quad q_g = 0, 1, \ldots, n_g, \quad g = 1, \ldots, F, \quad u = 0, 1, \ldots, \sum_{g=1}^{F} q_g.$$

Recursive Relation:

$$V(q_1, \ldots, q_F, u, h) = \min \Big(\min_{h' \in G(q_1, \ldots, q_F, u, h)} \Big(V(q'_1, \ldots, q'_F, u, h') \; + \; \tau \Big), \; V(q'_1, \ldots, q'_F, u-1, h) \Big),$$

where $q'_g = q_g$ *for* $g \neq h$, $q'_h = q_h - 1$, $\tau = s_{h'h} + p_{q_h,h}$,
$s_{h'h} = 0$ *if* $h = h'$, *and where* $G(q_1, \ldots, q_F, u, h) =$
$= \{h' \mid h' \in \{0, 1, \ldots, g\}, \; V(q'_1, \ldots, q'_g, u, h') + \tau \leq d_{q_h,h}\}$;
for $q_g = 0, 1, \ldots, n_g$, $g = 1, \ldots, F$, *and*
$u = 0, 1, \ldots, \sum_{g=1}^{F} q_g$ *and* $h = 1, \ldots, F$.

Optimal Value Function:

$$\min \Big(u \mid \min_{g=0,1,\ldots,F} \Big(V(n_1, \ldots, n_F, u, g) \Big) < \infty \Big) \qquad ||$$

In words, the procedure can be described as follows: the first term in the minimization of the recursion selects job (q_h, h) to be scheduled on time if this is possible and chooses a batch h' for the previous on-time job; the second term selects job q_h of batch h to be late.

Note that the optimal value function is equal to the smallest value of u for which

$$\min_{g=0,1,\ldots,F} \Big(V(n_1, \ldots, n_F, u, g) \Big) < \infty.$$

In order to determine the computational complexity of the procedure, note that the number of states that have to be evaluated is again $O(n^F n \; F)$. Since

each recursive step requires $O(F)$ steps to solve, the time complexity of this algorithm is $O(F^2 n^{F+1})$, which is polynomial for fixed F.

The more general problem $1 \mid fmls, s_{gh} \mid \sum w_j U_j$ can be solved in pseudopolynomial time when F is fixed. This result follows from the fact that Algorithm 4.5.7 can be generalized to minimize the weighted number of late jobs in $O(n^F W)$ time, where $W = \sum_{j=1}^{n} w_j$.

The total tardiness objective $\sum T_j$ and the total weighted tardiness objective $\sum w_j T_j$ turn out to be considerably harder than the $\sum U_j$ objective.

4.6 Batch Processing

Consider a machine that can process a number of jobs simultaneously, i.e., a machine that can process a batch of jobs at the same time. The processing times of the jobs in a batch may not be all the same and the entire batch is finished only when the last job of the batch has been completed, i.e., the completion time of the entire batch is determined by the job with the longest processing time. This type of machine is fairly common in industry. Consider, for example, the "burn-in" operations in the manufacturing process of circuit boards; these operations are performed in ovens that can handle many jobs simultaneously.

Let b denote the maximum number of jobs that can be processed in a batch. Clearly, the case $b = 1$ refers to the standard scheduling environment considered in previous sections. It is to be expected that the $b = 1$ case is easier than the case $b \geq 2$. Another special case that tends to be somewhat easier is the case $b = \infty$ (i.e., there is no limit on the batch size). This case is not uncommon in practice; it occurs frequently in practice when the items to be produced are relatively small and the equipment is geared for a high volume. In this section the case $b = \infty$ (or equivalently, $b \geq n$) is considered first; several objective functions are discussed. Subsequently, the case $2 \leq b \leq n - 1$ is considered; several objective functions are discussed.

When $b = \infty$ the minimization of the makespan is trivial. All jobs are processed together and the makespan is the maximum of the n processing times. However, other objective functions are not that easy. Assume $p_1 \leq p_2 \leq \cdots \leq p_n$. An *SPT-batch* schedule is defined as a schedule in which adjacent jobs in the sequence $1, \ldots, n$ are assembled in batches. For example, a possible batch schedule for an 8-job problem is a sequence of four batches $(\{1, 2\}, \{3, 4, 5\}, \{6\}, \{7, 8\})$. The following result holds for $1 \mid batch(\infty) \mid \gamma$ when the objective function γ is a regular performance measure.

Lemma 4.6.1. *If the objective function γ is regular and the batch size is unlimited, then the optimal schedule is an SPT-batch schedule.*

Proof. The proof is easy and left as an exercise (see Exercise 4.22). □

Consider the model $1 \mid batch(\infty) \mid \sum w_j C_j$. This problem can be solved via dynamic programming. Let $V(j)$ denote the minimum total weighted com-

pletion time of an *SPT-batch* schedule that contains jobs $j, \ldots, n$, assuming that the first batch starts at $t = 0$. Let $V(n+1)$ denote the minimum total weighted completion time of the empty set, which is zero. A backward dynamic programming procedure can be described as follows.

Algorithm 4.6.2 (Minimizing Total Weighted Completion Time – Batch Size Infinite)

Initial Condition:

$$V(n+1) = 0$$

Recursive Relation:

$$V(j) = \min_{k=j+1,\ldots,n+1} \left(V(k) + p_{k-1} \sum_{h=j}^{n} w_h \right)$$

$$j = n, \ldots, 1.$$

Optimal Value Function:

$$V(1) \qquad ||$$

The minimization in the recursive relationship of the dynamic program selects the batch of jobs $\{j, \ldots, k-1\}$ with processing time p_{k-1} for insertion at the start of a previously obtained schedule that comprises jobs $\{k, \ldots, n\}$. It is clear that this algorithm is $O(n^2)$.

Consider now the model $1 \mid batch(\infty) \mid L_{\max}$. This problem also can be solved via a backward dynamic programming procedure. Assume again that $p_1 \leq p_2 \leq \cdots \leq p_n$. Let $V(j)$ denote now the minimum value of the maximum lateness for *SPT-batch* schedules containing jobs $j, \ldots, n$, assuming their processing starts at time $t = 0$.

Algorithm 4.6.3 (Minimizing Maximum Lateness – Batch Size Infinite)

Initial Condition:

$$V(n+1) = -\infty$$

Recursive Relation:

$$V(j) = \min_{k=j+1,\ldots,n+1} \left(\max \left(V(k) + p_{k-1}, \max_{h=j,\ldots,k-1} (p_{k-1} - d_h) \right) \right)$$

$$j = n, \ldots, 1.$$

Optimal Value Function:

$$V(1) \qquad ||$$

The minimization in the recursive relationship assumes that if a batch of jobs $j, \ldots, k-1$ (with processing time p_{k-1}) is inserted at the beginning of a schedule for jobs $k, \ldots, n$, then the maximum lateness of jobs $k, \ldots, n$ increases by p_{k-1}, while the maximum lateness among jobs $j, \ldots, k-1$ is

$$\max_{h=j,\ldots,k-1}(p_{k-1} - d_h).$$

This algorithm also operates in $O(n^2)$.

Example 4.6.4 (Minimizing Maximum Lateness – Batch Size Infinite)

Consider the following scheduling with five jobs, i.e., $n = 5$.

jobs	1	2	3	4	5
p_j	2	3	8	10	27
d_j	10	7	6	16	43

The initial condition is $V(6) = -\infty$. The recursive relationships result in the following:

$$V(5) = \max(V(6) + p_5, p_5 - d_5) = -16.$$

$$\begin{aligned} V(4) &= \min_{k=5,6}\Big(\max(V(k) + p_{k-1},\ \max_{h=4,\ldots,k-1}(p_{k-1} - d_h))\Big) \\ &= \min\Big(\max(-16 + 10, 10 - 16)\ ,\ \max(-\infty, 11, -16)\Big) \\ &= \min(-6, 11) = -6 \end{aligned}$$

$$\begin{aligned} V(3) &= \min_{k=4,5,6}\Big(\max(V(k) + p_{k-1}, \max_{h=3,\ldots,k-1}(p_{k-1} - d_h))\Big) \\ &= \min\Big(\max(-6 + 8, 8 - 6)\ ,\ \max(-16 + 10, 10 - 6, 10 - 16)\ , \\ &\quad \max(-\infty, 27 - 6, 27 - 16, 27 - 43)\Big) \\ &= \min(2, 4, 21) = 2 \end{aligned}$$

$$V(2) = \min_{k=3,\ldots,6}\Big(\max(V(k) + p_{k-1}, \max_{h=2,\ldots,k-1}(p_{k-1} - d_h))\Big)$$

$$= \min\Big(\max(2+3, 3-7)\ ,\ \max(-6+8, 8-7, 8-6)\ ,$$
$$\max(-16+10, 10-7, 10-6, 10-16),$$
$$\max(-\infty, 27-7, 27-6, 27-16, 27-43)\Big)$$
$$= \min(5, 2, 4, 21) \ = \ 2$$

$$V(1) = \min_{k=2,\ldots,6} \Big(\max(V(k) + p_{k-1},\ \max_{h=1,\ldots,k-1}(p_{k-1} - d_h))\Big)$$
$$= \min\Big(\max(4, -8)\ ,\ \max(5, -7, -4)\ ,\ \max(2, -2, 1, 2),$$
$$\max(-6, 0, 3, 4, -6)\ ,\ \max(-\infty, 17, 20, 21, 11, -16)\Big)$$
$$= \min(4, 5, 2, 4, 21) \ = \ 2$$

Backtracking yields the following schedule: The fact that the minimum for $V(1)$ is reached for $k = 4$ implies that the first three jobs are put together in one batch. The minimum for $V(4)$ is reached for $k = 5$, implying that job 4 is put in a batch by itself. ||

The $1 \mid batch(\infty) \mid \sum U_j$ problem is slightly more complicated. In Chapter 3 it was already observed that there is not any backward algorithm for minimizing the number of late jobs when $b = 1$. It turns out that no backward algorithm has been found for the $b = \infty$ case either. However, the problem can be solved via a forward dynamic programming algorithm. Let $V(j, u, k)$ denote the minimum makespan of an *SPT-batch* schedule for jobs $1, \ldots, j$, with u being the number of late jobs among these j jobs and the last batch having a processing time p_k (implying that this last batch will end up containing also jobs $j+1, \ldots, k$, but not job $k+1$).

The dynamic program operates in a forward manner and distinguishes between two cases. First, it considers adding job j to the schedule while assuming that it does not initiate a new batch, i.e., job j is included in the same batch as job $j-1$ and that batch has a processing time p_k. This processing time p_k already contributes to the makespan of the previous state, which may be either $V(j-1, u, k)$ or $V(j-1, u-1, k)$ dependent upon whether job j is on time or not. If $V(j-1, u, k) \leq d_j$, then job j is on time and $(j-1, u, k)$ is the previous state; if $V(j-1, u-1, k) > d_j$, then job j is late and $(j-1, u-1, k)$ is the previous state.

Second, it considers adding job j to the schedule assuming that it initiates a new batch. The previous batch ends with job $j-1$ and the processing time of the new batch is p_k. After adding the contribution from the previous state, the makespan becomes either $V(j-1, u, j-1) + p_k$ or $V(j-1, u-1, j-1) + p_k$ dependent upon whether job j is on time or not. If $V(j-1, u, j-1) + p_k \leq d_j$, then job j is assumed to be on time and $(j-1, u, j-1)$ is the previous state; if $V(j-1, u-1, j-1) + p_k > d_j$, then job j is assumed to be tardy and $(j-1, u-1, j-1)$ is the previous state.

Algorithm 4.6.5 (Minimizing Number of Tardy Jobs – Batch Size Infinite)

Initial Condition:

$$V(0,0,k) = \begin{cases} 0, & \text{if } k = 0, \\ \infty, & \text{otherwise} \end{cases}$$

Recursive Relation:

$$V(j,u,k) = \min \begin{cases} V(j-1,u,k), & \text{if } V(j-1,u,k) \leq d_j, \\ V(j-1,u-1,k), & \text{if } V(j-1,u-1,k) > d_j, \\ V(j-1,u,j-1) + p_k, & \text{if } V(j-1,u,j-1) + p_k \leq d_j, \\ V(j-1,u-1,j-1) + p_k & \text{if } V(j-1,u-1,j-1) + p_k > d_j, \\ \infty, & \text{otherwise} \end{cases}$$

$$\text{for } j = 1,\ldots,n, \quad u = 0,\ldots,j, \quad k = j,\ldots,n.$$

Optimal Value Function:

$$\min \{\ u \mid V(n,u,n) \ < \infty\} \qquad ||$$

Note that the optimal value function is the smallest value of u for which $V(n,u,n) < \infty$. This algorithm also operates in $O(n^3)$.

Other batch scheduling problems with due date related objective functions and unlimited batch sizes tend to be harder. For example, $1 \mid batch(\infty) \mid \sum T_j$ is NP-Hard in the ordinary sense and can be solved in pseudopolynomial time.

Consider now the class of batch scheduling problems with finite and fixed batch sizes, i.e., the batch size is b and $2 \leq b \leq n-1$. It is to be expected that scheduling problems with finite and fixed batch sizes are harder than their counterparts with unlimited batch sizes. For starters, the result regarding the optimality of *SPT-batch* schedules when performance measures are regular (Lemma 4.6.1) does not hold here.

Already, the minimization of the makespan is not as trivial as in the case of an unlimited batch size. Consider the problem $1 \mid batch(b) \mid C_{\max}$. It is clear that in order to minimize the makespan it suffices to determine how the n jobs are combined with one another and assembled into batches; the order in which the batches are processed does not affect the makespan. A schedule that minimizes the makespan consists of $N = \lceil (n/b) \rceil$ batches. In the next algorithm J denotes at any point in time the set of jobs that remain to be scheduled.

Algorithm 4.6.6 (Minimizing Makespan – Batch Size Finite)

Step 1. (Initialization)

Set $J = \{1,\ldots,n\}$ *and* $k = N$.

Step 2. (Batch Assembly)

If $k > 1$, take from Set J the b jobs with the longest processing times and put them in batch k.
If $k = 1$, put all jobs still remaining in set J (i.e., $n - (N-1)b$ jobs) in one batch and STOP.

Step 3. (Update Counter)

Remove the jobs that have been put in batch k from set J; reduce k by 1 and return to Step 2. ||

So the algorithm starts with the assembly of the b longest jobs in the first batch; it proceeds with selecting among the remaining jobs the b longest ones and putting them in a second batch, and so on. If n is not a multiple of b, then the last batch (containing the shortest jobs) will not be a full batch. So there exists an optimal schedule with all batches, with the exception of one, being full. This full-batch property applies only to the makespan objective.

Other objective functions tend to be significantly harder than the makespan objective. The problem $1 \mid batch(b) \mid \sum C_j$ is already not very easy. However, some structural results can be obtained. Assume again that the jobs are ordered such that $p_1 \leq p_2 \leq \cdots \leq p_n$. Two batches are said to be not intertwined if either the longest job in the first batch is smaller than the shortest job in the second batch or if the shortest job in the first batch is longer than the longest job in the second batch.

Lemma 4.6.7. *There exists an optimal schedule for $1 \mid batch(b) \mid \sum C_j$ with no two batches intertwined.*

Proof. The proof is easy and is left as an exercise (see Exercise 4.23). □

Note that in an *SPT-batch* schedule for the case $b = \infty$ (see Lemma 4.6.1) no two batches are intertwined either. However, it is clear that the property described in Lemma 4.6.7 is weaker than the property described in Lemma 4.6.1 for unlimited batch sizes. Lemma 4.6.1 implies also that in an optimal schedule a batch of jobs with smaller processing times must precede a batch of jobs with longer processing times. If the batch size is finite, then this is not necessarily the case. Lemma 4.6.7 may still allow a batch of jobs with longer processing times to precede a batch of jobs with shorter processing times. The batch sequence depends now also on the numbers of jobs in the batches.

Let $p(\mathcal{B}_k)$ denote the maximum processing time of the jobs in batch k, i.e., $p(\mathcal{B}_k)$ is the time required to process batch k. Let $|\mathcal{B}_k|$ denote the number of jobs in batch k. The following lemma describes an important property of the optimal batch sequence.

Lemma 4.6.8. *A batch schedule for $1 \mid batch(b) \mid \sum C_j$ is optimal if and only if the batches are sequenced in decreasing order of $|\mathcal{B}_k|/p(\mathcal{B}_k)$.*

Proof. The proof is easy and similar to the proof of optimality of the WSPT rule for the total weighted completion time objective when there are no batches (see Theorem 3.1.1). A batch now corresponds to a job in Theorem 3.1.1. The processing time of a batch corresponds to the processing time of a job and the number of jobs in a batch corresponds to the weight of a job. □

Clearly, it may be possible for a batch with a long processing time to *precede* a batch with a short processing time; the batch with the long processing time must then contain more jobs than the batch with the short processing time.

A batch is said to be *full* if it contains exactly b jobs; otherwise it is *non-full*. Batch $\mathcal{B}_k$ is said to be *deferred* with respect to batch $\mathcal{B}_\ell$ if $p(\mathcal{B}_k) < p(\mathcal{B}_\ell)$ and $\mathcal{B}_k$ is sequenced after $\mathcal{B}_\ell$.

Lemma 4.6.9. *In any optimal schedule, there is no batch that is deferred with respect to a non-full batch.*

Proof. The proof is easy and is left as an exercise (see Exercise 4.24). □

So this lemma basically says that in an optimal schedule no non-full batch can precede a batch that has a smaller processing time.

In order to determine the optimal schedule it suffices to consider schedules that satisfy the properties described above: each batch contains jobs with consecutive indices, batches are ordered in decreasing order of $|\mathcal{B}_k|/p(\mathcal{B}_k)$, and no batch is deferred with respect to a non-full batch. There exists a rather involved dynamic program for this problem that runs in $O(n^{b(b-1)})$, i.e., polynomial as long as the batch size b is fixed. It is also possible to design fairly effective heuristics using the theoretical properties shown above. A heuristic must assemble the jobs in clusters of at most b. It must try to keep the differences in the processing times of the jobs in a batch somewhat small and then order the batches in decreasing order of $|\mathcal{B}_k|/p(\mathcal{B}_k)$.

Example 4.6.10 (Minimizing Total Completion Time - Batch Size Finite)

Consider a machine that allows a maximum batch size of b. Assume there are k ($k < b$) jobs with processing time 1 and b jobs with processing time p ($p > 1$). If the k jobs with processing time 1 are scheduled first as one batch followed by a second batch containing the b jobs with processing time p, then the total completion time is $k + b(p + 1)$. If the two batches are reversed, then the total completion time is $bp + k(p + 1)$.

The total completion time of the first sequence is lower than the total completion time of the second sequence when $b < kp$. It is, of course, easy to find numerical examples where the second sequence has a lower total completion time. ||

A fair amount of research has also been done on finite batch scheduling with due date related objectives. However, most finite batch scheduling problems

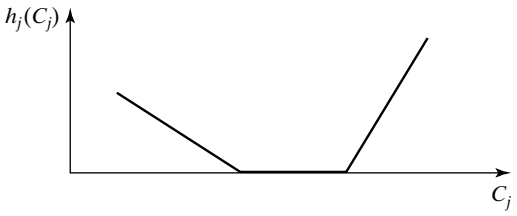


Fig. 4.8 Cost function with due date range

with due date related objective functions are strongly NP-Hard, including $1 \mid batch(b) \mid L_{\max}$, $1 \mid batch(b) \mid \sum U_j$, and $1 \mid batch(b) \mid \sum T_j$.

4.7 Discussion

Over the last decade problems with earliness and tardiness penalties have received a significant amount of attention. Even more general problems than those considered in this chapter have been studied. For example, some research has focused on problems with jobs that are subject to penalty functions such as the one presented in Figure 4.8.

Because of the importance of multiple objectives in practice a considerable amount of research has been done on problems with multiple objectives. Of course, these problems are harder than the problems with just a single objective. So, most problems with two objectives are NP-hard. These types of problems may attract in the near future the attention of investigators who specialize in PTAS and FPTAS.

The makespan minimization problem when the jobs are subject to sequence dependent setup times turns out to be equivalent to the Travelling Salesman Problem. Many combinatorial problems inspired by real world settings are equivalent to Travelling Salesman Problems. Another scheduling problem that is discussed in Chapter 6 is also equivalent to the particular Travelling Salesman Problem described in Section 4.4.

The models in the section focusing on job families are at times also referred to as batch scheduling models. Every time the machine has to be set up for a new family it is said that a batch of a particular family is about to start. This batch of jobs from that family are processed *sequentially*. This is in contrast to the setting in the last section where a batch of jobs is processed in *parallel* on the batch processing machine.

Exercises (Computational)

4.1. Consider the following instance with 6 jobs and $d = 156$.

jobs	1	2	3	4	5	6
p_j	4	18	25	93	102	114

Apply Algorithm 4.1.4 to find a sequence. Is the sequence generated by the heuristic optimal?

4.2. Consider the following instance with 7 jobs. For each job $w'_j = w''_j = w_j$. However, w_j is not necessarily equal to w_k.

jobs	1	2	3	4	5	6	7
p_j	4	7	5	9	12	2	6
w_j	4	7	5	9	12	2	6

All seven jobs have the same due date $d = 26$. Find all optimal sequences.

4.3. Consider again the instance of the previous exercise with 7 jobs. Again, for each job $w'_j = w''_j = w_j$. However, w_j is not necessarily equal to w_k. However, now the jobs have different due dates.

jobs	1	2	3	4	5	6	7
p_j	4	7	5	9	12	2	6
w_j	4	7	5	9	12	2	6
d_j	6	12	24	28	35	37	42

Find the optimal job sequence.

4.4. Give a numerical example of an instance with at most five jobs for which Algorithm 4.1.4 does not yield an optimal solution.

4.5. Consider the following instance of the $1 \mid\mid \sum w_j C_j^{(1)}, L_{\max}^{(2)}$ problem.

jobs	1	2	3	4	5
w_j	4	6	2	4	20
p_j	4	6	2	4	10
d_j	14	18	18	22	0

Find all optimal schedules.

4.6. Consider the following instance of the $1 \mid\mid L_{\max}^{(1)}, \sum w_j C_j^{(2)}$ problem.

jobs	1	2	3	4	5
w_j	4	6	2	4	20
p_j	4	6	2	4	10
d_j	14	18	18	22	0

Find all optimal schedules.

4.7. Apply Algorithm 4.3.2 to the following instance with 5 jobs and generate the entire trade-off curve.

jobs	1	2	3	4	5
p_j	4	6	2	4	2
d_j	2	4	6	10	10

4.8. Consider the instance of $1 \mid\mid \theta_1 L_{\max} + \theta_2 \sum C_j$ in Example 4.3.3. Find the ranges of θ_1 and θ_2 (assuming $\theta_1 + \theta_2 = 1$) for which each Pareto-optimal schedule minimizes $\theta_1 L_{\max} + \theta_2 \sum C_j$.

4.9. Consider an instance of the $1 \mid s_{jk} \mid C_{\max}$ problem with the sequence dependent setup times being of the form $s_{jk} = \mid a_k - b_j \mid$. The parameters a_j and b_k are in the table below. Find the optimal sequence.

cities	0	1	2	3	4	5	6
b_j	39	20	2	30	17	6	27
a_j	19	44	8	34	16	7	23

4.10. Consider the following instance of $1 \mid fmls, s_{gh} \mid \sum w_j C_j$ with $F = 2$. The sequence dependent setup times between the two families are $s_{12} = s_{21} = 2$ and $s_{01} = s_{02} = 0$. There are two jobs in family 1 and three jobs in family 2, i.e., $n_1 = 2$ and $n_2 = 3$. The processing times are in the table below:

jobs	(1,1)	(2,1)	(1,2)	(2,2)	(3,2)
p_{jg}	3	1	1	1	3
w_{jg}	27	2	30	1	1

Apply Algorithm 4.5.2 to find the optimal schedule.

Exercises (Theory)

4.11. Show that in an optimal schedule for an instance of $1 \mid d_j = d \mid \sum E_j + \sum T_j$ there is no unforced idleness in between any two consecutive jobs.

4.12. Prove Lemma 4.1.1.

4.13. Consider the single machine scheduling problem with objective $\sum w' E_j + \sum w'' T_j$ and all jobs having the same due date, i.e., $d_j = d$. Note that the weight of the earliness penalty w' is different from the weight of the tardiness penalty w'', but the penalty structure is the same for each job. Consider an instance where the due date d is so far out that the machine will not start processing any job at time zero. Describe an algorithm that yields an optimal solution (i.e., a generalization of Algorithm 4.1.3).

4.14. Consider the same problem as described in the previous exercise. However, now the due date is not far out and the machine does have to start processing a job immediately at time zero. Describe a heuristic that would yield a good solution (i.e., a generalization of Algorithm 4.1.4).

4.15. Consider an instance where each job is subject to earliness and tardiness penalties and $w'_j = w''_j = w_j$ for all j. However, w_j is not necessarily equal to w_k. The jobs have different due dates. Prove or disprove that EDD minimizes the sum of the earliness and tardiness penalties.

4.16. Describe the optimal schedule for $1 \mid\mid \sum w_j C_j^{(1)}, L_{\max}^{(2)}$ and prove its optimality.

4.17. Describe the optimal schedule for $1 \mid\mid \sum w_j C_j^{(1)}, \sum U_j^{(2)}$ and prove its optimality.

4.18. Describe the algorithm for $1 \mid\mid L_{\max}^{(1)}, \sum w_j C_j^{(2)}$. That is, generalize Lemma 4.2.1 and Algorithm 4.2.2.

4.19. Show that the maximum number of Pareto-optimal solutions for $1 \mid\mid \theta_1 \sum C_j + \theta_2 L_{\max}$ is $n(n-1)/2$.

4.20. Describe the optimal schedule for $1 \mid\mid \theta_1 \sum U_j + \theta_2 L_{\max}$ under the agreeability conditions

$$d_1 \leq \cdots \leq d_n,$$

and

$$p_1 \leq \cdots \leq p_n.$$

4.21. Prove Lemma 4.5.4.

4.22. Prove Lemma 4.6.1.

4.23. Prove Lemma 4.6.7.

4.24. Prove Lemma 4.6.9.

Comments and References

The survey paper by Baker and Scudder (1990) focuses on problems with earliness and tardiness penalties. The text by Baker (1995) has one chapter dedicated to problems with earliness and tardiness penalties. There are various papers on timing algorithms when the optimal order of the jobs is a given. The optimal timing Algorithm 4.1.8 is based on the paper by Szwarc and Mukhopadhyay (1995). An algorithm to find the optimal order of the jobs as well as their optimal start times and completion times, assuming $w'_j = w''_j = 1$ for all j, is presented by Kim and Yano (1994). For more results on models with earliness and tardiness penalties, see Sidney (1977), Hall and Posner (1991), Hall, Kubiak and Sethi (1991), and Wan and Yen (2002).

A fair amount of research has been done on single machine scheduling with multiple objectives. Some single machine problems with two objectives allow for polynomial time solutions; see, for example, Emmons (1975), Van Wassenhove and Gelders (1980), Nelson, Sarin and Daniels (1986), Chen and Bulfin (1994), and Hoogeveen and Van de Velde (1995). Potts and Van Wassenhove (1983) as well as Posner (1985) consider the problem of minimizing the total weighted completion time with the jobs being subject to deadlines (this problem is strongly NP-hard). Chen and Bulfin (1993) present a detailed overview of the state of the art in multi-objective single machine scheduling. The book by T'kindt and Billaut (2002, 2006) is entirely focused on multi-objective scheduling.

The material in Section 4.4 dealing with the Travelling Salesman Problem is entirely based on the famous paper by Gilmore and Gomory (1964). For more results on scheduling with sequence dependent setup times see Bianco, Ricciardelli, Rinaldi and Sassano (1988), Tang (1990) and Wittrock (1990).

Scheduling with the jobs belonging to a given (fixed) number of families has received a fair amount of attention in the literature. At times, these types of models have also been referred to as batch scheduling models (since the consecutive processing of a set of jobs from the same family may be regarded as a batch). Monma and Potts (1989) discuss the complexity of these scheduling problems. An excellent overview of the literature on this topic is presented in the paper by Potts and Kovalyov (2000). Brucker (2004) in his book also considers this class of models and refers to it as s-batching (batching with jobs processed in series).

When the machine is capable of processing multiple jobs in parallel, the machine is often referred to as a batching machine. An important paper concerning batch processing and batching machines is the one by Brucker, Gladky, Hoogeveen, Kovalyov, Potts, Tautenhahn, and van de Velde (1998). Potts and Kovalyov (2000) provides for this class of models also an excellent survey. Brucker (2004) considers this class of models as well and refers to them as p-batching (batching with jobs processed in parallel).

Chapter 5

Parallel Machine Models (Deterministic)

A bank of machines in parallel is a setting that is important from both a theoretical and a practical point of view. From a theoretical point of view it is a generalization of the single machine, and a special case of the flexible flow shop. From a practical point of view, it is important because the occurrence of resources in parallel is common in the real world. Also, techniques for machines in parallel are often used in decomposition procedures for multi-stage systems.

In this chapter several objectives are considered. The three principal objectives are the minimization of the makespan, the total completion time, and the maximum lateness. With a single machine the makespan objective is usually only of interest when there are sequence dependent setup times; otherwise the makespan is equal to the sum of the processing times and is independent of the sequence. When dealing with machines in parallel the makespan becomes an objective of considerable interest. In practice, one often has to deal with the problem of balancing the load on machines in parallel; by minimizing the makespan the scheduler ensures a good balance.

One may actually consider the scheduling of parallel machines as a two step process. First, one has to determine which jobs have to be allocated to which machines; second, one has to determine the sequence of the jobs allocated to each machine. With the makespan objective only the allocation process is important.

M.L. Pinedo, *Scheduling*, DOI: 10.1007/978-0-387-78935-4_5,

With parallel machines, preemptions play a more important role than with a single machine. With a single machine preemptions usually only play a role when jobs are released at different points in time. In contrast, with machines in parallel, preemptions are important even when all jobs are released at the same time.

For most models considered in this chapter there are optimal schedules that are non-delay. However, if there are unrelated machines in parallel and the total completion time must be minimized without preemptions, then the optimal schedule may not be non-delay.

Most models considered in this chapter fall in the category of the so-called offline scheduling problems. In an offline scheduling problem all data (e.g., processing times, release dates, due dates) are known in advance and can be taken into account in the optimization process. In contrast, in an online scheduling problem, the problem data are not known a priori. The processing time of a job only becomes known the moment it is completed and a release date only becomes known the moment a job is released. Clearly, the algorithms for online scheduling problems tend to be quite different from the algorithms for offline scheduling problems. The last section in this chapter focuses on online scheduling of parallel machines.

The processing characteristics and constraints considered in this chapter include precedence constraints as well as the set functions M_j. Throughout this chapter it is assumed that $p_1 \geq \cdots \geq p_n$.

5.1 The Makespan without Preemptions

First, the problem $Pm \mid\mid C_{\max}$ is considered. This problem is of interest because minimizing the makespan has the effect of balancing the load over the various machines, which is an important objective in practice.

It is easy to see that $P2 \mid\mid C_{\max}$ is NP-hard in the ordinary sense as it is equivalent to PARTITION (see Appendix D). During the last couple of decades many heuristics have been developed for $Pm \mid\mid C_{\max}$. One such heuristic is described below.

The Longest Processing Time first (LPT) rule assigns at $t = 0$ the m longest jobs to the m machines. After that, whenever a machine is freed the longest job among those not yet processed is put on the machine. This heuristic tries to place the shorter jobs more towards the end of the schedule, where they can be used for balancing the loads.

In the next theorem an upper bound is presented for

$$\frac{C_{\max}(LPT)}{C_{\max}(OPT)},$$

where $C_{\max}(LPT)$ denotes the makespan of the LPT schedule and $C_{\max}(OPT)$ denotes the makespan of the (possibly unknown) optimal schedule. This type

of worst case analysis is of interest as it gives an indication of how well the heuristic is guaranteed to perform as well as the type of instances for which the heuristic performs badly.

Theorem 5.1.1. *For* $Pm \mid\mid C_{\max}$

$$\frac{C_{\max}(LPT)}{C_{\max}(OPT)} \leq \frac{4}{3} - \frac{1}{3m}.$$

Proof. By contradiction. Assume that there exists one or more counterexamples with the ratio *strictly* larger than $4/3 - 1/3m$. If more than one such counterexample exist, there must exist an example with the smallest number of jobs.

Consider this "smallest" counterexample and assume it has n jobs. This smallest counterexample has a useful property: under LPT the shortest job is the last job to start its processing and also the last job to finish its processing. That this is true can be seen as follows: first, under LPT by definition the shortest job is the last to start its processing. Also, if this job is not the last to complete its processing, the deletion of this smallest job will result in a counterexample with fewer jobs (the $C_{\max}(LPT)$ remains the same while the $C_{\max}(OPT)$ may remain the same or may decrease). So for the smallest counterexample the starting time of the shortest job under LPT is $C_{\max}(LPT) - p_n$. Since at this point in time all other machines are still busy it follows that

$$C_{\max}(LPT) - p_n \leq \frac{\sum_{j=1}^{n-1} p_j}{m}.$$

The right hand side is an upper bound on the starting time of the shortest job. This upper bound is achieved when scheduling the first $n-1$ jobs according to LPT results in each machine having exactly the same amount of processing to do. Now

$$C_{\max}(LPT) \leq p_n + \frac{\sum_{j=1}^{n-1} p_j}{m} = p_n(1 - \frac{1}{m}) + \frac{\sum_{j=1}^{n} p_j}{m}.$$

Since

$$C_{\max}(OPT) \geq \frac{\sum_{j=1}^{n} p_j}{m}$$

the following series of inequalities holds for the counterexample:

$$\frac{4}{3} - \frac{1}{3m} < \frac{C_{\max}(LPT)}{C_{\max}(OPT)} \leq \frac{p_n(1 - 1/m) + \sum_{j=1}^{n} p_j/m}{C_{\max}(OPT)}$$

$$= \frac{p_n(1 - 1/m)}{C_{\max}(OPT)} + \frac{\sum_{j=1}^{n} p_j/m}{C_{\max}(OPT)} \leq \frac{p_n(1 - 1/m)}{C_{\max}(OPT)} + 1.$$

Thus

$$\frac{4}{3} - \frac{1}{3m} < \frac{p_n(1-1/m)}{C_{\max}(OPT)} + 1$$

and

$$C_{\max}(OPT) < 3p_n.$$

Note that this last inequality is a *strict* inequality. This implies that for the smallest counterexample the optimal schedule may result in at most two jobs on each machine. It can be shown that if an optimal schedule is a schedule with at most two jobs on each machine then the LPT schedule is optimal and the ratio of the two makespans is equal to one (see Exercise 5.11.b). This contradiction completes the proof of the theorem. □

Example 5.1.2 (A Worst Case Example of LPT)

Consider 4 parallel machines and 9 jobs, whose processing times are given in the table below:

jobs	1	2	3	4	5	6	7	8	9
p_j	7	7	6	6	5	5	4	4	4

Scheduling the jobs according to LPT results in a makespan of 15. It can be shown easily that for this set of jobs a schedule can be found with a makespan of 12 (see Figure 5.1). This particular instance is thus a worst case when there are 4 machines in parallel. ||

What would the worst case be, if instead of LPT an arbitrary priority rule is used? Consider the case where at time $t = 0$ the jobs are put in an arbitrary list. Whenever a machine is freed the job that ranks, among the remaining jobs, highest on the list is put on the machine. It can be shown that the worst case of this arbitrary list rule satisfies the inequality

$$\frac{C_{\max}(LIST)}{C_{\max}(OPT)} \leq 2 - \frac{1}{m}.$$

(This result can be shown via arguments that are similar to the proof of Theorem 5.6.1 in the section on online scheduling.)

However, there are also several other heuristics for the $Pm \mid\mid C_{\max}$ problem that are more sophisticated than LPT and that have tighter worst-case bounds. These heuristics are beyond the scope of this book.

Consider now the same problem with the jobs subject to precedence constraints, i.e., $Pm \mid prec \mid C_{\max}$. From a complexity point of view this problem has to be at least as hard as the problem without precedence constraints. To obtain some insights into the effects of precedence constraints, a number of special cases have to be considered. The special case with a single machine

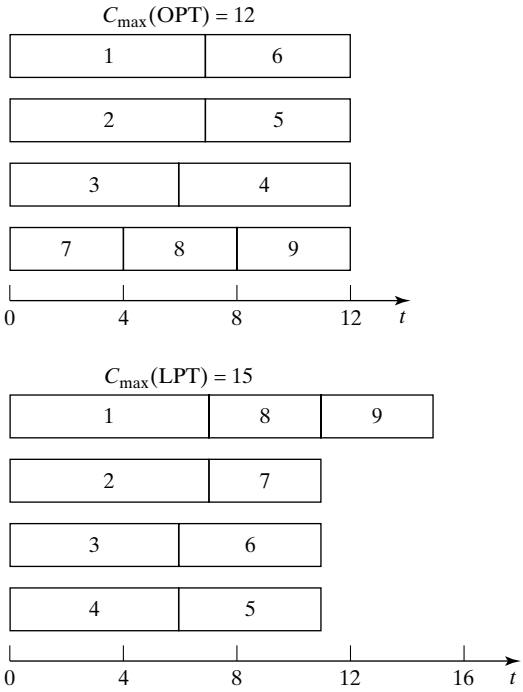


Fig. 5.1 Worst case example of LPT

is clearly trivial. It is enough to keep the machine continuously busy and the makespan will be equal to the sum of the processing times. Consider the special case where there are an unlimited number of machines in parallel, or where the number of machines is at least as large as the number of jobs, i.e., $m \geq n$. This problem may be denoted by $P\infty \mid prec \mid C_{\max}$. This is a classical problem in the field of project planning and its study has led to the development of the well-known *Critical Path Method (CPM)* and *Project Evaluation and Review Technique (PERT)*. The optimal schedule and the minimum makespan are determined through a very simple algorithm.

Algorithm 5.1.3 (Minimizing the Makespan of a Project)
Schedule the jobs one at a time starting at time zero. Whenever a job has been completed, start all jobs of which all predecessors have been completed (that is, all schedulable jobs.) ||

That this algorithm leads to an optimal schedule can be shown easily. The proof is left as an exercise. It turns out that in $P\infty \mid prec \mid C_{\max}$ the start of the processing of some jobs usually can be postponed without increasing the makespan. These jobs are referred to as the *slack* jobs. The jobs that cannot be postponed are referred to as the *critical* jobs. The set of critical jobs is referred to

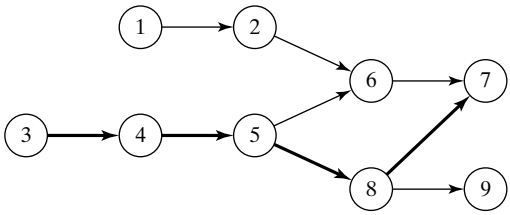


Fig. 5.2 Precedence constraints graph with critical path in Example 5.1.4

as the *critical path(s)*. In order to determine the critical jobs, perform the same procedure applied in Algorithm 5.1.3 backwards. Start at the makespan, which is now known, and work towards time zero, while adhering to the precedence relationships. Doing this, all jobs are completed at the latest possible completion times and therefore started at their latest possible starting times as well. Those jobs whose earliest possible starting times are equal to their latest possible starting times are the critical jobs.

Example 5.1.4 (Minimizing the Makespan of a Project)
Consider nine jobs with the following processing times.

jobs	1	2	3	4	5	6	7	8	9
p_j	4	9	3	3	6	8	8	12	6

The precedence constraints are depicted in Figure 5.2.

The earliest completion time C'_j of job j can be computed easily.

jobs	1	2	3	4	5	6	7	8	9
C'_j	4	13	3	6	12	21	32	24	30

This implies that the makespan is 32. Assuming that the makespan is 32, the latest possible completion times C''_j can be computed.

jobs	1	2	3	4	5	6	7	8	9
C''_j	7	16	3	6	12	24	32	24	32

Those jobs of which the earliest possible completion times are equal to the latest possible completion times are said to be on the critical path. So the critical path is the chain

$$3 \rightarrow 4 \rightarrow 5 \rightarrow 8 \rightarrow 7.$$

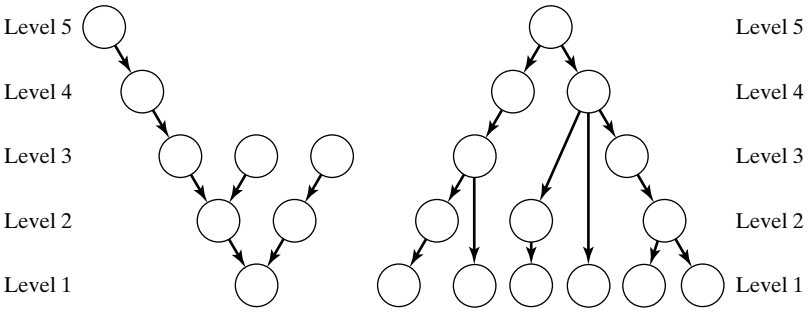


Fig. 5.3 Intree and outtree

The critical path in this case happens to be unique. The jobs that are not on the critical path are said to be slack. The amount of slack time for job j is the difference between its latest possible completion time and its earliest possible completion time. ||

In contrast to $1 \mid prec \mid C_{\max}$ and $P\infty \mid prec \mid C_{\max}$, the $Pm \mid prec \mid C_{\max}$ is strongly NP-hard when $2 \leq m < n$. Even the special case with all processing times equal to 1, i.e., $Pm \mid p_j = 1, prec \mid C_{\max}$, is not easy. However, constraining the problem further and assuming that the precedence graph takes the form of a tree (either an intree or an outtree) results in a problem, i.e., $Pm \mid p_j = 1, tree \mid C_{\max}$, that is easily solvable. This particular problem leads to a well-known scheduling rule, the *Critical Path (CP)* rule, which gives the highest priority to the job at the head of the longest string of jobs in the precedence graph (ties may be broken arbitrarily).

Before presenting the results concerning $Pm \mid p_j = 1, tree \mid C_{\max}$ it is necessary to introduce some notation. Consider an intree. The single job with no successors is called the *root* and is located at *level* 1. The jobs immediately preceding the root are at level 2; the jobs immediately preceding the jobs at level 2 are at level 3, and so on. In an outtree all jobs with no successors are located at level 1. Jobs that have *only* jobs at level 1 as their *immediate* successors are said to be at level 2; jobs that have only jobs at levels 1 and 2 as their immediate successors are at level 3, and so on (see Figure 5.3). From this definition it follows that the CP rule is equivalent to the *Highest Level* first rule. The number of jobs at level l is denoted by $N(l)$. Jobs with no predecessors are referred to as *starting* jobs; the nodes in the graph corresponding to these jobs are often referred to in graph theory terminology as *leaves*. The highest level in the graph is denoted by $l_{\max}$. Let

$$H(l_{\max} + 1 - r) = \sum_{k=1}^{r} N(l_{\max} + 1 - k).$$

Clearly, $H(l_{\max}+1-r)$ denotes the total number of nodes at level $l_{\max}+1-r$ or higher, that is, at the highest r levels.

Theorem 5.1.5. *The CP rule is optimal for* $Pm \mid p_j = 1, intree \mid C_{\max}$ *and for* $Pm \mid p_j = 1, outtree \mid C_{\max}$.

Proof. The proof for intrees is slightly harder than the proof for outtrees. In what follows only the proof for intrees is given (the proof for outtrees is left as an exercise). In the proof for intrees a distinction has to be made between two cases.

Case I. Assume the tree satisfies the following condition:

$$\max_r \Big(\frac{\sum_{k=1}^{r} N(l_{\max}+1-k)}{r} \Big) \leq m.$$

In this case, in every time interval, all the jobs available for processing can be processed and at most $l_{\max}$ time units are needed to complete all the jobs under the CP rule. But $l_{\max}$ is clearly a lower bound for $C_{\max}$. So the CP rule results in an optimal schedule.

Case II. Find for the tree the (smallest) integer $c \geq 1$ such that

$$\max_r \Big(\frac{\sum_{k=1}^{r} N(l_{\max}+1-k)}{r+c} \Big) \leq m < \max_r \Big(\frac{\sum_{k=1}^{r} N(l_{\max}+1-k)}{r+c-1} \Big).$$

The c basically represents the smallest amount of time beyond time r needed to complete all jobs at the r highest levels. Let

$$\max_r \Big(\frac{\sum_{k=1}^{r} N(l_{\max}+1-k)}{r+c-1} \Big) = \Big(\frac{\sum_{k=1}^{r^*} N(l_{\max}+1-k)}{r^*+c-1} \Big) > m.$$

The number of jobs completed at time (r^*+c-1) is at most $m(r^*+c-1)$. The number of jobs at levels higher than or equal to $l_{\max}+1-r^*$ is $\sum_{k=1}^{r^*} N(l_{\max}+1-k)$. As

$$\sum_{k=1}^{r^*} N(l_{\max}+1-k) > (r^*+c-1)m$$

there is at least one job at a level equal to or higher than $l_{\max}+1-r^*$ that is not processed by time r^*+c-1. Starting with this job there are at least $l_{\max}+1-r^*$ time units needed to complete all the jobs. A lower bound for the makespan under any type of scheduling rule is therefore

$$C_{\max} \geq (r^*+c-1) + (l_{\max}+1-r^*) = l_{\max}+c.$$

To complete the proof it suffices to show that the CP rule results in a makespan that is equal to this lower bound. This part of the proof is left as an exercise (see Exercise 5.14). □

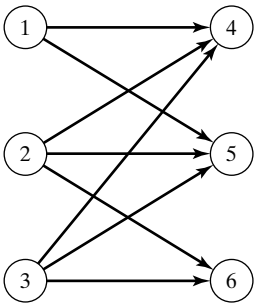


Fig. 5.4 Worst case example of the CP rule for two machines (Example 5.1.6)

The question arises: how well does the CP rule perform for arbitrary precedence constraints when all jobs have equal processing times? It has been shown that for two machines in parallel

$$\frac{C_{\max}(CP)}{C_{\max}(OPT)} \leq \frac{4}{3}.$$

When there are more than two machines in parallel, the worst case ratio is larger. That the worst case bound for two machines can be attained is shown in the following example.

Example 5.1.6 (A Worst-Case Example of CP)

Consider 6 jobs with unit processing times and two machines. The precedence relationships are depicted in Figure 5.4. Jobs 4, 5 and 6 are at level 1, while jobs 1, 2 and 3 are at level 2. As under the CP rule ties may be broken arbitrarily, a CP schedule may prescribe at time zero to start with jobs 1 and 2. At their completion only job 3 can be started. At time 2 jobs 4 and 5 are started. Job 6 goes last and is completed by time 4. Of course, an optimal schedule can be obtained by starting out at time zero with jobs 2 and 3. The makespan then equals 3. ||

Example 5.1.6 shows that processing the jobs with the largest number of successors first may result in a better schedule than processing the jobs at the highest level first. A priority rule often used when jobs are subject to arbitrary precedence constraints is indeed the so-called *Largest Number of Successors first (LNS)* rule. Under this rule the job with the largest total number of successors (not just the immediate successors) in the precedence constraints graph has the highest priority. Note that in the case of intrees the CP rule and the LNS rule are equivalent; the LNS rule therefore results in an optimal schedule in the case of intrees. It can be shown fairly easily that the LNS rule is also optimal for $Pm \mid p_j = 1, outtree \mid C_{\max}$. The following example shows that the LNS rule may not yield an optimal schedule with arbitrary precedence constraints.

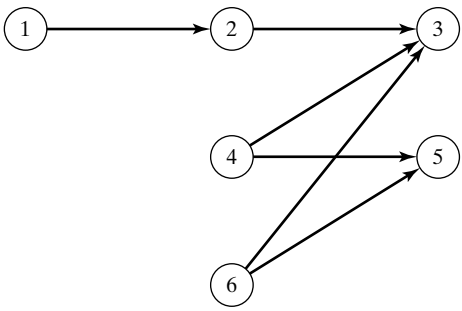


Fig. 5.5 The LNS rule is not necessarily optimal with arbitrary precedence constraints (Example 5.1.7)

Example 5.1.7 (Application of the LNS rule)
Consider 6 jobs with unit processing times and two machines. The precedence constraints are depicted in Figure 5.5. The LNS rule may start at time 0 with jobs 4 and 6. At time 1 jobs 1 and 5 start. Job 2 starts at time 2 and job 3 starts at time 3. The resulting makespan is 4. It is easy to see that the optimal makespan is 3 and that the CP rule actually achieves the optimal makespan. ||

Both the CP rule and the LNS rule have more generalized versions that can be applied to problems with arbitrary job processing times. Instead of counting the *number* of jobs (as in the case with unit processing times), these more generalized versions prioritize based on the total amount of processing remaining to be done on the jobs in question. The CP rule then gives the highest priority to the job that is heading the string of jobs with the largest total amount of processing (with the processing time of the job itself also being included in this total). The generalization of the LNS rule gives the highest priority to that job that precedes the largest total amount of processing; again the processing time of the job itself is also included in the total. The LNS name is clearly not appropriate for this generalization with arbitrary processing times, as it refers to a number of jobs rather than to a total amount of processing.

Another generalization of the $Pm \mid\mid C_{\max}$ problem that is of practical interest arises when job j is only allowed to be processed on subset M_j of the m parallel machines. Consider $Pm \mid p_j = 1, M_j \mid C_{\max}$ and assume that the sets M_j are *nested,* that is, one and only one of the following four conditions holds for jobs j and k.

(i) M_j is equal to M_k $(M_j = M_k)$
(ii) M_j is a subset of M_k $(M_j \subset M_k)$
(iii) M_k is a subset of M_j $(M_k \subset M_j)$
(iv) M_j and M_k do not overlap $(M_j \cap M_k = \emptyset)$

Under these conditions a well-known dispatching rule, the *Least Flexible Job first (LFJ)* rule, plays an important role. The LFJ rule selects, every time a machine is freed, among the available jobs the job that can be processed on the *smallest* number of machines, i.e., the least flexible job. Ties may be broken arbitrarily. This rule is rather crude as it does not specify, for example, which machine should be considered first when several machines become available at the same time.

Theorem 5.1.8. *The LFJ rule is optimal for* $Pm \mid p_j = 1, M_j \mid C_{\max}$ *when the* M_j *sets are nested.*

Proof. By contradiction. Suppose the LFJ rule does not yield an optimal schedule. Consider a schedule that is optimal, and that differs from a schedule that could have been generated via the LFJ rule. Without loss of generality, the jobs on each machine can be ordered in increasing order of $\mid M_j \mid$. Now consider the earliest time slot that is filled by a job that could not have been placed there by the LFJ rule. Call this job job j. There exists some job, say job j^*, that is scheduled to start later, and that could have been scheduled by LFJ in the position taken by job j. Note that job j^* is less flexible than job j (since $M_j \supset M_{j^*}$). Job j^* cannot start before job j, since j is the earliest non-LFJ job. It cannot start at the same time as job j, for in that case job j is in a position where LFJ could have placed it. These two jobs, j and j^*, can be exchanged without altering $C_{\max}$, since $p_j = 1$ for all jobs. Note that the slot previously filled by j is now filled by j^*, which is an LFJ position. By repeating this process of swapping the earliest non-LFJ job, all jobs can be moved into positions where they could have been placed using LFJ, without increasing $C_{\max}$. So it is possible to construct an LFJ schedule from an optimal schedule. This establishes the contradiction. □

It can be shown easily that the LFJ rule is optimal for $P2 \mid p_j = 1, M_j \mid C_{\max}$ because with two machines the M_j sets are always nested. However, with three or more machines the LFJ rule may not yield an optimal solution for $Pm \mid p_j = 1, M_j \mid C_{\max}$ with arbitrary M_j, as illustrated in the following example.

Example 5.1.9 (Application of the LFJ Rule)

Consider $P4 \mid p_j = 1, M_j \mid C_{\max}$ with eight jobs. The eight M_j sets are:

$$M_1 = \{1, 2\}$$

$$M_2 = M_3 = \{1, 3, 4\}$$

$$M_4 = \{2\}$$

$$M_5 = M_6 = M_7 = M_8 = \{3, 4\}$$

The M_j sets are clearly not nested. Under the LFJ rule the machines can be considered in any order. Consider machine 1 first. The least flexible job that can be processed on machine 1 is job 1 as it can be processed on only

two machines (jobs 2 and 3 can be processed on three machines). Consider machine 2 next. The least flexible job to be processed on machine 2 is clearly job 4. Least flexible jobs to be processed on machines 3 and 4 at time 0 could be jobs 5 and 6. At time 1, after jobs 1, 4, 5 and 6 have completed their processing on the four machines, the least flexible job to be processed on machine 1 is job 2. However, at this point none of the remaining jobs can be processed on machine 2; so machine 2 remains idle. The least flexible jobs to go on machines 3 and 4 are jobs 7 and 8. This implies that job 3 only can be started at time 2, completing its processing at time 3. The makespan is therefore equal to 3.

A better schedule with a makespan equal to 2 can be obtained by assigning jobs 2 and 3 to machine 1; jobs 1 and 4 to machine 2; jobs 5 and 6 to machine 3 and jobs 7 and 8 to machine 4. ||

From Example 5.1.9 one may expect that, if a number of machines are free at the same point in time, it is advantageous to consider first the least flexible *machine*. The flexibility of a machine could be defined as the number of remaining jobs that can be processed (or the total amount of processing that can be done) on that machine. Assigning at each point in time first a job, any job, to the *Least Flexible Machine (LFM)*, however, does not guarantee an optimal schedule in the case of Example 5.1.9.

Heuristics can be designed that combine the LFJ rule with the LFM rule, giving priority to the least flexible jobs on the least flexible machines. That is, consider at each point in time first the Least Flexible Machine (LFM) (that is, the machine that can process the smallest number of jobs) and assign to this machine the least flexible job that can be processed on it. Any ties may be broken arbitrarily. This heuristic may be referred to as the LFM-LFJ heuristic. However, in the case of Example 5.1.9 the LFM-LFJ does not yield an optimal schedule either.

5.2 The Makespan with Preemptions

Consider the same problem as the one discussed in the beginning of the previous section, but now with preemptions allowed, i.e., $Pm \mid prmp \mid C_{\max}$. Usually, but not always, allowing preemptions simplifies the analysis of a problem. This is indeed the case for this problem where it actually turns out that many schedules are optimal. First, consider the following linear programming formulation of the problem.

$$\textit{minimize} \quad C_{\max}$$

subject to

$$\sum_{i=1}^{m} x_{ij} = p_j, \qquad j = 1, \ldots, n$$

$$\sum_{i=1}^{m} x_{ij} \leq C_{\max}, \qquad j = 1, \ldots, n$$

$$\sum_{j=1}^{n} x_{ij} \leq C_{\max}, \qquad i = 1, \ldots, m$$

$$x_{ij} \geq 0, \qquad i = 1, \ldots, m, \qquad j = 1, \ldots, n.$$

The variable x_{ij} represents the total time job j spends on machine i. The first set of constraints makes sure that each job receives the required amount of processing. The second set of constraints ensures that the total amount of processing each job receives is less than or equal to the makespan. The third set makes sure that the total amount of processing on each machine is less than the makespan. Since the $C_{\max}$ basically is a decision variable and not an element of the resource vector of the linear program, the second and third set of constraints may be rewritten as follows:

$$C_{\max} - \sum_{i=1}^{m} x_{ij} \geq 0, \qquad j = 1, \ldots, n$$

$$C_{\max} - \sum_{j=1}^{n} x_{ij} \geq 0, \qquad i = 1, \ldots, m$$

Example 5.2.1 (LP Formulation for Minimizing Makespan with Preemptions)

Consider two machines and three jobs with $p_1 = 8, \quad p_2 = 7$ and $p_3 = 5$. There are thus 7 variables, namely x_{11}, x_{21}, x_{12}, x_{22}, x_{13}, x_{23} and $C_{\max}$ (see Appendix A). The $\mathbf{A}$ matrix is a matrix of 0's and 1's. The $\bar{c}$ vector contains six 0's and a single 1. The $\bar{b}$ vector contains the three processing times and five 0's. ||

This LP can be solved in polynomial time, but the solution of the LP does not prescribe an actual schedule; it merely specifies the amount of time job j should spend on machine i. However, with this information a schedule can easily be constructed.

There are several other algorithms for $Pm \mid prmp \mid C_{\max}$. One of these algorithms is based on the fact that it is easy to obtain an expression for the makespan under the optimal schedule. In the next lemma a lower bound is established.

Lemma 5.2.2. *Under the optimal schedule for* $Pm \mid prmp \mid C_{\max}$

$$C_{\max} \geq \max\left(p_1, \sum_{j=1}^{n} p_j/m\right) = C^*_{\max}.$$

Proof. Recall that job 1 is the job with the longest processing time. The proof is easy and left as an exercise. □

Having a lower bound allows for the construction of a very simple algorithm that minimizes the makespan. The fact that this algorithm actually produces a schedule with a makespan that is equal to the lower bound shows that the algorithm yields an optimal schedule.

Algorithm 5.2.3 (Minimizing Makespan with Preemptions)

Step 1.

Take the n jobs and process them one after another on a single machine in any sequence. The makespan is then equal to the sum of the n processing times and is less than or equal to $mC^*_{\max}$.

Step 2.

Take this single machine schedule and cut it into m parts. The first part constitutes the interval $[\,0,\, C^*_{\max}\,]$, *the second part the interval* $[\,C^*_{\max},\, 2C^*_{\max}\,]$, *the third part the interval* $[\,2C^*_{\max},\, 3C^*_{\max}\,]$, *etc.*

Step 3.

Take as the schedule for machine 1 in the bank of parallel machines the processing sequence of the first interval; take as the schedule for machine 2 the processing sequence of the second interval, and so on. ||

It is obvious that the resulting schedule is feasible. Part of a job may appear at the end of the schedule for machine i, while the remaining part may appear at the beginning of the schedule for machine $i+1$. As preemptions are allowed and the processing time of each job is less than $C^*_{\max}$ such a schedule is feasible. As this schedule has $C_{\max} = C^*_{\max}$, it is also optimal.

Another schedule that may appear appealing for $Pm \mid prmp \mid C_{\max}$ is the *Longest Remaining Processing Time first (LRPT)* schedule. This schedule is the preemptive counterpart of the (nonpreemptive) LPT schedule. It is a schedule that is structurally appealing, but mainly of academic interest. From a theoretical point of view it is important because of similarities with optimal policies in stochastic scheduling (see Chapter 12). From a practical point of view it has a serious drawback. The number of preemptions needed in the deterministic case is usually infinite.

Example 5.2.4 (Application of the LRPT Rule)

Consider 2 jobs with unit processing times and a single machine. Under LRPT the two jobs continuously have to rotate and wait for their next turn

on the machine (that is, a job stays on the machine for a time period ϵ and after every time period ϵ the job waiting preempts the machine). The makespan is equal to 2 and is, of course, independent of the schedule. But note that the sum of the two completion times under LRPT is 4, while under the nonpreemptive schedule it is 3. ||

In the subsequent lemma and theorem a proof technique is used that is based on a discrete time framework. All processing times are assumed to be integer and the decision-maker is allowed to preempt any machine only at integer times $1, 2, \ldots$ The proof that LRPT is optimal is based on a dynamic programming induction technique that requires some special notation. Assume that at some integer time t the remaining processing times of the n jobs are $p_1(t), p_2(t), \ldots, p_n(t)$. Let $\bar{p}(t)$ denote this vector of processing times. In the proof two different vectors of remaining processing times at time t, say $\bar{p}(t)$ and $\bar{q}(t)$, are repeatedly compared to one another. The vector $\bar{p}(t)$ is said to *majorize* the vector $\bar{q}(t)$, $\bar{p}(t) \geq_m \bar{q}(t)$, if

$$\sum_{j=1}^{k} p_{(j)}(t) \geq \sum_{j=1}^{k} q_{(j)}(t),$$

for all $k = 1, \ldots, n$, where $p_{(j)}(t)$ denotes the jth largest element of vector $\bar{p}(t)$ and $q_{(j)}(t)$ denotes the jth largest element of vector $\bar{q}(t)$.

Example 5.2.5 (Vector Majorization)

Consider the two vectors $\bar{p}(t) = (4,8,2,4)$ and $\bar{q}(t) = (3,0,6,6)$. Rearranging the elements within each vector and putting these in decreasing order results in vectors (8,4,4,2) and (6,6,3,0). It can be verified easily that $\bar{p}(t) \geq_m \bar{q}(t)$. ||

Lemma 5.2.6. *If $\bar{p}(t) >_m \bar{q}(t)$ then LRPT applied to $\bar{p}(t)$ results in a makespan that is larger than or equal to the makespan obtained by applying LRPT to $\bar{q}(t)$.*

Proof. The proof is by induction on the *total* amount of remaining processing. In order to show that the lemma holds for $\bar{p}(t)$ and $\bar{q}(t)$, with total remaining processing time $\sum_{j=1}^{n} p_j(t)$ and $\sum_{j=1}^{n} q_j(t)$ respectively, assume as induction hypothesis that the lemma holds for all pairs of vectors with total remaining processing less than or equal to $\sum_{j=1}^{n} p_j(t) - 1$ and $\sum_{j=1}^{n} q_j(t) - 1$ respectively. The induction base can be checked easily by considering the two vectors $1, 0, \ldots, 0$ and $1, 0, \ldots, 0$.

If LRPT is applied for one time unit on $\bar{p}(t)$ and $\bar{q}(t)$, respectively, then the vectors of remaining processing times at time $t+1$ are $\bar{p}(t+1)$ and $\bar{q}(t+1)$, respectively. Clearly,

$$\sum_{j=1}^{n} p_{(j)}(t+1) \leq \sum_{j=1}^{n} p_{(j)}(t) - 1$$

and

$$\sum_{j=1}^{n} q_{(j)}(t+1) \leq \sum_{j=1}^{n} q_{(j)}(t) - 1.$$

It can be shown that if $\bar{p}(t) \geq_m \bar{q}(t)$, then $\bar{p}(t+1) \geq_m \bar{q}(t+1)$. So if LRPT results in a larger makespan at time $t+1$ because of the induction hypothesis, it also results in a larger makespan at time t.

It is clear that if there are less than m jobs remaining to be processed, the lemma holds. □

Theorem 5.2.7. *LRPT yields an optimal schedule for* $Pm \mid prmp \mid C_{\max}$ *in discrete time.*

Proof. The proof is based on induction as well as on contradiction arguments.

The first step of the induction is shown as follows. Suppose not more than m jobs have processing times remaining and that these jobs all have only one unit of processing time left. Then clearly LRPT is optimal.

Assume LRPT is optimal for any vector $\bar{p}(t)$ for which

$$\sum_{j=1}^{n} p_{(j)}(t) \leq N - 1.$$

Consider now a vector $\bar{p}(t)$ for which

$$\sum_{j=1}^{n} p_{(j)}(t) = N.$$

The induction is based on the total amount of remaining processing, $N-1$, and *not* on the time t.

In order to show that LRPT is optimal for a vector of remaining processing times $\bar{p}(t)$ with a total amount of remaining processing $\sum_{j=1}^{n} p_j(t) = N$, assume that LRPT is optimal for all vectors with a smaller total amount of remaining processing. The proof of the induction step, showing that LRPT is optimal for $\bar{p}(t)$, is by contradiction. If LRPT is not optimal, another rule has to be optimal. This other rule does not act according to LRPT at time t, but from time $t+1$ onwards it must act according to LRPT because of the induction hypothesis (LRPT is optimal from $t+1$ on as the total amount of processing remaining at time $t+1$ is strictly less than N). Call this supposedly optimal rule, which between t and $t+1$ does not act according to LRPT, LRPT$'$. Now applying LRPT at time t on $\bar{p}(t)$ must be compared to applying LRPT$'$ at time t on the same vector $\bar{p}(t)$. Let $\bar{p}(t+1)$ and $\bar{p}'(t+1)$ denote the vectors of remaining processing times at time $t+1$ after applying LRPT and LRPT$'$. It is clear that $\bar{p}'(t+1) \geq_m \bar{p}(t+1)$. From Lemma 5.2.6 it follows that the makespan under LRPT$'$ is larger than the makespan under LRPT. This completes the proof of the induction step and the proof of the theorem. □

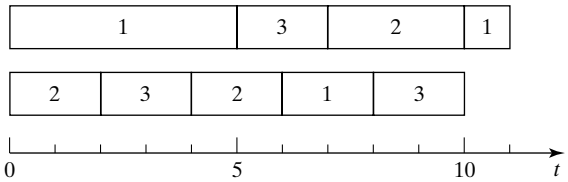


Fig. 5.6 LRPT with three jobs on two machines with preemptions allowed at integer points in time (Example 5.2.8)

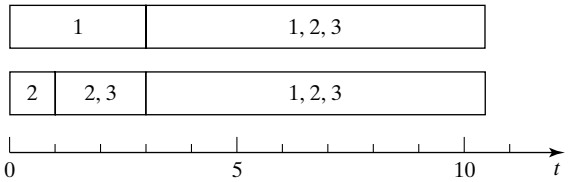


Fig. 5.7 LRPT with three jobs on two machines with preemptions allowed at any time (Example 5.2.9)

Example 5.2.8 (Application of LRPT in Discrete Time)

Consider two machines and three jobs, say jobs 1, 2 and 3, with processing times 8, 7 and 6. The schedule under LRPT is depicted in Figure 5.6 and the makespan is 11. ||

That LRPT is also optimal in continuous time (resulting in an infinite number of preemptions) can be argued easily. Multiply all processing times by K, K being a very large integer. The problem intrinsically does not change, as the relative lengths of the processing times remain the same. The optimal policy is, of course, again LRPT. But now there may be many more preemptions (recall preemptions must occur at integral time units). Basically, multiplying all processing times by K has the effect that the time slots become smaller relative to the processing times and the decision-maker is allowed to preempt after shorter intervals. Letting K go to ∞ shows that LRPT is optimal in continuous time as well.

Example 5.2.9 (Application of LRPT in Continuous Time)

Consider the same jobs as in the previous example. As preemptions may be done at any point in time, processor sharing takes place, see Figure 5.7. The makespan is now 10.5. ||

Consider the generalization to uniform machines, that is, m machines in parallel with machine i having speed v_i. Without loss of generality it may be assumed that $v_1 \geq v_2 \geq \cdots \geq v_m$. Similar to Lemma 5.2.2 a lower bound can be established for the makespan here as well.

Lemma 5.2.10. *Under the optimal schedule for* $Qm \mid prmp \mid C_{\max}$

$$C_{\max} \geq \max\left(\frac{p_1}{v_1}, \frac{p_1+p_2}{v_1+v_2}, \ldots, \frac{\sum_{j=1}^{m-1} p_j}{\sum_{i=1}^{m-1} v_i}, \frac{\sum_{j=1}^{n} p_j}{\sum_{i=1}^{m} v_i} \right)$$

Proof. The makespan has to be at least as large as the time it takes for the fastest machine to do the longest job. This time represents the first term within the "max" on the R.H.S. The makespan also has to be at least as large as the time needed for the fastest and second fastest machine to process the longest and second longest job while keeping the two machines occupied exactly the same amount of time. This amount of time represents the second term within the "max" expression. The remainder of the first $m-1$ terms are determined in the same way. The last term is a bit different as it is the minimum time needed to process all n jobs on the m machines while keeping all the m machines occupied exactly the same amount of time. □

If the largest term in the lower bound is determined by the sum of the processing times of the k longest jobs divided by the sum of the speeds of the k fastest machines, then the $n-k$ smallest jobs under the optimal schedule do not receive any processing on the k fastest machines; these jobs only receive processing on the $m-k$ slowest machines.

Example 5.2.11 (Minimizing Makespan on Uniform Machines)

Consider three machines, 1, 2 and 3, with respective speeds 3, 2 and 1. There are three jobs, 1, 2 and 3, with respective processing times 36, 34 and 12. The optimal schedule assigns the two longest jobs to the two fastest machines. Job 1 is processed for 8 units of time on machine 1 and for 6 units of time on machine 2, while job 2 is processed for 8 units of time on machine 2 and for 6 units of time on machine 1. These two jobs are completed after 14 time units. Job 3 is processed only on machine 3 and is completed at time 12. ||

A generalization of the LRPT schedule described before is the so-called *Longest Remaining Processing Time on the Fastest Machine first (LRPT-FM)* rule. This rule assigns, at any point in time, the job with the longest remaining processing time to the fastest machine; the job with the second longest remaining processing time to the second fastest machine, and so on.

This rule typically requires an infinite number of preemptions. If at time t two jobs have the same remaining processing time and this processing time is the longest among the jobs not yet completed by time t, then one of the two jobs has to go on the fastest machine while the other has to go on the second fastest. At time $t+\epsilon$, ϵ being very small, the remaining processing time of the job on the second fastest machine is longer than the remaining processing time of the job on the fastest machine. So the job on the second fastest machine has to move to the fastest and vice versa. Thus the LRPT-FM rule often results in so-called *processor sharing*. A number of machines, say m^*, process a number

of jobs, say n^*, $n^* \geq m^*$; the machines allocate their total processing capability uniformly over the n^* jobs and ensure that the remaining processing times of the n^* jobs remain equal at all times.

It can be shown that the LRPT-FM rule is indeed optimal for $Qm \mid prmp \mid C_{\max}$. It is again easier to construct first a proof of optimality in a discrete time framework and then extend the result to a continuous time setting. Assume that the processing times as well as the speeds are integers. Replace the original machine i that operates at speed v_i by v_i identical, so-called *unit-speed*, machines that operate at speed 1. A job may be processed for one time unit on a subset of the unit-speed machines that correspond to machine i. (It may not be processed at the same point in time by two unit-speed machines that do not correspond to the same original machine.)

Assume that preemptions are only allowed at integer times 1, 2, 3, ... If at time t job j has a remaining processing time p_j, its remaining processing time at time $t+1$ is equal to $(p_j - l)$ after being processed for one time unit on l unit-speed machines corresponding to the original machine i. Lemma 5.2.6 can be shown for this framework as well. Based on this lemma the following theorem can be shown.

Theorem 5.2.12. *LRPT-FM yields an optimal schedule for $Qm \mid prmp \mid C_{\max}$ in discrete time.*

Example 5.2.13 (Application of the LRPT-FM Rule)

Consider again 2 machines and three jobs. The processing times of the three jobs are again 8, 7 and 6. The two machines have now different speeds: $v_1 = 2$ and $v_2 = 1$. The first machine can be replaced by two machines with speed 1. Note that a job may be processed simultaneously by machines 1 and 2, but not simultaneously by machines 1 and 3 or 2 and 3. These three machines may be scheduled as follows: Job 1 is assigned to the first two unit-speed machines for one time unit and job 2 is assigned to the third unit-speed machine for one time unit. At time 1, all three jobs have a remaining processing time equal to 6. So from time 1 on, each job occupies one unit-speed machine and the makespan is equal to 7. ||

In order to show that LRPT-FM is also optimal in continuous time two limits have to be taken. First, all the processing times have to be multiplied by a large number K. In proportion to the new processing times a time unit is now very small. The possible number of preemptions goes up significantly this way, approaching preemptions in continuous time. Second, the speeds of the machines have to be multiplied with a large number as well. Thus the number of unit-speed machines for each original machine goes up dramatically. This implies that a finer partitioning of the processing capabilities of a given machine can be realized. Through such arguments it can be shown that LRPT-FM is optimal in continuous time. In addition, it can be shown that applying LRPT-FM to the available jobs when the jobs have different release dates is optimal as well (i.e., $Qm \mid r_j, prmp \mid C_{\max}$).

5.3 The Total Completion Time without Preemptions

Consider m machines in parallel and n jobs. Recall that $p_1 \geq \cdots \geq p_n$. The objective to be minimized is the total unweighted completion time $\sum C_j$. From Theorem 3.1.1 it follows that for a single machine the *Shortest Processing Time first (SPT)* rule minimizes the total completion time. This single machine result can also be shown in a different way fairly easily.

Let $p_{(j)}$ denote the processing time of the job in the jth position in the sequence. The total completion time can then be expressed as

$$\sum C_j = np_{(1)} + (n-1)p_{(2)} + \cdots + 2p_{(n-1)} + p_{(n)}.$$

This implies that there are n coefficients $n, n-1, \ldots, 1$ to be assigned to n different processing times. The processing times have to be assigned in such a way that the sum of the products is minimized. From the elementary Hardy, Littlewood and Polya inequality as well as common sense it follows that the highest coefficient, n, is assigned the smallest processing time, p_n, the second highest coefficient, $n-1$, is assigned the second smallest processing time, p_{n-1}, and so on. This implies that SPT is optimal.

This type of argument can be extended to the parallel machine setting as well.

Theorem 5.3.1. *The SPT rule is optimal for* $Pm \mid\mid \sum C_j$.

Proof. In the case of parallel machines there are nm coefficients to which processing times can be assigned. These coefficients are m n's, m $(n-1)$'s, ... , m ones. The processing times have to be assigned to a subset of these coefficients in order to minimize the sum of the products. Assume that n/m is an integer. If it is not an integer add a number of dummy jobs with zero processing times so that n/m is integer (adding jobs with zero processing times does not change the problem; these jobs would be instantaneously processed at time zero and would not contribute to the objective function). It is easy to see, in a similar manner as above, that the set of m longest processing times have to be assigned to the m ones, the set of second m longest processing times have to be assigned to the m twos, and so on. This results in the m longest jobs each being processed on different machines and so on. That this class of schedules includes SPT can be shown as follows. According to the SPT schedule the smallest job has to go on machine 1 at time zero, the second smallest one on machine 2, and so on; the $(m+1)$th smallest job follows the smallest job on machine 1, the $(m+2)$th smallest job follows the second smallest on machine 2, and so on. It is easy to verify that the SPT schedule corresponds to an optimal assignment of jobs to coefficients. □

From the proof of the theorem it is clear that the SPT schedule is not the only schedule that is optimal. Many more schedules also minimize the total completion time. The class of schedules that minimize the total completion time turns out to be fairly easy to characterize (see Exercise 5.21).

As pointed out in the previous chapter the more general WSPT rule minimizes the total *weighted* completion time in the case of a single machine. Unfortunately, this result cannot be generalized to parallel machines, as shown in the following example.

Example 5.3.2 (Application of the WSPT Rule)
Consider two machines and three jobs.

jobs	1	2	3
p_j	1	1	3
w_j	1	1	3

Scheduling jobs 1 and 2 at time zero and job 3 at time 1 results in a total weighted completion time of 14, while scheduling job 3 at time zero and jobs 1 and 2 on the other machine results in a total weighted completion time of 12. Clearly, with this set of data any schedule may be considered to be WSPT. However, making the weights of jobs 1 and 2 equal to $1 - \epsilon$ shows that WSPT does not necessarily yield an optimal schedule. ||

It has been shown in the literature that the WSPT heuristic is nevertheless a very good heuristic for the total weighted completion time on parallel machines. A worst case analysis of this heuristic yields the lower bound

$$\frac{\sum w_j C_j(WSPT)}{\sum w_j C_j(OPT)} < \frac{1}{2}(1+\sqrt{2}).$$

What happens now in the case of precedence constraints? The problem $Pm \mid prec \mid \sum C_j$ is known to be strongly NP-hard in the case of arbitrary precedence constraints. However, the special case with all processing times equal to 1 and precedence constraints that take the form of an outtree can be solved in polynomial time. In this special case the Critical Path rule again minimizes the total completion time.

Theorem 5.3.3. *The CP rule is optimal for* $Pm \mid p_j = 1, outtree \mid \sum C_j$.

Proof. Up to some integer point in time, say t_1, the number of schedulable jobs is less than or equal to the number of machines. Under the optimal schedule, at each point in time before t_1, all schedulable jobs have to be put on the machines. Such actions are in accordance with the CP rule. Time t_1 is the first point in time when the number of schedulable jobs is strictly larger than m. There are at least $m+1$ jobs available for processing and each one of these jobs is at the head of a subtree that includes a string of a given length.

The proof that applying CP from t_1 is optimal is by contradiction. Suppose that after time t_1 another rule is optimal. This rule must, at least once, prescribe

an action that is not according to CP. Consider the last point in time, say t_2, at which this rule prescribes an action not according to CP. So at t_2 there are m jobs, that are *not* heading the m longest strings, assigned to the m machines; from $t_2 + 1$ the CP rule is applied. Call the schedule from t_2 onwards CP′. It suffices to show that applying CP from t_2 onwards results in a schedule that is at least as good.

Consider under CP′ the longest string headed by a job that is *not* assigned at t_2, say string 1, and the shortest string headed by a job that *is* assigned at t_2, say string 2. The job at the head of string 1 has to start its processing under CP′ at time $t_2 + 1$. Let C_1' and C_2' denote the completion times of the last jobs of strings 1 and 2, respectively, under CP′. Under CP′ $C_1' \geq C_2'$. It is clear that under CP′ all m machines have to be busy at least up to $C_2' - 1$. If $C_1' \geq C_2' + 1$ and there are machines idle before $C_1' - 1$, the application of CP at t_2 results in less idle time and a smaller total completion time. Under CP the last job of string 1 is completed one time unit earlier, yielding one more completed job at or before $C_1' - 1$. In other cases the total completion time under CP is equal to the total completion time under CP′. This implies that CP is optimal from t_2 on. As there is not then a last time for a deviation from CP, the CP rule is optimal. □

In contrast to the makespan objective the CP rule is, somewhat surprisingly, not necessarily optimal for intrees. Counterexamples can be found easily (see Exercise 5.24).

Consider the problem $Pm \mid p_j = 1, M_j \mid \sum C_j$. Again, if the M_j sets are nested the Least Flexible Job first rule can be shown to be optimal.

Theorem 5.3.4. *The LFJ rule is optimal for* $Pm \mid p_j = 1, M_j \mid \sum C_j$ *when the* M_j *sets are nested.*

Proof. The proof is similar to the proof of Theorem 5.1.8. □

The previous model is a special case of $Rm \mid\mid \sum C_j$. As stated in Chapter 2, the machines in the Rm environment are entirely unrelated. That is, machine 1 may be able to process job 1 in a short time and may need a long time for job 2, while machine 2 may be able to process job 2 in a short time and may need a long time for job 1. That the Qm environment is a special case is clear. Identical machines in parallel with job j being restricted to machine set M_j is also a special case; the processing time of job j on a machine that is not part of M_j has to be considered very long making it therefore impossible to process the job on such a machine.

The $Rm \mid\mid \sum C_j$ problem can be formulated as an integer program with a special structure that makes it possible to solve the problem in polynomial time. Recall that if job j is processed on machine i and there are $k - 1$ jobs following job j on this machine i, then job j contributes kp_{ij} to the value of the objective function. Let x_{ikj} denote $0 - 1$ integer variables, where $x_{ikj} = 1$

if job j is scheduled as the kth to last job on machine i and 0 otherwise. The integer program is then formulated as follows:

$$\text{minimize} \qquad \sum_{i=1}^{m}\sum_{j=1}^{n}\sum_{k=1}^{n} kp_{ij}x_{ikj}$$

subject to

$$\sum_{i=1}^{m}\sum_{k=1}^{n} x_{ikj} = 1, \qquad j = 1,\ldots,n$$

$$\sum_{j=1}^{n} x_{ikj} \leq 1, \qquad i = 1,\ldots,m,\ \ k = 1,\ldots,n$$

$$x_{ikj} \in \{0,1\}, \qquad i = 1,\ldots,m,\ \ k = 1,\ldots,n\ \ j = 1,\ldots,n$$

The constraints make sure that each job is scheduled exactly once and each position on each machine is taken by at most one job. Note that the processing times only appear in the objective function.

This is a so-called weighted bipartite matching problem with on one side the n jobs and on the other side nm positions (each machine can process at most n jobs). If job j is matched with (assigned to) position ik there is a cost kp_{ij}. The objective is to determine the matching in this so-called bipartite graph with a minimum total cost. It is known from the theory of network flows that the integrality constraints on the x_{ikj} may be replaced by nonnegativity constraints without changing the feasible set. This weighted bipartite matching problem then reduces to a regular linear program for which there exist polynomial time algorithms. (see Appendix A).

Note that the optimal schedule does not have to be a non-delay schedule.

Example 5.3.5 (Minimizing Total Completion Time with Unrelated Machines)

Consider 2 machines and 3 jobs. The processing times of the three jobs on the two machines are presented in the table below.

jobs	1	2	3
p_{1j}	4	5	3
p_{2j}	8	9	3

The bipartite graph associated with this problem is depicted in Figure 5.8. According to the optimal schedule machine 1 processes job 1 first and job 2 second. Machine 2 processes job 3. This solution corresponds to

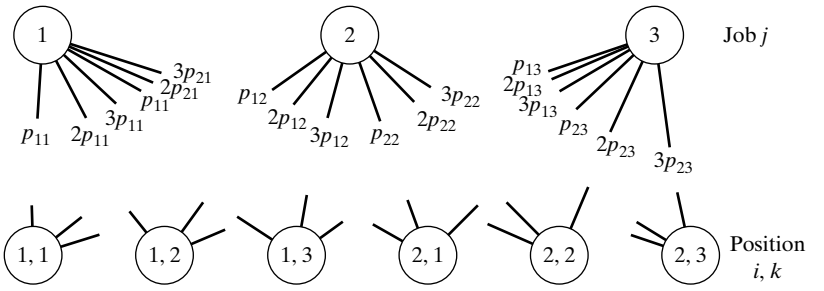


Fig. 5.8 Bipartite graph for $Rm \mid\mid \sum C_j$ with three jobs

$x_{121} = x_{112} = x_{213} = 1$ and all other x_{ikj} equal to zero. This optimal schedule is *not* non-delay (machine 2 is freed at time 3 and the waiting job is not put on the machine). ||

5.4 The Total Completion Time with Preemptions

In Theorem 5.3.1 it is shown that the nonpreemptive SPT rule minimizes $\sum C_j$ in a parallel machine environment. It turns out that the nonpreemptive SPT rule is also optimal when preemptions are allowed. This result is a special case of the more general result described below.

Consider m machines in parallel with different speeds, i.e., $Qm \mid prmp \mid \sum C_j$. This problem leads to the so-called *Shortest Remaining Processing Time on the Fastest Machine (SRPT-FM)* rule. According to this rule at any point in time the job with the shortest remaining processing time is assigned to the fastest machine, the second shortest remaining processing time on the second fastest machine, and so on. Clearly, this rule requires preemptions. Every time the fastest machine completes a job, the job on the second fastest machine moves to the fastest machine, the job on the third fastest machine moves to the second fastest machine, and so on. So, at the first job completion there are $m-1$ preemptions, at the second job completion there are $m-1$ preemptions, and so on until the number of remaining jobs is less than the number of machines. From that point in time the number of preemptions is equal to the number of remaining jobs.

The following lemma is needed for the proof.

Lemma 5.4.1. *There exists an optimal schedule under which $C_j \leq C_k$ when $p_j \leq p_k$ for all j and k.*

Proof. The proof is left as an exercise. □

Without loss of generality it may be assumed that there are as many machines as jobs. If the number of jobs is smaller than the number of machines then the

$m - n$ slowest machines are disregarded. If the number of jobs is larger than the number of machines, then $n - m$ machines are added with zero speeds.

Theorem 5.4.2. *The SRPT-FM rule is optimal for* $Qm \mid prmp \mid \sum C_j$.

Proof. Under SRPT-FM $C_n \leq C_{n-1} \leq \cdots \leq C_1$. It is clear that under SRPT-FM the following equations have to be satisfied:

$$\begin{aligned} v_1 C_n &= p_n \\ v_2 C_n + v_1(C_{n-1} - C_n) &= p_{n-1} \\ v_3 C_n + v_2(C_{n-1} - C_n) + v_1(C_{n-2} - C_{n-1}) &= p_{n-2} \\ &\vdots \\ v_n C_n + v_{n-1}(C_{n-1} - C_n) + \cdots + v_1(C_1 - C_2) &= p_1 \end{aligned}$$

Adding these equations yields the following set of equations.

$$\begin{aligned} v_1 C_n &= p_n \\ v_2 C_n + v_1 C_{n-1} &= p_n + p_{n-1} \\ v_3 C_n + v_2 C_{n-1} + v_1 C_{n-2} &= p_n + p_{n-1} + p_{n-2} \\ &\vdots \\ v_n C_n + v_{n-1} C_{n-1} + \cdots + v_1 C_1 &= p_n + p_{n-1} + \cdots + p_1 \end{aligned}$$

Suppose schedule $\mathcal{S}'$ is optimal. From the previous lemma it follows that

$$C'_n \leq C'_{n-1} \leq \cdots \leq C'_1.$$

The shortest job cannot be completed before p_n/v_1, i.e., $C'_n \geq p_n/v_1$ or

$$v_1 C'_n \geq p_n.$$

Given that jobs n and $n-1$ are completed at C'_n and C'_{n-1},

$$(v_1 + v_2)C'_n + v_1(C'_{n-1} - C'_n)$$

is an upper bound on the amount of processing that can be done on these two jobs. This implies that

$$v_2 C'_n + v_1 C'_{n-1} \geq p_n + p_{n-1}.$$

Continuing in this manner it is easy to show that

$$v_k C'_n + v_{k-1} C'_{n-1} + \cdots + v_1 C'_{n-k+1} \geq p_n + p_{n-1} + \cdots + p_{n-k+1}.$$

So

$$\begin{aligned} v_1 C'_n &\geq v_1 C_n \\ v_2 C'_n + v_1 C'_{n-1} &\geq v_2 C_n + v_1 C_{n-1} \\ &\vdots \\ v_n C'_n + v_{n-1} C'_{n-1} + \cdots + v_1 C'_1 &\geq v_n C_n + v_{n-1} C_{n-1} + \cdots + v_1 C_1 \end{aligned}$$

If a collection of positive numbers α_j can be found, such that multiplying the jth inequality by α_j and adding all inequalities yields the inequality $\sum C'_j \geq \sum C_j$, then the proof is complete. It can be shown that these α_j must satisfy the equations

$$\begin{aligned} v_1\alpha_1 + v_2\alpha_2 + \cdots + v_n\alpha_n &= 1 \\ v_1\alpha_2 + v_2\alpha_3 + \cdots + v_{n-1}\alpha_n &= 1 \\ &\vdots \\ v_1\alpha_n &= 1 \end{aligned}$$

As $v_1 \geq v_2 \geq \cdots \geq v_n$ such a collection does exist. □

Example 5.4.3 (Application of the SRPT-FM Rule)

Consider 4 machines and 7 jobs. The machine data and job data are contained in the two tables below.

machines	1	2	3	4
v_i	4	2	2	1

jobs	1	2	3	4	5	6	7
p_j	8	16	34	40	45	46	61

Assuming that preemptions are allowed only at integer points in time, the SRPT-FM rule results in the schedule depicted in Figure 5.9. Under this optimal schedule the total completion time is 116. ||

5.5 Due Date Related Objectives

Single machine problems with due date related objectives that are solvable in polynomial time typically have the maximum lateness as objective, e.g., $1 \mid\mid L_{\max}$, $1 \mid prmp \mid L_{\max}$ and $1 \mid r_j, prmp \mid L_{\max}$. Single machine problems with

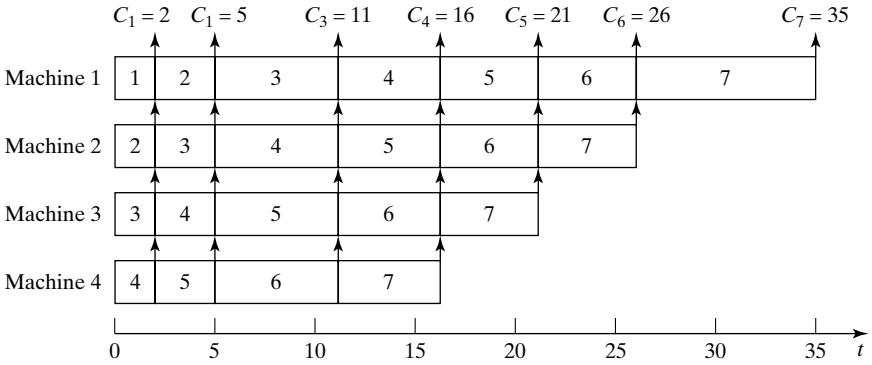


Fig. 5.9 The SRPT-FM schedule (Example 5.4.3)

the total tardiness or the total weighted tardiness as objective tend to be very hard.

It is easy to see that from a complexity point of view $Pm \mid\mid L_{\max}$ is not as easy as $1 \mid\mid L_{\max}$. Consider the special case where all jobs have due date 0. Finding a schedule with a minimum $L_{\max}$ is equivalent to $Pm \mid\mid C_{\max}$ and is therefore NP-hard.

Consider $Qm \mid prmp \mid L_{\max}$. This problem is one of the few parallel machine scheduling problems with a due date related objective that is solvable in polynomial time. Suppose one has to verify whether there exists a feasible schedule with $L_{\max} = z$. This implies that for job j the completion time C_j has to be less than or equal to $d_j + z$. Let $d_j + z$ be a hard deadline $\bar{d}_j$. Finding a feasible schedule with all jobs completing their processing before these deadlines is equivalent to solving the problem $Qm \mid r_j, prmp \mid C_{\max}$. In order to see this, reverse the direction of time in the due date problem. Apply the LRPT-FM rule starting with the last deadline and work backwards. The deadlines in the original problem play the role of the release dates in the reversed problem that is then equivalent to $Qm \mid r_j, prmp \mid C_{\max}$. If applying the LRPT-FM rule backwards results in a feasible schedule with all the jobs in the original problem starting at a time larger than or equal to zero, then there exists a schedule for $Qm \mid prmp \mid L_{\max}$ with $L_{\max} \leq z$. In order to find the minimum $L_{\max}$ a simple search has to be done to determine the appropriate minimum value of z.

Example 5.5.1 (Minimizing Maximum Lateness with Preemptions)
Consider the following instance of $P2 \mid prmp \mid L_{\max}$ with 4 jobs. The processing times and due dates are given in the table below. Preemptions are allowed at integer points in time.

jobs	1	2	3	4
d_j	4	5	8	9
p_j	3	3	3	8

First, it has to be checked whether there exists a feasible solution with $L_{\max} = 0$. The data of the instance created through time reversal are determined as follows. The release dates are obtained by determining the maximum due date in the original problem which is 9 and corresponds to job 4; the release date of job 4 in the new problem is then set equal to 0. The release dates of the remaining jobs are obtained by subtracting the original due dates from 9.

jobs	1	2	3	4
r_j	5	4	1	0
p_j	3	3	3	8

The question now is: in this new instance can a schedule be created with a makespan less than 9? Applying LRPT immediately yields a feasible schedule. ||

Consider now $Qm \mid r_j, prmp \mid L_{\max}$. Again a parametric study can be done. First an attempt is made to find a schedule with $L_{\max}$ equal to z. Due date d_j is replaced by a deadline $d_j + z$. Reversing this problem does not provide any additional insight as it results in a problem of the same type with release dates and due dates reversed. However, this problem still can be formulated as a network flow problem that is solvable in polynomial time.

5.6 Online Scheduling

In all previous sections the underlying assumptions were based on the fact that all the problem data (e.g., number of jobs, processing times, release dates, due dates, weights, and so on) are known in advance. The decision-maker can determine at time zero the entire schedule while having all the information at his disposal. This most common paradigm is usually referred to as *offline* scheduling.

One category of parallel machine scheduling problems that has not yet been addressed in this chapter are the so-called online scheduling problems. In an online scheduling problem the decision-maker does not know in advance how many jobs have to be processed and what the processing times are. The decision-maker becomes aware of the existence of a job only when the job is released and presented to him. Jobs that are released at the same point in time are presented to the decision-maker one after another. The decision-maker only knows the number of jobs released at that point in time after the last one has been presented to him. The processing time of a job becomes known only when the job has been completed. If the assumption is made that jobs are going to be released at different points in time, then the decision-maker does not know at any given point in time how many jobs are still going to be released and

what their release dates are going to be. (In an offline scheduling problem all information regarding all n jobs is known a priori.)

An online counterpart of $Pm \mid\mid \gamma$ can be described as follows. The jobs are going to be presented to the decision-maker one after another going down a list. The decision-maker only knows how long the list is when the end of the list has been reached. When a job has been presented to the decision-maker (or, equivalently, when the decision-maker has taken a job from the list), he may have to wait till one (or more) machines have become idle before he assigns the job to a machine. After he has assigned the job to a machine, the decision-maker can consider the next job on the list. After a job has been put on a machine starting at a certain point in time, the decision-maker is not allowed to preempt and has to wait till the job is completed. If the objective function is a regular performance measure, then it may not make sense for the decision-maker to leave a machine idle when there are still one or more jobs on the list.

The objective functions in online scheduling are similar to those in offline scheduling. The effectiveness of an online scheduling algorithm is measured by its *competitive ratio* with respect to the objective function. An online algorithm is ρ-competitive if for any problem instance the objective value of the schedule generated by the algorithm is at most ρ times larger than the optimal objective value in case the schedule had been created in an offline manner with all data known beforehand. The competitive ratio is basically equivalent to a worst case bound.

Consider the following online counterpart of $Pm \mid\mid C_{\max}$. There are a fixed number of machines (m) in parallel; this number is known to the decision-maker. The processing time of a job is at time zero not known to the decision-maker; it only becomes known upon the completion of a job. When a machine is freed the decision-maker has to decide whether to assign a job to that machine or keep it idle. He has to decide without knowing the remaining processing times of the jobs that are not yet completed and without knowing how many jobs are still waiting for processing. One well-known algorithm for this problem is usually referred to as the *List Scheduling (LIST)* algorithm. According to LIST, the jobs are presented to the decision-maker according to a list and every time the decision-maker considers the assignment of a job to a machine, he checks the list and takes the next one from the list. So, every time a machine completes a job, the decision maker takes the next job from the list and assigns it to that machine (the decision-maker does not allow for any idle time on the machine).

Theorem 5.6.1. *The competitive ratio of the LIST algorithm is* $2 - \frac{1}{m}$.

Proof. First, it has to be shown that the competitive ratio of LIST cannot be better (less) than $2 - 1/m$. Consider a sequence of $m(m-1)$ jobs with running time 1 followed by one job with running time m. A LIST schedule following this sequence finishes by time $2m - 1$, while the optimal schedule has a makespan of m.

In order to show that the competitive ratio cannot be larger than $2 - 1/m$, consider the job that finishes last. Suppose it starts at time t and its processing

time is p. At all times before t all machines must have been busy, otherwise the last job could have started earlier. Hence the optimal makespan $C_{\max}(OPT)$ must satisfy

$$C_{\max}(OPT) \geq t + \frac{p}{m}.$$

In addition, $C_{\max}(OPT) > p$, as the optimal schedule must process the last job. From these two inequalities, it follows that the makespan of the online solution, $t + p$, is bounded from above by

$$t + p \;=\; t + \frac{p}{m} + \Big(1 - \frac{1}{m}\Big)p \;\leq\; \Big(2 - \frac{1}{m}\Big)C_{\max}(OPT). \qquad \square$$

Consider now the online counterpart of $Pm \mid prmp \mid \sum C_j$. The decision-maker only finds out about the processing time of a job the moment it has been completed.

The following algorithm for this online scheduling problem is quite different from the LIST algorithm. The so-called Round Robin (RR) algorithm cycles through the list of jobs, giving each job a fixed unit of processing time in turn. The Round Robin algorithm ensures that at all times any two uncompleted jobs have received an equal amount of processing time or one job has received just one unit of processing more than the other. If the unit of processing is made very small, then the Round Robin rule becomes equivalent to the *Processor Sharing* rule (see Example 5.2.9). If the total completion time is the objective to be minimized, then the competitive ratio of RR can be determined.

Theorem 5.6.2. *The competitive ratio of the RR algorithm is* 2.

Proof. Assume, for the time being, that the number of jobs, n, is known. In what follows, it will actually be shown that the worst case ratio of RR is $2 - 2m/(n+m)$.

In order to show that the worst case ratio cannot be better (lower) than $2-2m/(n+m)$, it suffices to find an example that attains this bound. Consider n identical jobs with processing time equal to 1 and let n be a multiple of m. It is clear that under the Round Robin rule all n jobs are completed at time n/m, whereas under the nonpreemptive scheduling rule (which is also equivalent to SPT), m jobs are completed at time 1, m jobs at time 2, and so on. So for this example the ratio is n^2/m divided by

$$\frac{m}{2}\Big(\frac{n}{m}\Big(\frac{n}{m}+1\Big)\Big),$$

which equals $2 - 2m/(n+m)$.

It remains to be shown that the worst case ratio cannot be worse (larger) than $2-2m/(n+m)$. Assume the processing times of the jobs are $p_1 \geq p_2 \geq \cdots \geq p_n$. Let $R(\ell)$, $\ell = 1, \ldots, \lceil n/m \rceil$, denote the subset of jobs j that satisfy

$$(\ell - 1)m < j \leq \ell m.$$

So $R(1)$ contains jobs $1, \ldots, m$ (the longest jobs); $R(2)$ contains jobs $m+1, m+2, \ldots, 2m$, and so on. It can be shown that the schedule that minimizes the total completion time is SPT and that the total completion time under SPT is

$$\sum_{j=1}^{n} C_j(OPT) = \sum_{j=1}^{n} C_j(SPT) = \sum_{\ell=1}^{\lceil n/m \rceil} \sum_{j \in R(\ell)} \ell p_j.$$

Consider now the total completion time under RR. Note that C_j denotes the completion time of the jth longest job. It can be shown now that

$$C_j(RR) = C_{j+1}(RR) + (j/m)(p_j - p_{j+1})$$

for $j \geq m$, while

$$C_j(RR) = C_{m+1}(RR) + p_j - p_{m+1}$$

for $j < m$. Eliminating the recurrence yields for $j \geq m$

$$C_j(RR) = \frac{j}{m} p_j + \frac{1}{m} \sum_{k=j+1}^{n} p_k$$

and for $j < m$

$$C_j(RR) = p_j + \frac{1}{m} \sum_{k=m+1}^{n} p_k.$$

A simple calculation establishes that

$$\sum_{j=1}^{n} C_j(RR) = \sum_{j=1}^{m} p_j + \sum_{j=m+1}^{n} \frac{2j-1}{m} p_j.$$

The ratio $\sum C_j(RR) / \sum C_j(OPT)$ is maximized when all the jobs in the same subset have the same processing time. To see this, note that for OPT (SPT) the coefficient of p_j's contribution to the total completion time is determined solely by its subset index. On the other hand, for RR, the coefficient is smaller for the longer jobs within a specific group. Thus, reducing the value of each p_j to be equal to the smallest processing time of any job in its group can only increase the ratio. By a similar argument, it can be shown that the worst case ratio is achieved when n is a multiple of m.

Assume now that each subset contains exactly m jobs of the same length. Let q_ℓ denote the common processing time of any job in subset $R(\ell)$. Then, a simple calculation shows that

$$\sum_{j=1}^{n} C_j(OPT) = \sum_{\ell=1}^{n/m} m\ell q_\ell,$$

and

$$\sum_{j=1}^{n} C_j(RR) = \sum_{\ell=1}^{n/m} m(2\ell - 1)q_\ell.$$

Once again, the ratio is maximized when all the q_ℓ are equal, implying that the worst case ratio is exactly $2 - 2m/(n + m)$.

Since in online scheduling a competitive ratio is usually not expressed as a function of n (since the number of jobs is typically not known in advance), the competitive ratio has to hold for any value of n. It follows that the competitive ratio for RR is equal to 2. □

Actually, there are several other variants of the online scheduling paradigm. The variant considered in this section assumes that the decision-maker does not know the processing time of a job when it is released. The decision-maker only finds out what the processing time is when the job is completed. This form of online scheduling is at times referred to as *non-clairvoyant* online scheduling. In another variant of online scheduling, the processing time of a job becomes known to the decision-maker immediately upon the job's release. This variant is often referred to as *clairvoyant* online scheduling. However, in clairvoyant online scheduling the decision-maker still does not know how many jobs are going to be released and when the releases will occur.

An entirely different class of online algorithms are the so-called randomized online algorithms. A randomized algorithm allows the decision-maker to make random choices (for example, instead of assigning a job to the machine with the smallest load, the decision-maker may assign a job to a machine at random). If randomization is allowed, then it is of interest to know the expected objective value, where the expectation is taken over the random choices of the algorithm. A randomized algorithm is σ-competitive if for each instance this expectation is within a factor of σ of the optimal objective value.

5.7 Discussion

This chapter focuses primarily on parallel machine problems that either are polynomial time solvable or have certain properties that are of interest. This chapter does not address the more complicated parallel machine problems that are strongly NP-hard and have little structure.

A significant amount of research has been done on parallel machine scheduling problems that are strongly NP-hard. A variety of integer programming formulations have been developed for $Rm \mid\mid \sum w_j C_j$ and $Rm \mid\mid \sum w_j U_j$. These integer programs can be solved using a special form of branch-and-bound that is called branch-and-price and that is often referred to as column generation (see Appendix A). However, there are many other parallel machine problems that are more complicated and that have not yet been tackled with exact methods.

An example of such a very hard problem is $Qm \mid s_{ijk} \mid \sum w_j T_j$. This problem is extremely hard to solve to optimality. It is already hard to find an optimal solution for instances with, say, 5 machines and 30 jobs. However, this problem is of considerable interest to industry and many heuristics have been developed and experimented with. Part III of this book describes several heuristic methods that have been applied to this problem.

Online scheduling in a parallel machine environment has received a significant amount of attention during the last couple of years. Online scheduling is important for several reasons. In practice, it is often the case that a very limited amount of information is available when a decision must be made, see Example 1.1.4. From a theoretical point of view, online scheduling is of interest because it establishes a bridge between deterministic and stochastic scheduling. In stochastic scheduling decisions also have to be made with only a limited amount of information available. However, the stochastic scheduling paradigm is still quite different from the online paradigm. Nevertheless, the bounds obtained in online scheduling often give rise to bounds in stochastic scheduling.

Exercises (Computational)

5.1. Consider $P6 \mid\mid C_{\max}$ with 13 jobs.

jobs	1	2	3	4	5	6	7	8	9	10	11	12	13
p_j	6	6	6	7	7	8	8	9	9	10	10	11	11

(a) Compute the makespan under LPT.
(b) Find the optimal schedule.

5.2. Consider $P4 \mid prec \mid C_{\max}$ with 12 jobs.

jobs	1	2	3	4	5	6	7	8	9	10	11	12
p_j	10	10	10	12	11	10	12	12	10	10	10	10

The jobs are subject to the precedence constraints depicted in Figure 5.10.

(a) Apply the generalized version of the CP rule: every time a machine is freed select the job at the head of the string with the largest total amount of processing.
(b) Apply the generalized version of the LNS rule: every time a machine is freed select the job that precedes the largest total amount of processing.
(c) Is either one of these two schedules optimal?

5.3. Consider $P3 \mid brkdwn, M_j \mid C_{\max}$ with 8 jobs.

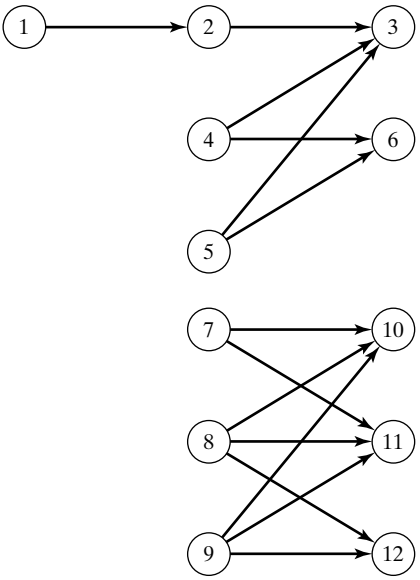


Fig. 5.10 Precedence constraints graph (Exercise 5.2)

jobs	1	2	3	4	5	6	7	8
p_j	10	10	7	7	7	7	7	7

Machines 1 and 2 are available continuously. Machine 3 is not available during the interval $[0, 1]$; after time 1 it is available throughout. The M_j sets are defined as follows:

$$M_1 = \{1, 3\}$$
$$M_2 = \{2, 3\}$$
$$M_3 = M_4 = M_5 = \{1\}$$
$$M_6 = M_7 = M_8 = \{2\}$$

(a) Apply the LPT rule, i.e., give always priority to the longest job that can be processed on the machine freed.

(b) Apply the LFJ rule, i.e., give always priority to the least flexible job while disregarding processing times.

(c) Compute the ratio $C_{\max}(LPT)/C_{\max}(LFJ)$.

5.4. Consider $P3 \mid prmp \mid \sum C_j$ with the additional constraint that the completion of job j has to be less than or equal to a given fixed deadline d_j. Preemptions may occur only at integer times $1, 2, 3, \ldots$

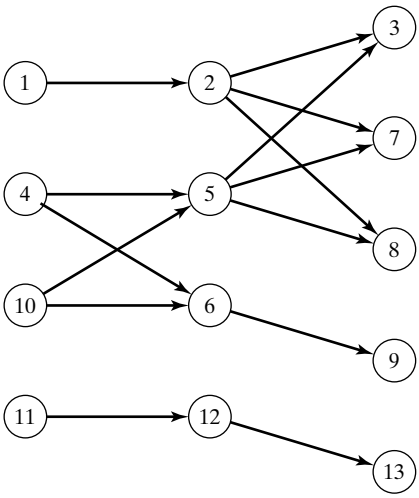


Fig. 5.11 Precedence constraints graph ($P\infty \mid prec \mid C_{\max}$) for Exercise 5.5

jobs	1	2	3	4	5	6	7	8	9	10	11
p_j	2	3	3	5	8	8	8	9	12	14	16
d_j	∞	∞	∞	∞	∞	∞	11	12	13	28	29

Find the optimal schedule and compute the total completion time.

5.5. Consider $P\infty \mid prec \mid C_{\max}$

jobs	1	2	3	4	5	6	7	8	9	10	11	12	13
p_j	5	11	9	8	7	3	8	6	9	2	5	2	9

The precedence constraints are depicted in Figure 5.11. Determine the optimal makespan and which jobs are critical and which jobs are slack.

5.6. Consider $P5 \mid\mid \sum h(C_j)$ with 11 jobs.

jobs	1	2	3	4	5	6	7	8	9	10	11
p_j	5	5	5	6	6	7	7	8	8	9	9

The function $h(C_j)$ is defined as follows.

$$h(C_j) = \begin{cases} 0 & \text{if } C_j \leq 15 \\ C_j - 15 & \text{if } C_j > 15 \end{cases}$$

(a) Compute the value of the objective under SPT.

(b) Compute the value of the objective under the optimal schedule.

5.7. Consider again $P5 \mid\mid \sum h(C_j)$ with the 11 jobs of the previous exercise. The function $h(C_j)$ is now defined as follows:

$$h(C_j) = \begin{cases} C_j \text{ if } C_j \leq 15 \\ 15 \text{ if } C_j > 15. \end{cases}$$

(a) Compute the value of the objective function under SPT.

(b) Compute the value of the objective under the optimal schedule.

5.8. Consider $Q2 \mid prmp \mid C_{\max}$ with the jobs

jobs	1	2	3	4
p_j	36	24	16	12

and machine speeds $v_1 = 2$ and $v_2 = 1$.

(a) Find the makespan under LRPT when preemptions can only be made at the time points 0, 4, 8, 12, and so on.

(b) Find the makespan under LRPT when preemptions can only be made at the time points 0, 2, 4, 6, 8, 10, 12, and so on.

(c) Find the makespan under LRPT when preemptions can be made at *any* time.

(d) Compare the makespans under (a), (b) and (c).

5.9. Consider the following example of $P3 \mid prmp, brkdwn \mid \sum C_j$ with 6 jobs. Three jobs have a processing time of 1, while the remaining three have a processing time of 2. There are three machines, but two machines are not available from time 2 onwards. Determine the optimal schedule. Show that SRPT is not optimal.

5.10. Consider the following instance of $P2 \mid prmp \mid L_{\max}$ with 4 jobs. Preemptions are allowed at integer points in time. Find an optimal schedule.

jobs	1	2	3	4
d_j	5	6	9	10
p_j	4	5	7	9

Exercises (Theory)

5.11. Consider $Pm \,||\, C_{\max}$.

(a) Give an example showing that LPT is not necessarily optimal when the number of jobs is less than or equal to twice the number of machines ($n \leq 2m$).

(b) Show that if an optimal schedule results in at most two jobs on any machine, then LPT is optimal.

5.12. Consider $Pm \,||\, C_{\max}$. Describe the processing times of the instance that attains the worst case bound in Theorem 5.1.1 (as a function of m). (*Hint:* see Exercise 5.1.)

5.13. Show that the CP rule is optimal for $Pm \mid outtree, p_j = 1 \mid C_{\max}$.

5.14. Complete the proof of Theorem 5.1.5. That is, show that the CP rule applied to $Pm \mid intree, p_j = 1 \mid C_{\max}$ results in a makespan that is equal to $l_{\max} + c$.

5.15. Consider $Pm \mid r_j, prmp \mid C_{\max}$. Formulate the optimal policy and prove its optimality.

5.16. Consider $Pm \mid prmp, brkdwn \mid C_{\max}$ with the number of machines available a function of time, i.e., $m(t)$. Show that for any function $m(t)$ LRPT minimizes the makespan.

5.17. Consider $Pm \mid brkdwn \mid \sum C_j$ with the number of machines available a function of time, i.e., $m(t)$. Show that if $m(t)$ is increasing the nonpreemptive SPT rule is optimal.

5.18. Consider $Pm \mid prmp, brkdwn \mid \sum C_j$ with the number of machines available a function of time, i.e., $m(t)$. Show that the preemptive SRPT rule is optimal if $m(t) \geq m(s) - 1$ for all $s < t$.

5.19. Consider $Pm \mid prmp \mid \sum C_j$ with the added restriction that all jobs *must* be finished by some fixed deadline $\bar{d}$, where

$$\bar{d} \geq \max\Big(\frac{\sum p_j}{m}, p_1, \ldots, p_n\Big).$$

Find the rule that minimizes the total completion time and prove its optimality.

5.20. Consider $Pm \,||\, \sum w_j C_j$. Show that in the worst case example of the WSPT rule w_j has to be approximately equal to p_j, for each j.

5.21. Give a characterization of the class of all schedules that are optimal for $Pm \,||\, \sum C_j$. Determine the number of schedules in this class as a function of n and m.

5.22. Consider $P2 \mid\mid \sum C_j$. Develop a heuristic for minimizing the makespan subject to total completion time optimality. (*Hint:* Say a job is of *Rank* j if $j-1$ jobs follow the job on its machine. With two machines in parallel there are two jobs in each rank. Consider the difference in the processing times of the two jobs in the same rank. Base the heuristic on these differences.)

5.23. Consider $Pm \mid M_j \mid \gamma$. The sets M_j are given. Let J_i denote the set of jobs that are allowed to be processed on machine i. Show, through a counterexample, that the sets M_j being nested does not necessarily imply that sets J_i are nested. Give sufficiency conditions on the set structures under which the M_j sets as well as the J_i sets are nested.

5.24. Show, through a counterexample, that the CP rule is not necessarily optimal for $Pm \mid intree, p_j = 1 \mid \sum C_j$.

5.25. Consider $Pm \mid r_j, prmp \mid L_{\max}$. Show through a counterexample that the preemptive EDD rule does *not* necessarily yield an optimal schedule.

5.26. Consider $Pm \mid intree, prmp \mid C_{\max}$ with the processing time of each job at level k equal to p_k. Show that a preemptive version of the generalized CP rule minimizes the makespan.

5.27. Consider $Q\infty \mid prec, prmp \mid C_{\max}$. There are an unlimited number of machines that operate at the same speed. There is *one* machine that is faster. Give an algorithm that minimizes the makespan and prove its optimality.

5.28. Consider an online version of $Pm \mid r_j, prec \mid C_{\max}$. An online algorithm for this problem can be described as follows. The jobs are again presented in a list; whenever a machine is freed, the job that ranks highest among the remaining jobs which are ready for processing is assigned to that machine (i.e., it must be a job that already has been released and of which all predecessors already have been completed). Show that the bound presented in Theorem 5.6.1 applies to this more general problem as well.

Comments and References

The worst case analysis of the LPT rule for $Pm \mid\mid C_{\max}$ is from the classic paper by Graham (1969). This paper gives one of the first examples of worst case analyses of heuristics (see also Graham (1966)). It also provides a worst case analysis of an arbitrary list schedule for $Pm \mid\mid C_{\max}$. A more sophisticated heuristic for $Pm \mid\mid C_{\max}$, with a tighter worst case bound, is the so-called MULTIFIT heuristic, see Coffman, Garey and Johnson (1978) and Friesen (1984a). Lee and Massey (1988) analyze a heuristic that is based on LPT as well as on MULTIFIT. Hwang, Lee and Chang (2005) perform a worst case analysis of the LPT rule for $Pm \mid brkdwn \mid C_{\max}$. For results on heuristics for the more general $Qm \mid\mid C_{\max}$, see Friesen and Langston (1983), Friesen (1984b), and Dobson

(1984). Davis and Jaffe (1981) present an algorithm for $Rm \mid\mid C_{\max}$. The CPM and PERT procedures have been covered in many papers and textbooks, see for example, French (1982). The CP result in Theorem 5.1.5 is due to Hu (1961). See Lenstra and Rinnooy Kan (1978) with regard to $Pm \mid p_j = 1, prec \mid C_{\max}$, Du and Leung (1989) with regard to $P2 \mid tree \mid C_{\max}$ and Du, Leung and Young (1991) with regard to $P2 \mid chains \mid C_{\max}$. Chen and Liu (1975) and Kunde (1976) analyze the worst case behavior of the CP rule for $Pm \mid p_j = 1, prec \mid C_{\max}$. Lin and Li (2004) and Li (2006) do a complexity analysis of $Pm \mid p_j = 1, M_j \mid C_{\max}$ and $Qm \mid p_j = p, M_j \mid C_{\max}$. Apparently, no worst case analysis has been done for the LFJ rule.

For $Pm \mid prmp \mid C_{\max}$, see McNaughton (1959). For $Qm \mid prmp \mid C_{\max}$, see Horvath, Lam and Sethi (1977), Gonzalez and Sahni (1978a) and McCormick and Pinedo (1995).

Conway, Maxwell and Miller (1967) discuss the SPT rule for $Pm \mid\mid \sum C_j$; they also give a characterization of the class of optimal schedules. For a discussion of $Qm \mid\mid \sum C_j$, see Horowitz and Sahni (1976). The worst case bound for the WSPT rule for $Pm \mid\mid \sum w_j C_j$ is from Kawaguchi and Kyan (1986). Elmaghraby and Park (1974) and Sarin, Ahn and Bishop (1988) present branch-and-bound algorithms for this problem. Eck and Pinedo (1993) present a heuristic for minimizing the makespan and the total completion time simultaneously. The optimality of the CP rule for $Pm \mid p_j = 1, outtree \mid \sum C_j$ is due to Hu (1961). For complexity results with regard to $Pm \mid prec \mid \sum C_j$, see Sethi (1977) and Du, Leung and Young (1990).

For an analysis of the $Qm \mid prmp \mid \sum C_j$ problem, see Lawler and Labetoulle (1978), Gonzalez and Sahni (1978a), McCormick and Pinedo (1995), Leung and Pinedo (2003) and Gonzalez, Leung and Pinedo (2006).

A significant amount of work has been done on $Qm \mid r_j, p_j = p, prmp \mid \gamma$; see Garey, Johnson, Simons and Tarjan (1981), Federgruen and Groenevelt (1986), Lawler and Martel (1989), Martel (1982) and Simons (1983).

For results with regard to $Qm \mid prmp \mid L_{\max}$, see Bruno and Gonzalez (1976) and Labetoulle, Lawler, Lenstra and Rinnooy Kan (1984). For other due date related results, see Sahni and Cho (1979b).

Chen and Powell (1999) and Van den Akker, Hoogeveen and Van de Velde (1999) applied branch-and-bound methods (including column generation) to $Rm \mid\mid \sum w_j C_j$ and $Rm \mid\mid \sum w_j U_j$.

The worst case analysis of an arbitrary list schedule for $Pm \mid\mid C_{\max}$ is regarded as one of the basic results in online scheduling. Theorem 5.6.1 is due to Graham (1966). The analysis of the Round Robin rule and the total completion time objective is due to Motwani, Phillips and Torng (1994). Research in online scheduling has focused on other parallel machine scheduling problems as well; see, for example, Shmoys, Wein and Williamson (1995). For an overview of online scheduling on parallel machines with the makespan objective, see Fleischer and Wahl (2000). For comprehensive overviews of online scheduling, see Sgall (1998) and Pruhs, Sgall and Torng (2004).

Chapter 6

Flow Shops and Flexible Flow Shops (Deterministic)

In many manufacturing and assembly facilities each job has to undergo a series of operations. Often, these operations have to be done on all jobs in the same order implying that the jobs have to follow the same route. The machines are then assumed to be set up in series and the environment is referred to as a flow shop.

The storage or buffer capacities in between successive machines may sometimes be, for all practical purposes, virtually unlimited. This is often the case when the products that are being processed are physically small (e.g., printed circuit boards, integrated circuits), making it relatively easy to store large quantities between machines. When the products are physically large (e.g., television sets, copiers), then the buffer space in between two successive machines may have a limited capacity, causing *blocking*. Blocking occurs when the buffer is full and the upstream machine is not allowed to release a job into the buffer after completing its processing. If this is the case, then the job has to remain at the upstream machine, preventing a job in the queue at that machine from beginning its processing.

A somewhat more general machine environment consists of a number of stages in series with a number of machines in parallel at each stage. A job has to be processed at each stage only on one of the machines. This machine environment is often referred to as a flexible flow shop, compound flow shop, multi-processor flow shop, or hybrid flow shop.

Most of the material in this chapter concerns the makespan objective. The makespan objective is of considerable practical interest as its minimization is

M.L. Pinedo, *Scheduling*, DOI: 10.1007/978-0-387-78935-4_6,

to a certain extent equivalent to the maximization of the utilization of the machines. The models, however, tend to be of such complexity that makespan results are already relatively hard to obtain. Total completion time and due date related objectives tend to be even harder.

6.1 Flow Shops with Unlimited Intermediate Storage

When searching for an optimal schedule for $Fm \mid\mid C_{\max}$ the question arises whether it suffices merely to determine a permutation in which the jobs traverse the entire system. Physically it may be possible for one job to "pass" another while waiting in queue for a machine that is busy. The machines may not operate according to the *First Come First Served* principle and the sequence in which the jobs go through the machines may change from one machine to another. Changing the sequence of the jobs waiting in a queue between two machines may at times result in a smaller makespan. However, it can be shown that there always exists an optimal schedule without job sequence changes between the first two machines and between the last two machines (see Exercise 6.11). This implies that there are optimal schedules for $F2 \mid\mid C_{\max}$ and $F3 \mid\mid C_{\max}$ that do not require sequence changes between machines. One can find examples of flow shops with four machines in which the optimal schedule does require a job sequence change in between the second and the third machine.

Finding an optimal schedule when sequence changes are allowed is significantly harder than finding an optimal schedule when sequence changes are not allowed. Flow shops that do not allow sequence changes between machines are called *permutation* flow shops. In these flow shops the same sequence, or permutation, of jobs is maintained throughout. The results in this chapter are mostly limited to permutation flow shops.

Given a permutation schedule $j_1, \ldots, j_n$ for an m machine flow shop, the completion time of job j_k at machine i can be computed easily through a set of recursive equations:

$$C_{i,j_1} = \sum_{l=1}^{i} p_{l,j_1} \qquad i = 1, \ldots, m$$

$$C_{1,j_k} = \sum_{l=1}^{k} p_{1,j_l} \qquad k = 1, \ldots, n$$

$$C_{i,j_k} = \max(C_{i-1,j_k}, C_{i,j_{k-1}}) + p_{i,j_k} \qquad i = 2, \ldots, m; \quad k = 2, \ldots, n$$

The value of the makespan under a given permutation schedule can also be computed by determining the *critical path* in a directed graph that corresponds to the schedule. For a given sequence $j_1, \ldots, j_n$ this directed graph is constructed as follows: for each operation, say the processing of job j_k on machine i, there is a node (i, j_k) with a weight that is equal to the processing time of job j_k on machine i. Node (i, j_k), $i = 1, \ldots, m-1$, and $k = 1, \ldots, n-1$, has arcs

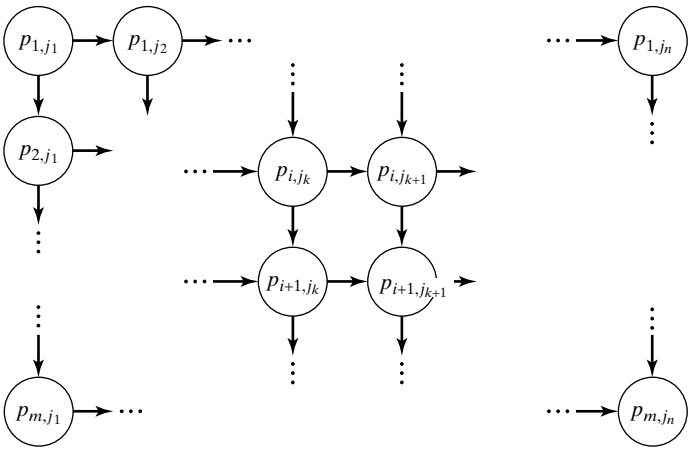


Fig. 6.1 Directed Graph for the Computation of the Makespan in $Fm \mid prmu \mid C_{\max}$ under sequence $j_1, \ldots, j_n$

going out to nodes $(i+1, j_k)$ and (i, j_{k+1}). Nodes corresponding to machine m have only one outgoing arc, as do nodes corresponding to job j_n. Node (m, j_n) has no outgoing arcs (see Figure 6.1). The total weight of the maximum weight path from node $(1, j_1)$ to node (m, j_n) corresponds to the makespan under the permutation schedule $j_1, \ldots, j_n$.

Example 6.1.1 (Graph Representation of Flow Shop)

Consider 5 jobs on 4 machines with the processing times presented in the table below.

jobs	j_1	j_2	j_3	j_4	j_5
p_{1,j_k}	5	5	3	6	3
p_{2,j_k}	4	4	2	4	4
p_{3,j_k}	4	4	3	4	1
p_{4,j_k}	3	6	3	2	5

The corresponding graph and Gantt chart are depicted in Figure 6.2. From the directed graph it follows that the makespan is 34. This makespan is determined by two critical paths. ||

An interesting result can be obtained by comparing two m machine permutation flow shops with n jobs. Let $p_{ij}^{(1)}$ and $p_{ij}^{(2)}$ denote the processing time of job j on machine i in the first and second flow shop, respectively. Assume

$$p_{ij}^{(1)} = p_{m+1-i,j}^{(2)}.$$

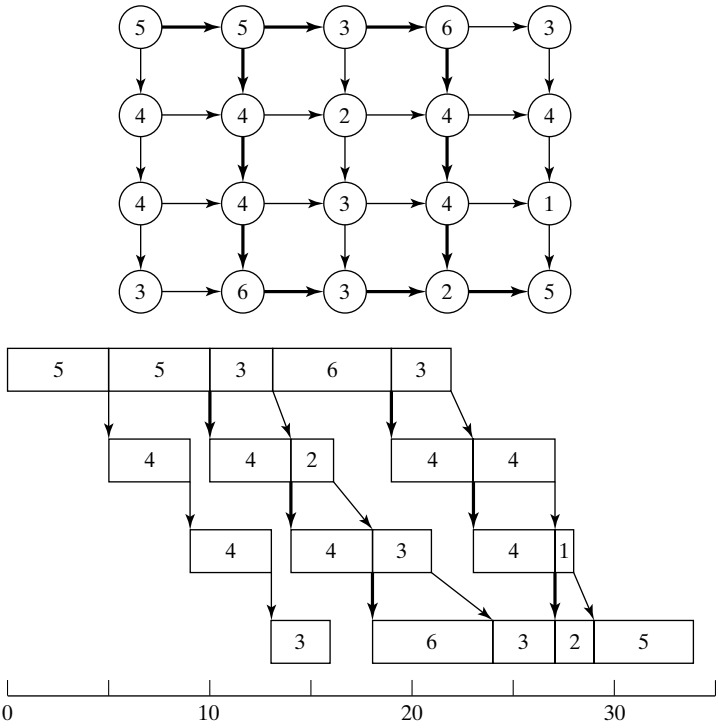


Fig. 6.2 Directed graph, critical paths and Gantt chart (the numerical entries represent the processing times of the jobs and not the job indexes)

This basically implies that the first machine in the second flow shop is identical to the last machine in the first flow shop; the second machine in the second flow shop is identical to the machine immediately before the last in the first flow shop, and so on. The following lemma applies to these two flow shops.

Lemma 6.1.2. *Sequencing the jobs according to permutation $j_1, \ldots, j_n$ in the first flow shop results in the same makespan as sequencing the jobs according to permutation $j_n, \ldots, j_1$ in the second flow shop.*

Proof. If the first flow shop under sequence $j_1, \ldots, j_n$ corresponds to the diagram in Figure 6.1, then the second flow shop under sequence $j_n, \ldots, j_1$ corresponds to the same diagram with all arcs reversed. The weight of the maximum weight path from one corner node to the other corner node does not change if all arcs are reversed. □

Lemma 6.1.2 states the following *reversibility* result: the makespan does not change if the jobs traverse the flow shop in the opposite direction in reverse order.

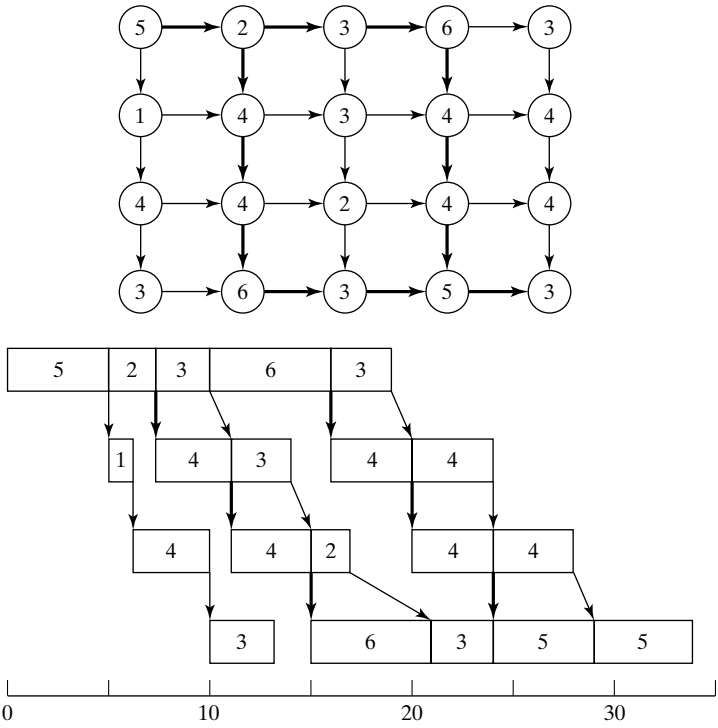


Fig. 6.3 Directed graph, critical paths and Gantt chart (the numerical entries represent the processing times of the jobs and not the job indexes)

Example 6.1.3 (Graph Representations and Reversibility)
Consider the instance of Example 6.1.1. The dual of this instance is given in the table below.

jobs	j_1	j_2	j_3	j_4	j_5
p_{1,j_k}	5	2	3	6	3
p_{2,j_k}	1	4	3	4	4
p_{3,j_k}	4	4	2	4	4
p_{4,j_k}	3	6	3	5	5

The corresponding directed graph, its critical paths and the Gantt charts are depicted in Figure 6.3. It is clear that the critical paths are determined by the same set of processing times and that the makespan, therefore, is 34 as well. ||

Consider now the $F2 \mid\mid C_{\max}$ problem: a flow shop with two machines in series with unlimited storage in between the two machines. There are n jobs and the processing time of job j on machine 1 is p_{1j} and its processing time on machine 2 is p_{2j}. This was one of the first problems to be analyzed in the early days of Operations Research and led to a classical paper in scheduling theory by S.M. Johnson. The rule that minimizes the makespan is commonly referred to as Johnson's rule.

An optimal sequence can be described as follows. Partition the jobs into two sets with Set I containing all jobs with $p_{1j} < p_{2j}$ and Set II all jobs with $p_{1j} > p_{2j}$. The jobs with $p_{1j} = p_{2j}$ may be put in either set. The jobs in Set I go first and they go in increasing order of p_{1j} (SPT); the jobs in Set II follow in decreasing order of p_{2j} (LPT). Ties may be broken arbitrarily. In what follows such a schedule is referred to as an *SPT(1)-LPT(2)* schedule. Of course, multiple schedules may be generated this way.

Theorem 6.1.4. *Any SPT(1)-LPT(2) schedule is optimal for* $F2 \mid\mid C_{\max}$.

Proof. The proof is by contradiction. Suppose another type of schedule is optimal. In this optimal schedule there must be a pair of adjacent jobs, say job j followed by job k, that satisfies one of the following three conditions:

(i) job j belongs to Set II and job k to Set I;
(ii) jobs j and k belong to Set I and $p_{1j} > p_{1k}$;
(iii) jobs j and k belong to Set II and $p_{2j} < p_{2k}$.

It suffices to show that under any of these three conditions the makespan is reduced after a pairwise interchange of jobs j and k. Assume that in the original schedule job l precedes job j and job m follows job k. Let C_{ij} denote the completion of job j on machine i under the original schedule and let C'_{ij} denote the completion time of job j on machine i after the pairwise interchange. Interchanging jobs j and k clearly does not affect the starting time of job m on machine 1, as its starting time on machine 1 equals $C_{1l} + p_{1j} + p_{1k}$. However, it is of interest to know at what time machine 2 becomes available for job m. Under the original schedule this is the completion time of job k on machine 2, i.e., C_{2k}, and after the interchange this is the completion time of job j on machine 2, i.e., C'_{2j}. It suffices to show that $C'_{2j} \leq C_{2k}$ under any one of the three conditions described above.

The completion time of job k on machine 2 under the original schedule is

$$\begin{aligned} C_{2k} &= \max\Big(\max\left(C_{2l},\ \ C_{1l} + p_{1j}\right) + p_{2j},\ \ C_{1l} + p_{1j} + p_{1k}\Big) + p_{2k} \\ &= \max\Big(C_{2l} + p_{2k} + p_{2j},\ \ C_{1l} + p_{1j} + p_{2j} + p_{2k},\ \ C_{1l} + p_{1j} + p_{1k} + p_{2k}\Big), \end{aligned}$$

whereas the completion time of job j on machine 2 after the pairwise interchange is

$$C'_{2j} = \max\Big(C_{2l} + p_{2k} + p_{2j},\ \ C_{1l} + p_{1k} + p_{2k} + p_{2j},\ \ C_{1l} + p_{1k} + p_{1j} + p_{2j}\Big).$$

Under condition (i) $p_{1j} > p_{2j}$ and $p_{1k} < p_{2k}$. It is clear that the first terms within the max expressions of C_{2k} and C'_{2j} are identical. The second term in the last expression is smaller than the third in the first expression and the third term in the last expression is smaller than the second in the first expression. So, under condition (i) $C'_{2j} \leq C_{2k}$.

Under condition (ii) $p_{1j} < p_{2j}$, $p_{1k} < p_{2k}$ and $p_{1j} > p_{1k}$. Now the second as well as the third term in the last expression are smaller than the second term in the first expression. So, under condition (ii) $C'_{2j} \leq C_{2k}$ as well.

Condition (iii) can be shown in a similar way as the second condition. Actually, condition (iii) follows immediately from the reversibility property of flow shops. □

These SPT(1)-LPT(2) schedules are by no means the only schedules that are optimal for $F2 \mid\mid C_{\max}$. The class of optimal schedules appears to be hard to characterize and data dependent.

Example 6.1.5 (Multiple Optimal Schedules)

Consider a set of jobs with one job that has a very small processing time on machine 1 and a very large processing time on machine 2, say K, with $K \geq \sum_{j=1}^{n} p_{1j}$. It is clear that under the optimal sequence this job should go first in the schedule. However, the order of the remaining jobs does not affect the makespan. ||

Unfortunately, the SPT(1)-LPT(2) schedule structure cannot be generalized to characterize optimal schedules for flow shops with more than two machines. However, minimizing the makespan in a permutation flow shop with an arbitrary number of machines, i.e., $Fm \mid prmu \mid C_{\max}$, can be formulated as a Mixed Integer Program (MIP).

In order to formulate the problem as a MIP a number of variables have to be defined: The decision variable x_{jk} equals 1 if job j is the kth job in the sequence and 0 otherwise. The auxiliary variable I_{ik} denotes the idle time on machine i between the processing of the jobs in the kth position and $(k+1)$th position and the auxiliary variable W_{ik} denotes the waiting time of the job in the kth position in between machines i and $i+1$. Of course, there exists a strong relationship between the variables W_{ik} and the variables I_{ik}. For example, if $I_{ik} > 0$, then $W_{i-1,k+1}$ has to be zero. Formally, this relationship can be established by considering the difference between the time the job in the $(k+1)$th position starts on machine $i+1$ and the time the job in the kth position completes its processing on machine i. If Δ_{ik} denotes this difference and if $p_{i(k)}$ denotes the processing time on machine i of the job in the kth position in the sequence, then (see Figure 6.4)

$$\Delta_{ik} = I_{ik} + p_{i(k+1)} + W_{i,k+1} = W_{ik} + p_{i+1(k)} + I_{i+1,k}.$$

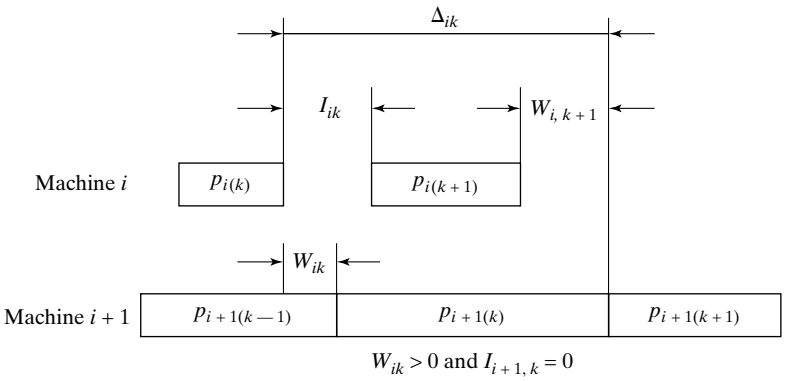


Fig. 6.4 Constraints in the integer programming formulation

Note that minimizing the makespan is equivalent to minimizing the total idle time on the last machine, machine m. This idle time is equal to

$$\sum_{i=1}^{m-1} p_{i(1)} + \sum_{j=1}^{n-1} I_{mj},$$

which is the idle time that must occur before the job in the first position reaches the last machine and the sum of the idle times between the jobs on the last machine. Using the identity

$$p_{i(k)} = \sum_{j=1}^{n} x_{jk} p_{ij},$$

the MIP can now be formulated as follows.

$$\min \Big(\sum_{i=1}^{m-1} \sum_{j=1}^{n} x_{j1} p_{ij} + \sum_{j=1}^{n-1} I_{mj} \Big),$$

subject to

$$\sum_{j=1}^{n} x_{jk} = 1 \qquad k = 1, \ldots, n,$$

$$\sum_{k=1}^{n} x_{jk} = 1 \qquad j = 1, \ldots, n,$$

$$I_{ik} + \sum_{j=1}^{n} x_{j,k+1} p_{ij} + W_{i,k+1}$$

$$-W_{ik} - \sum_{j=1}^{n} x_{jk} p_{i+1,j} - I_{i+1,k} = 0 \qquad k = 1, \ldots, n-1; \;\; i = 1, \ldots, m-1,$$

$$W_{i1} = 0 \qquad i = 1, \ldots, m-1,$$

$$I_{1k} = 0 \qquad k = 1, \ldots, n-1.$$

The first set of constraints specifies that exactly one job has to be assigned to position k for any k. The second set of constraints specifies that job j has to be assigned to exactly one position. The third set of constraints relate the decision variables x_{jk} to the physical constraints. These physical constraints enforce the necessary relationships between the idle time variables and the waiting time variables. Thus, the problem of minimizing the makespan in an m machine permutation flow shop is formulated as a MIP. The only integer variables are the binary $(0-1)$ decision variables x_{jk}. The idle time and waiting time variables are nonnegative continuous variables.

Example 6.1.6 (Mixed Integer Programming Formulation)

Consider again the instance in Example 6.1.1. Because the sequence is now not given, the subscript j_k in the table of Example 6.1.1 is replaced by the subscript j and the headings j_1, j_2, j_3, j_4 and j_5 are replaced by 1, 2, 3, 4, and 5, respectively.

jobs	1	2	3	4	5
$p_{1,j}$	5	5	3	6	3
$p_{2,j}$	4	4	2	4	4
$p_{3,j}$	4	4	3	4	1
$p_{4,j}$	3	6	3	2	5

With these data the objective of the MIP is

$$5x_{11} + 5x_{21} + 3x_{31} + 6x_{41} + 3x_{51} + 4x_{11} + 4x_{21} + 2x_{31} + 4x_{41} + 4x_{51} +$$

$$+4x_{11} + 4x_{21} + 3x_{31} + 4x_{41} + x_{51} + I_{41} + I_{42} + I_{43} + I_{44} =$$

$$13x_{11} + 13x_{21} + 8x_{31} + 14x_{41} + 8x_{51} + I_{41} + I_{42} + I_{43} + I_{44}$$

The first and second set of constraints of the program contain 5 constraints each. The third set contains $(5-1)(4-1) = 12$ constraints. For example, the constraint corresponding to $k = 2$ and $i = 3$ is

$$I_{32} + 4x_{13} + 4x_{23} + 3x_{33} + 4x_{43} + x_{53} + W_{33}$$

$$-W_{32} - 3x_{12} - 6x_{22} - 3x_{32} - 2x_{42} - 5x_{52} - I_{42} = 0. \qquad ||$$

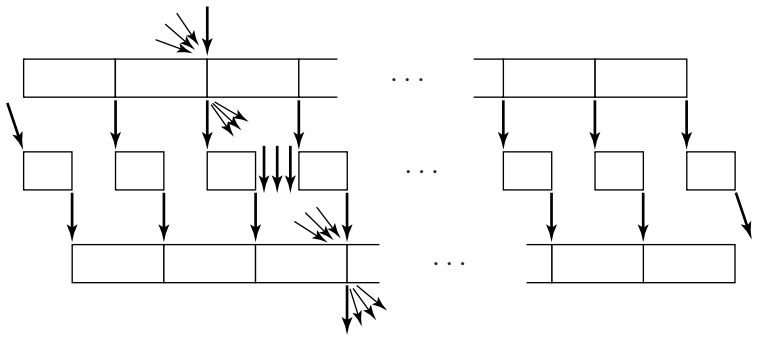


Fig. 6.5 3-PARTITION reduces to $F3 \mid\mid C_{\max}$

The fact that the problem can be formulated as a MIP does not immediately imply that the problem is NP-hard. It could be that the MIP has a special structure that allows for a polynomial time algorithm (see, for example, the integer programming formulation for $Rm \mid\mid \sum C_j$). In this case, however, it turns out that the problem *is* hard.

Theorem 6.1.7. *$F3 \mid\mid C_{\max}$ is strongly NP-hard.*

Proof. By reduction from 3-PARTITION. Given integers $a_1, \ldots, a_{3t}, b$, under the usual assumptions, let the number of jobs n equal $4t+1$ and let

$$\begin{array}{llll} p_{10}=0, & p_{20}=b, & p_{30}=2b, & \\ p_{1j}=2b, & p_{2j}=b, & p_{3j}=2b, & j=1,\ldots,t-1, \\ p_{1t}=2b, & p_{2t}=b, & p_{3t}=0, & \\ p_{1,t+j}=0, & p_{2,t+j}=a_j, & p_{3,t+j}=0, & j=1,\ldots,3t. \end{array}$$

Let $z=(2t+1)b$. A makespan of value z can be obtained if the first $t+1$ jobs are scheduled according to sequence $0,1,\ldots,t$. These $t+1$ jobs then form a framework, leaving t gaps on machine 2. Jobs $t+1,\ldots,t+3t$ have to be partitioned into t sets of three jobs each and these t sets have to be scheduled in between the first $t+1$ jobs. A makespan of value z can be obtained if and only if 3-PARTITION has a solution (see Figure 6.5). □

This complexity proof applies to permutation flow shops as well as to flow shops that allow sequence changes midstream (as said before, for three machine flow shops it is known that a permutation schedule is optimal in the larger class of schedules).

Even though $Fm \mid prmu \mid C_{\max}$ is strongly NP-hard it is of interest to study special cases that have nice structural properties. A number of special cases are important.

One important special case of $Fm \mid prmu \mid C_{\max}$ is the so-called *proportionate* permutation flow shop. In this flow shop the processing times of job j on each of the m machines are equal to p_j, i.e., $p_{1j} = p_{2j} = \cdots = p_{mj} = p_j$. Minimizing the makespan in a proportionate permutation flow shop is denoted by $Fm \mid prmu, p_{ij} = p_j \mid C_{\max}$. For a proportionate flow shop an *SPT-LPT* sequence can be defined as follows. The jobs are partitioned into two sets; the jobs in one subset go first according to SPT and the remaining jobs follow according to LPT. So a sequence $j_1, \ldots, j_n$ is SPT-LPT if and only if there is a job j_k such that

$$p_{j_1} \leq p_{j_2} \leq \cdots \leq p_{j_k}$$

and

$$p_{j_k} \geq p_{j_{k+1}} \geq \cdots \geq p_{j_n}.$$

From Theorem 6.1.4 it follows that when $m = 2$ any SPT-LPT sequence must be optimal. As might be expected, these are not the only sequences that are optimal. This flow shop has a very special structure.

Theorem 6.1.8. *For $Fm \mid prmu, p_{ij} = p_j \mid C_{\max}$ the makespan equals*

$$C_{\max} = \sum_{j=1}^{n} p_j + (m-1)\max(p_1, \ldots, p_n)$$

and is independent of the schedule.

Proof. The proof is left as an exercise. □

It can be shown that permutation schedules are also optimal in the larger class of schedules that allow jobs to pass one another while waiting for a machine, i.e., $Fm \mid p_{ij} = p_j \mid C_{\max}$ (see Exercise 6.17).

The result stated in the last theorem indicates that the proportionate flow shop is in one aspect similar to the single machine: the makespan does not depend on the sequence. Actually there are many more similarities between the proportionate flow shop and the single machine. The following results illustrate this fact.

(i) The SPT rule is optimal for $1 \mid\mid \sum C_j$ as well as for $Fm \mid prmu, p_{ij} = p_j \mid \sum C_j$.

(ii) The algorithm that results in an optimal schedule for $1 \mid\mid \sum U_j$ also results in an optimal schedule for $Fm \mid prmu, p_{ij} = p_j \mid \sum U_j$.

(iii) The algorithm that results in an optimal schedule for $1 \mid\mid h_{\max}$ also results in an optimal schedule for $Fm \mid prmu, p_{ij} = p_j \mid h_{\max}$.

(iv) The pseudo-polynomial dynamic programming algorithm for $1 \mid\mid \sum T_j$ can also be applied to $Fm \mid prmu, p_{ij} = p_j \mid \sum T_j$.

(v) The elimination criteria that hold for $1 \mid\mid \sum w_j T_j$ also hold for $Fm \mid prmu, p_{ij} = p_j \mid \sum w_j T_j$.

Not all results that hold for the single machine hold for the proportionate flow shop. For example, WSPT does *not* necessarily minimize the total weighted completion time in proportionate flow shops. Counterexamples can be found easily. However, $Fm \mid p_{ij} = p_j \mid \sum w_j C_j$ can still be solved in polynomial time.

The proportionate permutation flow shop model can be generalized to include machines with different speeds. If the speed of machine i is v_i, then the time job j spends on machine i is $p_{ij} = p_j/v_i$. The machine with the smallest v_i is called the *bottleneck* machine. The makespan is now no longer schedule independent.

Theorem 6.1.9. *If in a proportionate permutation flow shop with different speeds the first (last) machine is the bottleneck, then LPT (SPT) minimizes the makespan.*

Proof. From the reversibility property it immediately follows that it suffices to prove only one of the two results stated in the theorem. Only the case where the last machine is the bottleneck is shown here.

Consider first the special subcase with

$$v_m \leq v_1 \leq \min\,(v_2, \ldots, v_{m-1});$$

that is, the last machine is the bottleneck and the first machine requires the second longest processing times for each one of the n jobs. It is easy to see that in such a flow shop the critical path only turns to the right at machine m and therefore turns down only once at some job j_k in the sequence $j_1, \ldots, j_n$. So the critical path starts out on machine 1 going to the right, turns down at job j_k and goes all the way down to machine m before turning to the right again. That SPT is optimal can be shown through a standard adjacent pairwise interchange argument. Consider a schedule that is not SPT. There are two adjacent jobs of which the first one is longer than the second. Interchanging these two jobs affects the makespan if and only if one of the two jobs is the job through which the critical path goes from machine 1 to m. It can be shown that an interchange then reduces the makespan and that SPT minimizes the makespan.

In order to complete the proof for the general case, call machine h an *intermediate bottleneck* if $v_h < \min\,(v_1, \ldots, v_{h-1})$. There may be a number of intermediate bottlenecks in the proportionate flow shop. The arguments presented above for the case with the only intermediate bottleneck being machine 1 extend to the general case with multiple intermediate bottlenecks. The critical path now only turns right at intermediate bottleneck machines. This structure can be exploited again with an adjacent pairwise interchange argument showing that SPT minimizes the makespan. □

As $Fm \mid prmu \mid C_{\max}$ is one of the more basic scheduling problems, it has attracted a great deal of attention over the years. Many heuristics have been developed for dealing with this problem. One of the first heuristics developed for this problem is the *Slope* heuristic. According to this heuristic a slope index

is computed for each job. The slope index A_j for job j is defined as

$$A_j = -\sum_{i=1}^{m} \Big(m - (2i-1)\Big) p_{ij}.$$

The jobs are then sequenced in decreasing order of the slope index. The reasoning behind this heuristic is simple. From Theorem 6.1.4 it is already clear that jobs with small processing times on the first machine and large processing times on the second machine should be positioned more towards the beginning of the schedule, while jobs with large processing times on the first machine and small processing times on the second machine should be positioned more towards the end of the schedule. The slope index is large if the processing times on the downstream machines are large relative to the processing times on the upstream machines; the slope index is small if the processing times on the downstream machines are relatively small in comparison with the processing times on the upstream machines.

Example 6.1.10 (Application of the Slope Heuristic)
Consider again the instance in Examples 6.1.1 and 6.1.6. Replace the j_k by j and the $j_1, \ldots, j_5$ by $1, \ldots, 5$. The slope indices are:

$$A_1 = -(3 \times 5) - (1 \times 4) + (1 \times 4) + (3 \times 3) = -6$$

$$A_2 = -(3 \times 5) - (1 \times 4) + (1 \times 4) + (3 \times 6) = +3$$

$$A_3 = -(3 \times 3) - (1 \times 2) + (1 \times 3) + (3 \times 3) = +1$$

$$A_4 = -(3 \times 6) - (1 \times 4) + (1 \times 4) + (3 \times 2) = -12$$

$$A_5 = -(3 \times 3) - (1 \times 4) + (1 \times 1) + (3 \times 5) = +3$$

The two sequences suggested by the heuristic are therefore $2, 5, 3, 1, 4$ and $5, 2, 3, 1, 4$. The makespan under both these sequences is 32. Complete enumeration verifies that both sequences are optimal. ||

In contrast to the makespan objective, results with regard to the total completion time objective are harder to obtain. It can be shown that $F2 \,||\, \sum C_j$ is already strongly NP-hard. The proof of this result is somewhat involved and is therefore not presented here.

However, Theorem 6.1.8 facilitates the analysis of the flow time $Fm \mid p_{ij} = p_j \mid \sum C_j$ problem considerably and it can be shown fairly easily that SPT minimizes the total completion time in a proportionate flow shop.

6.2 Flow Shops with Limited Intermediate Storage

Consider m machines in series with *zero* intermediate storage between successive machines. When a machine finishes with the processing of a job, that job cannot

proceed to the next machine if that machine is busy; the job must remain on the first machine, which thus cannot start any processing of subsequent jobs. As stated before, this phenomenon is referred to as *blocking*.

In what follows only flow shops with zero intermediate storages are considered, since any flow shop with positive (but finite) intermediate storages between machines can be modeled as a flow shop with zero intermediate storages. This follows from the fact that a storage space capable of containing one job may be regarded as a *machine* on which the processing times of all jobs are equal to zero.

The problem of minimizing the makespan in a flow shop with zero intermediate storages is referred to in what follows as $Fm \mid block \mid C_{\max}$.

Let D_{ij} denote the time that job j actually departs machine i. Clearly, $D_{ij} \geq C_{ij}$. Equality holds when job j is not blocked. The time job j starts its processing at the first machine is denoted by D_{0j}. The following recursive relationships hold under sequence $j_1, \ldots, j_n$.

$$\begin{aligned} D_{i,j_1} &= \sum_{l=1}^{i} p_{l,j_1} \\ D_{i,j_k} &= \max(D_{i-1,j_k} + p_{i,j_k}, D_{i+1,j_{k-1}}) \\ D_{m,j_k} &= D_{m-1,j_k} + p_{m,j_k} \end{aligned}$$

For this model the makespan under a given permutation schedule can also be computed by determining the critical path in a directed graph. In this directed graph node (i, j_k) denotes the departure time of job j_k from machine i. In contrast to the graph in Section 6.1 for flow shops with unlimited intermediate storages, in this graph the arcs, rather than the nodes, have weights. Node (i, j_k), $i = 1, \ldots, m-1$; $\quad k = 1, \ldots, n-1$, has two outgoing arcs; one arc goes to node $(i+1, j_k)$ and has a weight or distance p_{i+1,j_k}, the other arc goes to node $(i-1, j_{k+1})$ and has weight zero. Node (m, j_k) has only one outgoing arc to node $(m-1, j_{k+1})$ with zero weight. Node (i, j_n) has only one outgoing arc to node $(i+1, j_n)$ with weight p_{i+1,j_n}. Node (m, j_n) has no outgoing arcs (see Figure 6.6). The $C_{\max}$ under sequence $j_1, \ldots, j_n$ is equal to the length of the maximum weight path from node $(0, j_1)$ to node (m, j_n).

Example 6.2.1 (Graph Representation of Flow Shop with Blocking)

Consider the instance of Example 6.1.1. Assume that the same 5 jobs with the same processing times have to traverse the same four machines. The only difference is that now there is zero intermediate storage between the machines. The directed graph and the Gantt chart corresponding to this situation is depicted in Figure 6.7. There is now only one critical path that determines the makespan of 35. ||

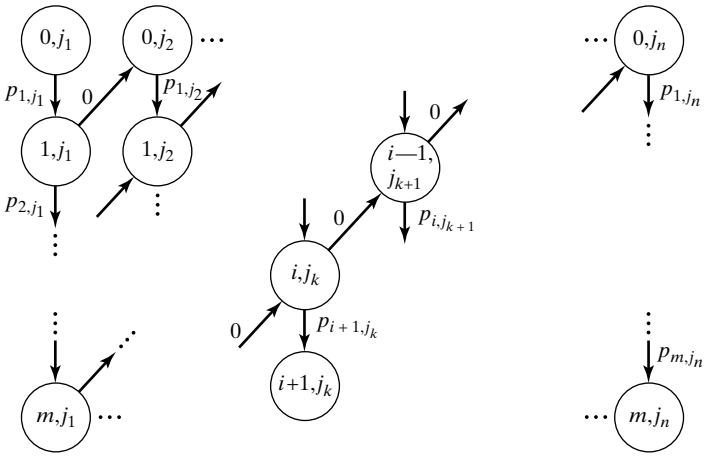


Fig. 6.6 Directed graph for the computation of the makespan

The following lemma shows that the reversibility property extends to flow shops with zero intermediate storage. Consider two m machine flow shops with blocking and let $p_{ij}^{(1)}$ and $p_{ij}^{(2)}$ denote the processing times of job j on machine i in the first and second flow shop respectively.

Lemma 6.2.2. *If*

$$p_{ij}^{(1)} = p_{m+1-i,j}^{(2)}$$

then sequence $j_1, \ldots, j_n$ in the first flow shop results in the same makespan as sequence $j_n, \ldots, j_1$ in the second flow shop.

Proof. It can be shown that there is a one-to-one correspondence between paths of equal weight in the two directed graphs corresponding to the two flow shops. This implies that the paths with maximal weights in the two directed graphs must have the same total weight. □

This reversibility result is similar to the result in Lemma 6.1.2. Actually, one can argue that the result in Lemma 6.1.2 is a special case of the result in Lemma 6.2.2. The unlimited intermediate storages can be regarded as sets of machines on which all processing is equal to zero.

Consider the $F2 \mid block \mid C_{\max}$ problem with two machines in series and zero intermediate storage in between. Note that in this flow shop, whenever a job starts its processing on the first machine, the preceding job starts its processing on the second machine. The time job j_k spends on machine 1, in process or blocked, is therefore $\max(p_{1,j_k}, p_{2,j_{k-1}})$. The first job in the sequence spends only p_{1,j_k} on machine 1. This makespan minimization problem is equivalent to a Travelling Salesman Problem with $n+1$ cities. Let the distance from city j

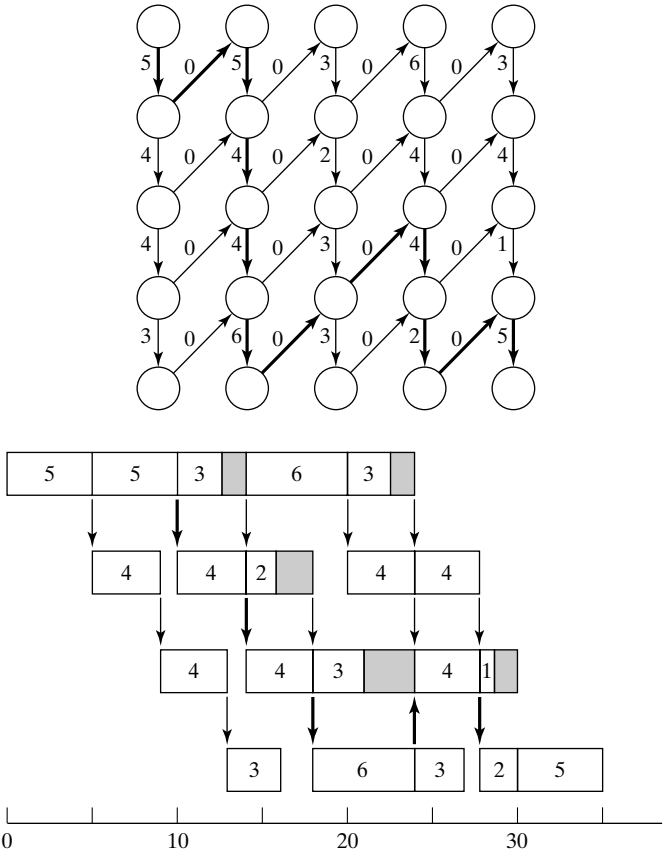


Fig. 6.7 Directed graph, critical path and Gantt chart (the numerical entries represent the processing times of the jobs and not the job indexes)

to city k be equal to

$$\begin{aligned} d_{0k} &= p_{1k} \\ d_{j0} &= p_{2j} \\ d_{jk} &= \max(p_{2j}, p_{1k}) \end{aligned}$$

The total distance travelled is then equal to the makespan of the flow shop. Actually, the distance matrix can be simplified somewhat. Instead of minimizing the makespan, one can minimize the total time between 0 and $C_{\max}$ that one of the two machines is either idle or blocked. The two objectives are equivalent since twice the makespan is equal to the sum of the $2n$ processing times plus

the sum of all idle times. Minimizing the sum of all idle times is equivalent to the following Travelling Salesman Problem with $n+1$ cities:

$$\begin{aligned} d_{0k} &= p_{1k} \\ d_{j0} &= p_{2j} \\ d_{jk} &= \mid\mid p_{2j} - p_{1k} \mid\mid \end{aligned}$$

The idle time on one of the machines, when job j starts on machine 2 and job k starts on machine 1, is the difference between the processing times p_{2j} and p_{1k}. If p_{2j} is larger than p_{1k} job k will be blocked for the time difference on machine 1, otherwise machine 2 will remain idle for the time difference. The distance matrix of this Travelling Salesman Problem is identical to the one discussed in Section 4.5. The values for b_0 and a_0 in Section 4.5 have to be chosen equal to zero.

The algorithm for the $1 \mid s_{jk} \mid C_{\max}$ problem with $s_{jk} = \parallel a_k - b_j \parallel$ can now be used for $F2 \mid block \mid C_{\max}$ as well, which implies that there exists an $O(n^2)$ algorithm for $F2 \mid block \mid C_{\max}$.

Example 6.2.3 (A Two Machine Flow Shop with Blocking and the TSP)

Consider a 4 job instance with processing times

jobs	1	2	3	4
$p_{1,j}$	2	3	3	9
$p_{2,j}$	8	4	6	2

This translates into a TSP with 5 cities. In the notation of Section 4.5 this instance of the TSP is specified by following a_j and b_j values.

cities	0	1	2	3	4
b_j	0	2	3	3	9
a_j	0	8	4	6	2

Applying Algorithm 4.5.5 on this instance results in the tour $0 \rightarrow 1 \rightarrow 4 \rightarrow 2 \rightarrow 3 \rightarrow 0$. Actually, two schedules are optimal for the flow shop with blocking, namely schedule $1, 4, 2, 3$ and schedule $1, 4, 3, 2$. These are different from the SPT(1)-LPT(2) schedules that are optimal in the case of unlimited buffers. With the same four jobs and unlimited buffers the following three schedules are optimal: $1, 3, 4, 2$; $1, 2, 3, 4$ and $1, 3, 2, 4$. ||

The three machine version of this problem cannot be described as a Travelling Salesman Problem and is known to be strongly NP-hard. The proof, however, is rather complicated and therefore omitted.

Certain special cases of $Fm \mid block \mid C_{\max}$ are tractable. Consider the proportionate case where $p_{1j} = \cdots = p_{mj} = p_j$, for $j = 1, \ldots, m$. That is, consider $Fm \mid block, p_{ij} = p_j \mid C_{\max}$.

Theorem 6.2.4. *A schedule is optimal for* $Fm \mid block, p_{ij} = p_j \mid C_{\max}$ *if and only if it is an SPT-LPT schedule.*

Proof. The makespan has to satisfy the inequality

$$C_{\max} \geq \sum_{j=1}^{n} p_j + (m-1) \max(p_1, \ldots, p_n),$$

as the R.H.S. is the optimal makespan when there are unlimited buffers between any two successive machines. Clearly, the makespan with limited or no buffers has to be at least as large. It suffices to show that the makespan under any SPT-LPT schedule is equal to the lower bound, while the makespan under any schedule that is not SPT-LPT is strictly larger than the lower bound.

That the makespan under any SPT-LPT schedule is equal to the lower bound can be shown easily. Under the SPT part of the schedule the jobs are never blocked. In other words, each job in this first part of the schedule, once started on the first machine, proceeds through the system without stopping. If in the SPT-LPT schedule $j_1, \ldots, j_n$ job j_k is the job with the longest processing time, then job j_k departs the system at

$$C_{j_k} = \sum_{l=1}^{k-1} p_{j_l} + m\, p_{j_k}.$$

The jobs in the LPT part of the sequence, of course, do experience blocking as shorter jobs follow longer jobs. However, it is clear that now, in this part of the schedule, a machine never has to wait for a job. Every time a machine has completed processing a job from the LPT part of the schedule, the next job is ready to start (as shorter jobs follow longer jobs). So, after job j_k has completed its processing on machine m, machine m remains continuously busy until it has completed all the remaining jobs. The makespan under an SPT-LPT schedule is therefore equal to the makespan under an SPT-LPT schedule in the case of unlimited buffers. SPT-LPT schedules therefore have to be optimal.

That SPT-LPT schedules are the only schedules that are optimal can be shown by contradiction. Suppose that another schedule, that is not SPT-LPT, is also optimal. Again, the job with the longest processing time, job j_k, contributes m times its processing to the makespan. However, there must be some job, say job j_h, (not the longest job) that is positioned in between two jobs that are both longer. If job j_h appears in the schedule before job j_k it remains on machine 1 for

an amount of time that is larger than its processing time, since it is blocked by the preceding job on machine 2. So its contribution to the makespan is strictly larger than its processing time, causing the makespan to be strictly larger than the lower bound. If job j_h appears in the schedule after job j_k, then the jobs following job j_k on machine m are not processed one after another without any idle times in between. After job j_h there is an idle time on machine m as the next job has a processing time that is strictly larger than the processing time of job j_h. □

There exists some similarity between this model and the proportionate flow shop model with unlimited intermediate storage. For the case with blocking it also can be shown that the SPT rule minimizes $\sum C_j$.

As with flow shops with unlimited intermediate storage, a fair amount of research has been done in the development of heuristics for the minimization of the makespan in flow shops with limited intermediate storage and blocking. One popular heuristic for $Fm \mid block \mid C_{\max}$ is the *Profile Fitting (PF)* heuristic, which works as follows: one job is selected to go first, possibly according to some scheme, e.g., the job with the smallest sum of processing times. This job, say job j_1, does not encounter any blocking and proceeds smoothly from one machine to the next, generating a profile. The profile is determined by its departure from machine i. If job j_1 corresponds to job k, then

$$D_{i,j_1} = \sum_{h=1}^{i} p_{h,j_1} = \sum_{h=1}^{i} p_{hk}$$

To determine which job should go second, every remaining unscheduled job is tried out. For each candidate job a computation is carried out to determine the amount of time machines are idle and the amount of time the job is blocked at a machine. The departure epochs of a candidate for the second position, say job j_2, can be computed recursively:

$$\begin{aligned} D_{1,j_2} &= \max(D_{1,j_1} + p_{1,j_2}, D_{2,j_1}) \\ D_{i,j_2} &= \max(D_{i-1,j_2} + p_{i,j_2}, D_{i+1,j_1}), \qquad i = 2, \ldots, m-1 \\ D_{m,j_2} &= \max(D_{m-1,j_2}, D_{m,j_1}) + p_{m,j_2} \end{aligned}$$

The time wasted at machine i, that is, the time the machine is either idle or blocked, is $D_{i,j_2} - D_{i,j_1} - p_{i,j_2}$. The sum of these idle and blocked times over all m machines is then computed. The candidate with the smallest total is selected as the second job.

After selecting the job that fits best as the job for second position, the new profile, i.e., the departure times of this second job from the m machines, is computed and the procedure repeats itself. From the remaining jobs in the set of unscheduled jobs the best fit is again selected and so on.

In this description the goodness of fit of a particular job was measured by the total time wasted on all m machines. Each machine was considered equally

important. It is intuitive that lost time on a bottleneck machine is worse than lost time on a machine that does not have much processing to do. When measuring the total amount of lost time, it may be appropriate to multiply each of these inactive time periods on a given machine by a factor that is proportional to the degree of congestion at that machine. The higher the degree of congestion at a particular machine, the larger the weight. One measure for the degree of congestion of a machine that is easy to calculate is simply the total amount of processing to be done on all the jobs at the machine in question. Experiments have shown that such a weighted version of the PF heuristic works quite well.

Example 6.2.5 (Application of the PF Heuristic)

Consider again 5 jobs and 4 machines. The processing times of the five jobs are in the tables in Examples 6.1.1 and 6.1.6. Assume that there is zero storage in between the successive machines.

Take as the first job the job with the smallest total processing time, i.e., job 3. Apply the unweighted PF heuristic. Each one of the four remaining jobs has to be tried out. If either job 1 or job 2 would go second, the total idle time of the four machines would be 11; if job 4 would go second the total idle time of the four machines would be 15 and if job 5 would be second, the total idle time would be 3. It is clear that job 5 is the best fit. Continuing in this manner the PF heuristic results in the sequence $3, 5, 1, 2, 4$ with a makespan equal to 32. From the fact that this makespan is equal to the minimum makespan in the case of unlimited intermediate storage, it follows that sequence $3, 5, 1, 2, 4$ is also optimal in the case of zero intermediate storage.

In order to study the effect of the selection of the first job, consider the application of the PF heuristic after selecting the job with the largest total processing time as the initial job. So job 2 goes first. Application of the unweighted PF heuristic leads to the sequence $2, 1, 3, 5, 4$ with makespan 35. This sequence is clearly not optimal. ||

Consider now a flow shop with zero intermediate storage that is subject to different operational procedures. A job, when it goes through the system, is not allowed to wait at any machine. That is, whenever it has completed its processing on one machine the next machine has to be idle, so that the job does not have to wait. In contrast to the blocking case where jobs are *pushed* down the line by machines upstream that have completed their processing, in this case the jobs are actually *pulled* down the line by machines that have become idle. This constraint is referred to as the *no-wait* constraint and minimizing the makespan in such a flow shop is referred to as the $Fm \mid nwt \mid C_{\max}$ problem. It is easy to see that $F2 \mid block \mid C_{\max}$ is equivalent to $F2 \mid nwt \mid C_{\max}$. However, when there are more than two machines in series the two problems are different. The $Fm \mid nwt \mid C_{\max}$ problem, in contrast to the $Fm \mid block \mid C_{\max}$ problem, can still be formulated as a Travelling Salesman Problem. The intercity distances

are

$$d_{jk} = \max_{1 \le i \le m} \Big(\sum_{h=1}^{i} p_{hj} - \sum_{h=1}^{i-1} p_{hk} \Big)$$

for $j, k = 0, \ldots, n$. When there are more than two machines in series this Travelling Salesman Problem is known to be strongly NP-hard.

6.3 Flexible Flow Shops with Unlimited Intermediate Storage

The flexible flow shop is a machine environment with c stages in series; at stage l, $l = 1, \ldots, c$, there are m_l identical machines in parallel. There is an unlimited intermediate storage between any two successive stages. The machine environment in the first example of Section 1.1 constitutes a flexible flow shop. Job j, $j = 1, \ldots, n$, has to be processed at each stage on one machine, any one will do. The processing times of job j at the various stages are $p_{1j}, p_{2j}, \ldots, p_{cj}$. Minimizing the makespan and total completion time are respectively referred to as $FFc \,||\, C_{\max}$ and $FFc \,||\, \sum C_j$. The parallel machine environment as well as the flow shop with unlimited intermediate storages are special cases of this machine environment. As this environment is rather complex only the special case with proportionate processing times, i.e., $p_{1j} = p_{2j} = \cdots = p_{cj} = p_j$, is considered here.

Consider $FFc \mid p_{ij} = p_j \mid C_{\max}$. One would expect the LPT heuristic to perform well in the nonpreemptive case and the LRPT heuristic to perform well in the preemptive case. Of course, the LPT rule cannot guarantee an optimal schedule; a single stage (a parallel machine environment) is already NP-hard. The worst case behaviour of the LPT rule when applied to multiple stages in series may be worse than when applied to a single stage.

Example 6.3.1 (Minimizing Makespan in a Flexible Flow Shop)

Consider two stages with two machines in parallel at the first stage and a single machine at the second stage. There are two jobs with $p_1 = p_2 = 100$ and a hundred jobs with $p_3 = p_4 = \cdots = p_{102} = 1$. It is clear that in order to minimize the makespan one long job should go at time zero on machine 1 and the 100 short jobs should be processed on machine 2 between time 0 and time 100. Under this schedule the makespan is 301. Under the LPT schedule the makespan is 400. ||

In a preemptive setting the LRPT rule is optimal for a single stage. When there are multiple stages this is not true any more. The LRPT schedule has the disadvantage that at the first stage all jobs are finished very late, leaving the machines at the second stage idle for a very long time.

Consider now the proportionate flexible flow shop problem $FFc \mid p_{ij} = p_j \mid \sum C_j$. The SPT rule is known to be optimal for a single stage and for

any number of stages with a single machine at each stage. Consider now the additional constraint where each stage has at least as many machines as the previous stage (the flow shop is said to be diverging).

Theorem 6.3.2. *The SPT rule is optimal for* $FFc \mid p_{ij} = p_j \mid \sum C_j$ *if each stage has at least as many machines as the preceding stage.*

Proof. Theorem 5.3.1 implies that SPT minimizes the total completion time when the flexible flow shop consists of a single stage. It is clear that SPT not only minimizes the total completion time in this case, but also the sum of the starting times (the only difference between the sum of the completion times and the sum of the starting times is the sum of the processing times, which is independent of the schedule).

In a proportionate flexible flow shop with c stages, the completion time of job j at the last stage occurs at the earliest cp_j time units after its starting time at the first stage.

Consider now a flexible flow shop with the same number of machines at each stage, say m. It is clear that under SPT each job when completed at one stage does not have to wait for processing at the next stage. Immediately after completion at one stage it can start its processing at the next stage (as all preceding jobs have smaller processing times than the current job). So, under SPT the sum of the completion times is equal to the sum of the starting times at the first stage plus $\sum_{j=1}^{n} cp_j$. As SPT minimizes the sum of the starting times at the first stage and job j must remain at least cp_j time units in the system, SPT has to be optimal. □

It is easy to verify that the SPT rule does not always lead to an optimal schedule for arbitrary proportionate flexible flow shops. A counter-example with only two stages can be found easily.

6.4 Discussion

This chapter has an emphasis on the makespan objective. In most machine environments the makespan is usually the easiest objective. In the flow shop environment, the makespan is already hard when there are three or more machines in series. Other objectives tend to be even more difficult.

In any case, some research has been done on flow shops with the total completion time objective. Minimizing the total completion time in a two machine flow shop, i.e., $F2 \mid\mid \sum C_j$, is already strongly NP-Hard. Several integer programming formulations have been proposed for this problem and various branch-and-bound approaches have been developed. Still, only instances with 50 jobs can be solved in a reasonable time. Minimizing the total completion time in a *proportionate* flow shop is, of course, very easy and can be solved via the SPT rule. Even the minimization of the total weighted completion time in a proportionate flow shop can be solved in polynomial time.

Flow shops with due date related objective functions have received very little attention in the literature. On the other hand, more complicated flow shops, e.g., robotic cells, have received a considerable amount of attention.

Exercises (Computational)

6.1. Consider $F4 \mid prmu \mid C_{\max}$ with the following 5 jobs under the given sequence $j_1, \ldots, j_5$.

jobs	j_1	j_2	j_3	j_4	j_5
p_{1,j_k}	5	3	6	4	9
p_{2,j_k}	4	8	2	9	13
p_{3,j_k}	7	8	7	6	5
p_{4,j_k}	8	4	2	9	1

Find the critical path and compute the makespan under the given sequence.

6.2. Write the integer programming formulation of $F4 \mid prmu \mid C_{\max}$ with the set of jobs in Exercise 6.1.

6.3. Apply the Slope heuristic to the set of jobs in Exercise 6.1. Is (Are) the sequence(s) generated actually optimal?

6.4. Consider $F4 \mid block \mid C_{\max}$ with 5 jobs and the same set of processing times as in Exercise 6.1. Assume there is no buffer in between any two successive machines. Apply the Profile Fitting heuristic to determine a sequence for this problem. Take job j_1 as the first job. If there are ties consider all the possibilities. Is (any one of) the sequence(s) generated optimal?

6.5. Consider $F4 \mid prmu \mid C_{\max}$ with the following jobs

jobs	1	2	3	4	5
$p_{1,j}$	18	16	21	16	22
$p_{2,j}$	6	5	6	6	5
$p_{3,j}$	5	4	5	5	4
$p_{4,j}$	4	2	1	3	4

(a) Can this problem be reduced to a similar problem with a smaller number of machines and the same optimal sequence?

(b) Determine whether Theorem 6.1.4 can be applied to the reduced problem.

(c) Find the optimal sequence.

6.6. Apply Algorithm 3.3.1 to find an optimal schedule for the proportionate flow shop $F3 \mid p_{ij} = p_j \mid \sum U_j$ with the following jobs.

jobs	1	2	3	4	5	6
p_j	5	3	4	4	9	3
d_j	17	19	21	22	24	24

6.7. Find the optimal schedule for the instance of the proportionate flow shop $F2 \mid p_{ij} = p_j \mid h_{\max}$ with the following jobs.

jobs	1	2	3	4	5
p_j	5	3	6	4	9
$h_j(C_j)$	$12\sqrt{C_1}$	72	$2C_3$	$54 + .5C_4$	$66 + \sqrt{C_5}$

6.8. Apply a variant of Algorithm 3.4.4 to find an optimal schedule for the instance of the proportionate flow shop $F2 \mid p_{ij} = p_j \mid \sum T_j$ with the following 5 jobs.

jobs	1	2	3	4	5
p_j	5	3	6	4	9
d_j	4	11	2	9	13

6.9. Consider $F2 \mid block \mid C_{\max}$ with zero intermediate storage and 4 jobs.

jobs	1	2	3	4
$p_{1,j}$	2	5	5	11
$p_{2,j}$	10	6	6	4

(a) Apply Algorithm 4.4.5 to find the optimal sequence.

(b) Find the optimal sequence when there is an unlimited intermediate storage.

6.10. Find the optimal schedule for a proportionate flexible flow shop $FF2 \mid p_{ij} = p_j \mid \sum C_j$ with three machines at the first stage and one machine at the second stage. There are 5 jobs. Determine whether SPT is optimal.

jobs	1	2	3	4	5
p_j	2	2	2	2	5

Exercises (Theory)

6.11. Consider the problem $Fm \mid\mid C_{\max}$. Assume that the schedule does allow one job to pass another while they are waiting for processing on a machine.

(a) Show that there always exists an optimal schedule that does not require sequence changes between machines 1 and 2 and between machines $m-1$ and m. (*Hint:* By contradiction. Suppose the optimal schedule requires a sequence change between machines 1 and 2. Modify the schedule in such a way that there is no sequence change and the makespan remains the same.)

(b) Find an instance of $F4 \mid\mid C_{\max}$ where a sequence change between machines 2 and 3 results in a smaller makespan than in the case where sequence changes are not allowed.

6.12. Consider $Fm \mid prmu \mid C_{\max}$. Let

$$p_{i1} = p_{i2} = \cdots = p_{in} = p_i$$

for $i = 2, \ldots, m-1$. Furthermore, let

$$p_{11} \leq p_{12} \leq \cdots \leq p_{1n}$$

and

$$p_{m1} \geq p_{m2} \geq \cdots \geq p_{mn}.$$

Show that sequence $1, 2, \ldots, n$, i.e., SPT(1)-LPT(m), is optimal.

6.13. Consider $Fm \mid prmu \mid C_{\max}$ where $p_{ij} = a_i + b_j$, i.e., the processing time of job j on machine i consists of a component that is job dependent and a component that is machine dependent. Find the optimal sequence when $a_1 \leq a_2 \leq \cdots \leq a_m$ and prove your result.

6.14. Consider $Fm \mid prmu \mid C_{\max}$. Let $p_{ij} = a_j + ib_j$ with $b_j > -a_j/m$.

(a) Find the optimal sequence.

(b) Does the Slope heuristic lead to an optimal schedule?

6.15. Consider $F2 \mid\mid C_{\max}$.

(a) Show that the Slope heuristic for two machines reduces to sequencing the jobs in decreasing order of $p_{2j} - p_{1j}$.

(b) Show that the Slope heuristic is not necessarily optimal for two machines.

(c) Show that sequencing the jobs in decreasing order of p_{2j}/p_{1j} is not necessarily optimal either.

6.16. Consider $F3 \mid\mid C_{\max}$. Assume

$$\max_{j \in \{1,\ldots,n\}} p_{2j} \leq \min_{j \in \{1,\ldots,n\}} p_{1j}$$

and

$$\max_{j\in\{1,\ldots,n\}} p_{2j} \leq \min_{j\in\{1,\ldots,n\}} p_{3j}.$$

Show that the optimal sequence is the same as the optimal sequence for $F2 \mid\mid C_{\max}$ with processing times p'_{ij} where $p'_{1j} = p_{1j} + p_{2j}$ and $p'_{2j} = p_{2j} + p_{3j}$.

6.17. Show that in the proportionate flow shop problem $Fm \mid p_{ij} = p_j \mid C_{\max}$ a permutation sequence is optimal in the class of schedules that do allow sequence changes midstream.

6.18. Show that if in a sequence for $F2 \mid\mid C_{\max}$ any two adjacent jobs j and k satisfy the condition

$$\min(p_{1j}, p_{2k}) \leq \min(p_{1k}, p_{2j})$$

then the sequence minimizes the makespan. (Note that this is a sufficiency condition and not a necessary condition for optimality.)

6.19. Show that for $Fm \mid prmu \mid C_{\max}$ the makespan under an arbitrary permutation sequence cannot be longer than m times the makespan under the optimal sequence. Show how this worst case bound actually can be attained.

6.20. Consider a proportionate flow shop with two objectives, namely the total completion time and the maximum lateness, i.e., $Fm \mid p_{ij} = p_j \mid \sum C_j + L_{\max}$. Develop a polynomial time algorithm for this problem. (*Hint:* Parametrize on the maximum lateness. Assume the maximum lateness to be z; then consider new due dates $d_j + z$ which basically are hard deadlines. Start out with the SPT rule and modify when necessary.)

6.21. Consider a proportionate flow shop with n jobs. Assume that there are no two jobs with equal processing times. Determine the number of different SPT-LPT schedules.

6.22. Consider $Fm \mid prmu, p_{ij} = p_j \mid \sum w_j C_j$. Show that if $w_j/p_j > w_k/p_k$ and $p_j < p_k$, then there exists an optimal sequence in which job j precedes job k.

6.23. Consider the following hybrid between $Fm \mid prmu \mid C_{\max}$ and $Fm \mid block \mid C_{\max}$. Between some machines there is no intermediate storage and between other machines there is an infinite intermediate storage. Suppose a job sequence is given. Give a description of the graph through which the length of the makespan can be computed.

Comments and References

The solution for the $F2 \mid\mid C_{\max}$ problem is presented in the famous paper by S.M. Johnson (1954). The integer programming formulation of $Fm \mid\mid C_{\max}$ is due to

Wagner (1959) and the NP-Hardness proof for $F3 \mid\mid C_{\max}$ is from Garey, Johnson and Sethi (1976). A definition of SPT-LPT schedules appears in Pinedo (1982). Theorem 6.1.9 is from Eck and Pinedo (1988). For results regarding proportionate flow shops see Ow (1985), Pinedo (1985), and Shakhlevich, Hoogeveen and Pinedo (1998). For an overview of $Fm \mid\mid C_{\max}$ models with special structures that can be solved easily, see Monma and Rinnooy Kan (1983); their framework includes the results obtained earlier by Smith, Panwalkar and Dudek (1975, 1976) and Szwarc (1971, 1973, 1978). The slope heuristic for permutation flow shops is from Palmer (1965). Many other heuristics have been developed for $Fm \mid\mid C_{\max}$; see, for example, Campbell, Dudek and Smith (1970), Gupta (1972), Baker (1975), Dannenbring (1977), Widmer and Hertz (1989) and Taillard (1990). For complexity results with regard to various objective functions, see Gonzalez and Sahni (1978b) and Du and Leung (1993a, 1993b).

The flow shop with limited intermediate storage $Fm \mid block \mid C_{\max}$ is studied in detail by Levner (1969), Reddy and Ramamoorthy (1972) and Pinedo (1982). The reversibility result in Lemma 6.2.1 is due to Muth (1979). The Profile Fitting heuristic is from McCormick, Pinedo, Shenker and Wolf (1989). Wismer (1972) establishes the link between $Fm \mid nwt \mid C_{\max}$ and the Travelling Salesman Problem. Sahni and Cho (1979a), Papadimitriou and Kannelakis (1980) and Röck (1984) obtain complexity results for $Fm \mid nwt \mid C_{\max}$. Goyal and Sriskandrajah (1988) present a review of complexity results and approximation algorithms for $Fm \mid nwt \mid \gamma$. For an overview of models in the classes $Fm \mid\mid \gamma$, $Fm \mid block \mid \gamma$ and $Fm \mid nwt \mid \gamma$, see Hall and Sriskandarajah (1996).

Theorem 6.3.2 is from Eck and Pinedo (1988). For makespan results with regard to the flexible flow shops see Sriskandarajah and Sethi (1989). Yang, Kreipl and Pinedo (2000) present heuristics for the flexible flow shop with the total weighted tardiness as objective. For more applied issues concerning flexible flow shops, see Hodgson and McDonald (1981a, 1981b, 1981c).

For research concerning the two machine flow shop with the total completion time objective, see van de Velde (1990), Della Croce, Narayan and Tadei (1996), Shakhlevich, Hoogeveen and Pinedo (1998), Della Croce, Ghirardi and Tadei (2002), Akkan and Karabati (2004), and Hoogeveen, van Norden and van de Velde (2006).

Dawande, Geismar, Sethi and Sriskandarajah (2007) present an extensive overview of one of the more important application areas of flow shops, namely robotic cells.

Chapter 7

Job Shops (Deterministic)

This chapter deals with multi-operation models that are different from the flow shop models discussed in the previous chapter. In a flow shop model all jobs follow the same route. When the routes are fixed, but not necessarily the same for each job, the model is called a job shop. If a job in a job shop has to visit certain machines more than once, the job is said to recirculate. Recirculation is a common phenomenon in the real world. For example, in semiconductor manufacturing jobs have to recirculate several times before they complete all their processing.

The first section focuses on representations and formulations of the classical job shop problem with the makespan objective and no recirculation. It also describes a branch-and-bound procedure that is designed to find the optimal solution. The second section describes a popular heuristic for job shops with the makespan objective and no recirculation. This heuristic is typically referred to as the *Shifting Bottleneck* heuristic. The third section focuses on a more elaborate version of the shifting bottleneck heuristic that is designed specifically for the total weighted tardiness objective. The fourth section describes an application of a constraint programming procedure for the minimization of the makespan. The last section discusses possible extensions.

7.1 Disjunctive Programming and Branch-and-Bound

Consider $J2 \mid\mid C_{\max}$. There are two machines and n jobs. Some jobs have to be processed first on machine 1 and then on machine 2, while the remaining jobs

M.L. Pinedo, *Scheduling*, DOI: 10.1007/978-0-387-78935-4_7,

have to be processed first on machine 2 and then on machine 1. The processing time of job j on machine 1 (2) is p_{1j} (p_{2j}). The objective is to minimize the makespan.

This problem can be reduced to $F2 \mid\mid C_{\max}$ as follows. Let $J_{1,2}$ denote the set of jobs that have to be processed first on machine 1, and $J_{2,1}$ the set of jobs that have to be processed first on machine 2. Observe that when a job from $J_{1,2}$ has completed its processing on machine 1, postponing its processing on machine 2 does not affect the makespan as long as machine 2 is kept busy. The same can be said about a job from $J_{2,1}$; if such a job has completed its processing on machine 2, postponing its processing on machine 1 (as long as machine 1 is kept busy) does not affect the makespan. Hence a job from $J_{1,2}$ has on machine 1 a higher priority than any job from $J_{2,1}$, while a job from $J_{2,1}$ has on machine 2 a higher priority than any job from $J_{1,2}$. It remains to be determined in what sequence jobs in $J_{1,2}$ go through machine 1 and jobs in $J_{2,1}$ go through machine 2. The first of these two sequences can be determined by considering $J_{1,2}$ as an $F2 \mid\mid C_{\max}$ problem with machine 1 set up first and machine 2 set up second and the second sequence can be determined by considering $J_{2,1}$ as another $F2 \mid\mid C_{\max}$ problem with machine 2 set up first and machine 1 second. This leads to SPT(1)-LPT(2) sequences for each of the two sets, with priorities between sets as specified above.

This two machine problem is one of the few job shop scheduling problems for which a polynomial time algorithm can be found. The few other job shop scheduling problems for which polynomial time algorithms can be obtained usually require all processing times to be either 0 or 1.

The remainder of this section is dedicated to the $Jm \mid\mid C_{\max}$ problem with arbitrary processing times and no recirculation.

Minimizing the makespan in a job shop without recirculation, $Jm \mid\mid C_{\max}$, can be represented in a very nice way by a disjunctive graph. Consider a directed graph G with a set of nodes N and two sets of arcs A and B. The nodes N correspond to all the operations (i, j) that must be performed on the n jobs. The so-called *conjunctive* (solid) arcs A represent the routes of the jobs. If arc $(i, j) \rightarrow (k, j)$ is part of A, then job j has to be processed on machine i before it is processed on machine k, i.e., operation (i, j) precedes operation (k, j). Two operations that belong to two different jobs and that have to be processed on the same machine are connected to one another by two so-called *disjunctive* (broken) arcs that go in opposite directions. The disjunctive arcs B form m cliques of double arcs, one clique for each machine. (A clique is a term in graph theory that refers to a graph in which any two nodes are connected to one another; in this case each connection within a clique consists of a pair of disjunctive arcs.) All operations (nodes) in the same clique have to be done on the same machine. All arcs emanating from a node, conjunctive as well as disjunctive, have as length the processing time of the operation that is represented by that node. In addition there is a source U and a sink V, which are dummy nodes. The source node U has n conjunctive arcs emanating to the first operations of the n jobs and the sink node V has n conjunctive arcs coming

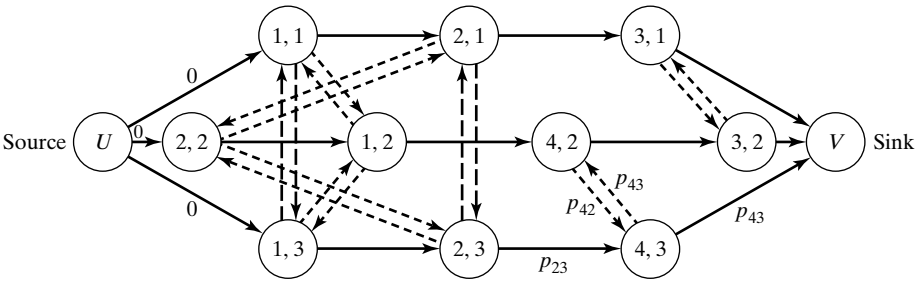


Fig. 7.1 Directed graph for job shop with makespan as objective

in from all the last operations. The arcs emanating from the source have length zero (see Figure 7.1). This graph is denoted by $G = (N, A, B)$.

A feasible schedule corresponds to a *selection* of one disjunctive arc from each pair such that the resulting directed graph is acyclic. This implies that a selection of disjunctive arcs from a clique has to be acyclic. Such a selection determines the sequence in which the operations are to be performed on that machine. That a selection from a clique has to be acyclic can be argued as follows: If there were a cycle within a clique, a feasible sequence of the operations on the corresponding machine would not have been possible. It may not be immediately obvious why there should not be any cycle formed by conjunctive arcs and disjunctive arcs from different cliques. However, such a cycle would correspond also to a situation that is infeasible. For example, let (h, j) and (i, j) denote two consecutive operations that belong to job j and let (i, k) and (h, k) denote two consecutive operations that belong to job k. If under a given schedule operation (i, j) precedes operation (i, k) on machine i and operation (h, k) precedes operation (h, j) on machine h, then the graph contains a cycle with four arcs, two conjunctive arcs and two disjunctive arcs from different cliques. Such a schedule is physically impossible. Summarizing, if D denotes the subset of the selected disjunctive arcs and the graph $G(D)$ is defined by the set of conjunctive arcs and the subset D, then D corresponds to a feasible schedule if and only if $G(D)$ contains no directed cycles.

The makespan of a feasible schedule is determined by the longest path in $G(D)$ from the source U to the sink V. This longest path consists of a set of operations of which the first starts at time 0 and the last finishes at the time of the makespan. Each operation on this path is immediately followed by either the next operation on the same machine or the next operation of the same job on another machine. The problem of minimizing the makespan is reduced to finding a selection of disjunctive arcs that minimizes the length of the longest path (that is, the *critical* path).

There are several mathematical programming formulations for the job shop without recirculation, including a number of integer programming formulations. However, the formulation most often used is the so-called disjunctive program-

ming formulation (see also Appendix A). This disjunctive programming formulation is closely related to the disjunctive graph representation of the job shop.

To present the disjunctive programming formulation, let the variable y_{ij} denote the starting time of operation (i, j). Recall that set N denotes the set of all operations (i, j), and set A the set of all routing constraints $(i, j) \rightarrow (k, j)$ that require job j to be processed on machine i before it is processed on machine k. The following mathematical program minimizes the makespan.

$$\text{minimize } C_{\max}$$

subject to

$$\begin{aligned}
y_{kj} - y_{ij} &\geq p_{ij} && && \text{for all } (i,j) \rightarrow (k,j) \in A \\
C_{\max} - y_{ij} &\geq p_{ij} && && \text{for all } (i,j) \in N \\
y_{ij} - y_{il} &\geq p_{il} \quad \text{or} \quad y_{il} - y_{ij} \geq p_{ij} && && \text{for all } (i,l) \text{ and } (i,j), \quad i = 1, \ldots, m \\
y_{ij} &\geq 0 && && \text{for all } (i,j) \in N
\end{aligned}$$

In this formulation, the first set of constraints ensure that operation (k, j) cannot start before operation (i, j) is completed. The third set of constraints are called the disjunctive constraints; they ensure that some ordering exists among operations of different jobs that have to be processed on the same machine. Because of these constraints this formulation is referred to as a disjunctive programming formulation.

Example 7.1.1 (Disjunctive Programming Formulation)

Consider the following example with four machines and three jobs. The route, i.e., the machine sequence, as well as the processing times are given in the table below.

jobs	*machine sequence*	*processing times*
1	1, 2, 3	$p_{11} = 10, \quad p_{21} = 8, \quad p_{31} = 4$
2	2, 1, 4, 3	$p_{22} = 8, \quad p_{12} = 3, \quad p_{42} = 5, \quad p_{32} = 6$
3	1, 2, 4	$p_{13} = 4, \quad p_{23} = 7, \quad p_{43} = 3$

The objective consists of the single variable $C_{\max}$. The first set of constraints consists of seven constraints: two for job 1, three for job 2 and two for job 3. For example, one of these is

$$y_{21} - y_{11} \geq 10 \ \ (= p_{11}).$$

The second set consists of ten constraints, one for each operation. An example is

$$C_{\max} - y_{11} \geq 10 \ \ (= p_{11}).$$

The set of disjunctive constraints contains eight constraints: three each for machines 1 and 2 and one each for machines 3 and 4 (there are three operations to be performed on machines 1 and 2 and two operations on machines 3 and 4). An example of a disjunctive constraint is

$$y_{11} - y_{12} \geq 3 \ \ (= p_{12}) \quad \text{or} \quad y_{12} - y_{11} \geq 10 \ \ (= p_{11}).$$

The last set includes ten nonnegativity constraints, one for each starting time. ||

That a scheduling problem can be formulated as a disjunctive program does not imply that there is a standard solution procedure available that will work satisfactorily. Minimizing the makespan in a job shop is a very hard problem and solution procedures are either based on enumeration or on heuristics.

To obtain optimal solutions branch-and-bound methods are required. The branching as well as the bounding procedures that are applicable to this problem are usually of a special design. In order to describe one of the branching procedures a specific class of schedules is considered.

Definition 7.1.2 (Active Schedule). *A feasible schedule is called active if it cannot be altered in any way such that some operation is completed earlier and no other operation is completed later.*

A schedule being active implies that when a job arrives at a machine, this job is processed in the prescribed sequence as early as possible. An active schedule cannot have any idle period in which the operation of a waiting job could fit.

From the definition it follows that an active schedule has the property that it is impossible to reduce the makespan without increasing the starting time of some operation. Of course, there are many different active schedules. It can be shown that there exists among all possible schedules an active schedule that minimizes the makespan.

A branching scheme that is often used is based on the generation of all active schedules. All such active schedules can be generated by a simple algorithm. In this algorithm Ω denotes the set of all operations of which all predecessors already have been scheduled (i.e., the set of all schedulable operations) and r_{ij} the earliest possible starting time of operation (i, j) in Ω. The set Ω' is a subset of set Ω.

Algorithm 7.1.3 (Generation of all Active Schedules)

Step 1. (Initial Condition)

Let Ω contain the first operation of each job;
Let $r_{ij} = 0$, for all $(i, j) \in \Omega$.

Step 2. (Machine Selection)

Compute for the current partial schedule

$$t(\Omega) = \min_{(i,j)\in\Omega} \{r_{ij} + p_{ij}\}$$

and let i^* denote the machine on which the minimum is achieved.

Step 3. (Branching)

Let Ω' denote the set of all operations (i^*, j) on machine i^* such that

$$r_{i^*j} < t(\Omega).$$

For each operation in Ω' consider an (extended) partial schedule with that operation as the next one on machine i^*.
For each such (extended) partial schedule delete the operation from Ω, include its immediate follower in Ω and return to Step 2. ||

Algorithm 7.1.3 is the basis for the branching process. Step 3 performs the branching from the node that is characterized by the given partial schedule; the number of branches is equal to the number of operations in Ω'. With this algorithm one can generate the entire tree and the nodes at the very bottom of the tree correspond to all the active schedules.

So a node $\mathcal{V}$ in the tree corresponds to a partial schedule and the partial schedule is characterized by a selection of disjunctive arcs that corresponds to the order in which all the predecessors of a given set Ω have been scheduled. A branch out of node $\mathcal{V}$ corresponds to the selection of an operation $(i^*, j) \in \Omega'$ as the next one to go on machine i^*. The disjunctive arcs $(i^*, j) \rightarrow (i^*, k)$ then have to be added to machine i^* for all operations (i^*, k) still to be scheduled on machine i^*. This implies that the newly created node at the lower level, say node $\mathcal{V}'$, which corresponds to a partial schedule with only one more operation in place, contains various additional disjunctive arcs that are now selected (see Figure 7.2). Let D' denote the set of disjunctive arcs selected at the newly created node. Refer to the graph that includes all the conjunctive arcs and set D' as graph $G(D')$. The number of branches sprouting from node $\mathcal{V}$ is equal to the number of operations in Ω'.

To find a lower bound for the makespan at node $\mathcal{V}'$, consider graph $G(D')$. The length of the critical path in this graph already results in a lower bound for the makespan at node $\mathcal{V}'$. Call this lower bound $LB(\mathcal{V}')$. Better (higher) lower bounds for this node can be obtained as follows.

Consider machine i and assume that all *other* machines are allowed to process, at any point in time, multiple operations simultaneously (since not all disjunctive arcs have been selected yet in $G(D')$, it may be the case that, at some points in time, multiple operations require processing on the same machine at the same time). However, machine i must process its operations one after another. First, compute the earliest possible starting times r_{ij} of all the operations (i, j) on machine i; that is, determine in graph $G(D')$ the length

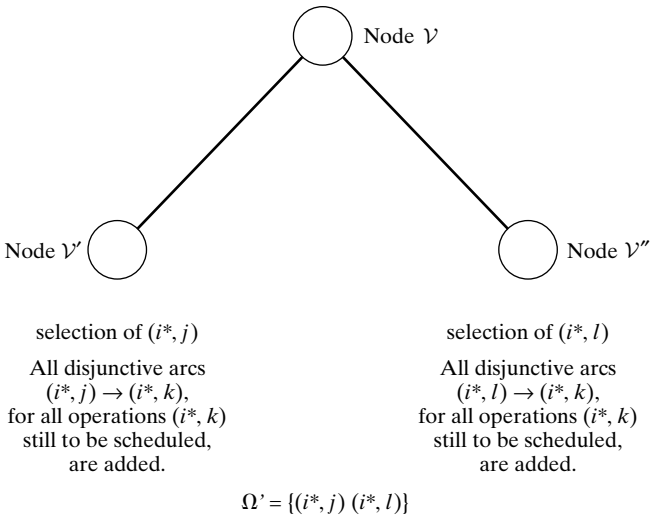


Fig. 7.2 Branching tree for branch-and-bound approach

of the longest path from the source to node (i, j). Second, for each operation (i, j) on machine i, compute the minimum amount of time needed between the completion of operation (i, j) and the lower bound $LB(\mathcal{V}')$, by determining the longest path from node (i, j) to the sink in $G(D')$. This amount of time, together with the lower bound on the makespan, translates into a due date d_{ij} for operation (i, j), i.e., d_{ij} is equal to $LB(\mathcal{V}')$ minus the length of the longest path from node (i, j) to the sink plus p_{ij}. Consider now the problem of sequencing the operations on machine i as a single machine problem with jobs arriving at different release dates, no preemptions allowed and the maximum lateness as the objective to be minimized, i.e., $1 \mid r_j \mid L_{\max}$ (see Section 3.2). Even though this problem is strongly NP-hard, there are relatively effective algorithms that generate good solutions. The optimal sequence obtained for this problem implies a selection of disjunctive arcs that can be added (temporarily) to D'. This then may lead to a longer overall critical path in the graph, a larger makespan and a better (higher) lower bound for node $\mathcal{V}'$. At node $\mathcal{V}'$ this can be done for each of the m machines separately. The largest makespan obtained this way can be used as a lower bound at node $\mathcal{V}'$. Of course, the temporary disjunctive arcs inserted to obtain the lower bound are deleted as soon as the best lower bound is determined.

Although it appears somewhat of a burden to have to solve m strongly NP-hard scheduling problems in order to obtain one lower bound for another strongly NP-hard problem, this type of bounding procedure has performed reasonably well in computational experiments.

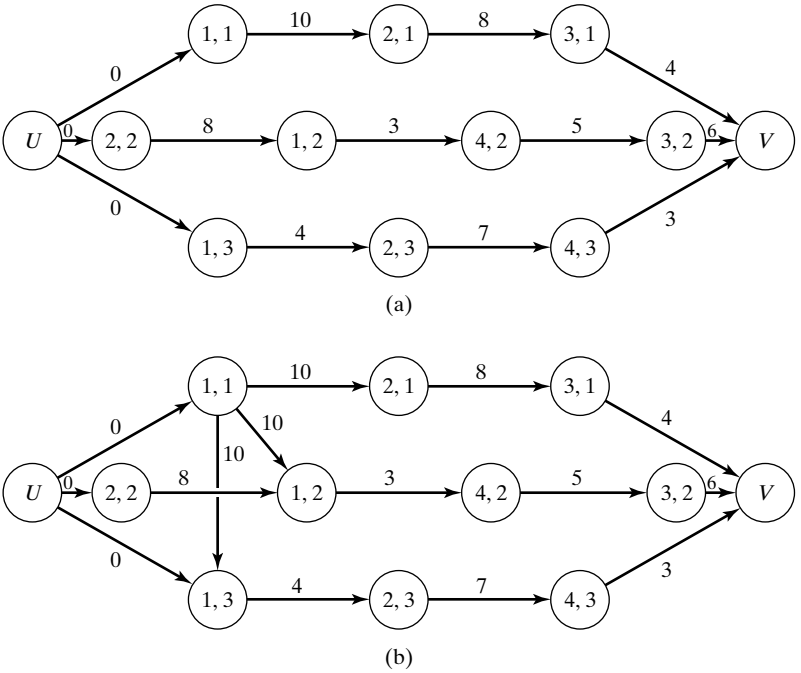


Fig. 7.3 Precedence graphs at Level 1 in Example 7.1.4

Example 7.1.4 (Application of Branch-and-Bound)
Consider the instance described in Example 7.1.1. The initial graph contains only conjunctive arcs and is depicted in Figure 7.3.a. The makespan corresponding to this graph is 22. Applying the branch-and-bound procedure to this instance results in the following branch-and-bound tree.

Level 1: Applying Algorithm 7.1.3 yields

$$\begin{aligned} \Omega &= \{(1,1),\ (2,2),\ (1,3)\}, \\ t(\Omega) &= \min\,(0+10, 0+8, 0+4) = 4, \\ i^* &= 1, \\ \Omega' &= \{(1,1),(1,3)\}. \end{aligned}$$

So there are two nodes of interest at level 1, one corresponding to operation $(1,1)$ being processed first on machine 1 and the other to operation $(1,3)$ being processed first on machine 1.

If operation $(1,1)$ is scheduled first, then the two disjunctive arcs depicted in Figure 7.3.b are added to the graph. The node is characterized by the two disjunctive arcs

$$(1,1) \rightarrow (1,2),$$

$$(1,1) \rightarrow (1,3).$$

The addition of these two disjunctive arcs immediately increases the lower bound on the makespan to 24. In order to improve this lower bound one can generate for machine 1 an instance of $1 \mid r_j \mid L_{\max}$. The release date of job j in this single machine problem is determined by the longest path from the source U to node $(1,j)$ in Figure 7.3.b. The due date of job j is computed by finding the longest path from node $(1,j)$ to the sink, subtracting p_{1j} from the length of this longest path, and subtracting the resulting value from 24. These computations lead to the following single machine problem for machine 1.

jobs	1	2	3
p_{1j}	10	3	4
r_{1j}	0	10	10
d_{1j}	10	13	14

The sequence that minimizes $L_{\max}$ is $1,2,3$ with $L_{\max} = 3$. This implies that a lower bound for the makespan at the corresponding node is $24 + 3 = 27$. An instance of $1 \mid r_j \mid L_{\max}$ corresponding to machine 2 can be generated in the same way. The release dates and due dates also follow from Figure 7.3.b (assuming a makespan of 24), and are as follows.

jobs	1	2	3
p_{2j}	8	8	7
r_{2j}	10	0	14
d_{2j}	20	10	21

The optimal sequence is $2,1,3$ with $L_{\max} = 4$. This yields a better lower bound for the makespan at the node that corresponds to operation $(1,1)$ being scheduled first, i.e., $24 + 4 = 28$. Analyzing machines 3 and 4 in the same way does not yield a better lower bound.

The second node at Level 1 corresponds to operation $(1,3)$ being scheduled first. If $(1,3)$ is scheduled to go first, two different disjunctive arcs are added to the original graph, yielding a lower bound of 26. The associated instance of the maximum lateness problem for machine 1 has an optimal sequence $3,1,2$ with $L_{\max} = 2$. This implies that the lower bound for the makespan at this node, corresponding to operation $(1,3)$ scheduled first, is also equal to 28. Analyzing machines 2, 3 and 4 does not result in a better lower bound.

The next step is to branch from node (1,1) at Level 1 and generate the nodes at the next level.

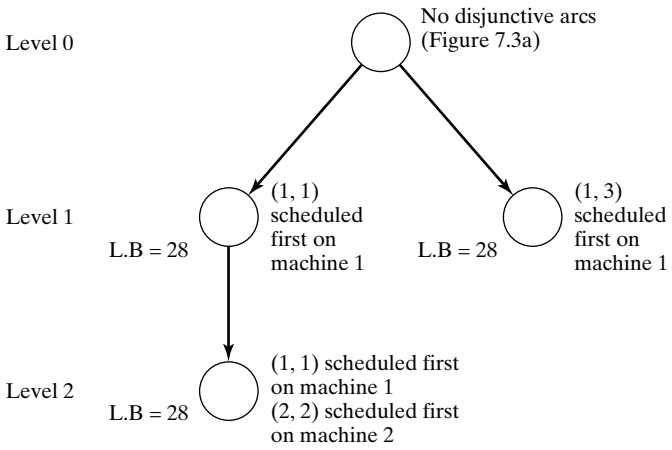


Fig. 7.4 Branching tree in Example 7.1.4

Level 2: Applying Algorithm 7.1.3 now yields

$$\begin{aligned} \Omega &= \{(2,2),\ \ (2,1),\ \ (1,3)\}, \\ t(\Omega) &= \min\,(0+8, 10+8, 10+4) = 8, \\ i^* &= 2, \\ \Omega' &= \{(2,2)\}. \end{aligned}$$

There is one node of interest at this part of Level 2, the node corresponding to operation $(2,2)$ being processed first on machine 2 (see Figure 7.4). Two disjunctive arcs are added to the graph, namely $(2,2) \rightarrow (2,1)$ and $(2,2) \rightarrow (2,3)$. So this node is characterized by a total of four disjunctive arcs:

$$(1,1) \rightarrow (1,2),$$

$$(1,1) \rightarrow (1,3),$$

$$(2,2) \rightarrow (2,1),$$

$$(2,2) \rightarrow (2,3).$$

This leads to an instance of $1 \mid r_j \mid L_{\max}$ for machine 1 with the following release dates and due dates (assuming a makespan of 28).

jobs	1	2	3
p_{1j}	10	3	4
r_{1j}	0	10	10
d_{1j}	14	17	18

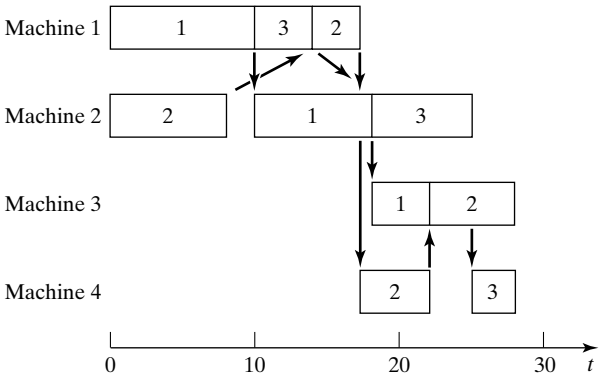


Fig. 7.5 Gantt chart for $J4 \mid\mid C_{\max}$ (Example 7.1.4)

The optimal job sequence is $1, 3, 2$ and $L_{\max} = 0$. This implies that the lower bound for the makespan at the corresponding node is $28 + 0 = 28$. Analyzing machines 2, 3 and 4 in the same way does not increase the lower bound.

Continuing the branch-and-bound procedure results in the following job sequences for the four machines.

machine	*job sequence*
1	$1, 3, 2$ (or $1, 2, 3$)
2	$2, 1, 3$
3	$1, 2$
4	$2, 3$

The makespan under this optimal schedule is 28 (see Figure 7.5). ||

The approach described above is based on complete enumeration and is guaranteed to lead to an optimal schedule. However, with a large number of machines and a large number of jobs the computation time is prohibitive. Already with 20 machines and 20 jobs it is hard to find an optimal schedule.

It is therefore necessary to develop heuristics that lead to reasonably good schedules in a reasonably short time. The next section describes a well-known heuristic with an excellent track record.

7.2 The Shifting Bottleneck Heuristic and the Makespan

One of the most successful heuristic procedures developed for $Jm \mid\mid C_{\max}$ is the *Shifting Bottleneck* heuristic.

In the following overview of the Shifting Bottleneck heuristic M denotes the set of all m machines. In the description of an iteration of the heuristic it is

assumed that in previous iterations a selection of disjunctive arcs already has been fixed for a subset M_0 of machines. So for each one of the machines in M_0 a sequence of operations has already been determined.

An iteration determines which machine in $M - M_0$ has to be included next in set M_0. The sequence in which the operations on this machine have to be processed is also generated in this iteration. In order to select the machine to be included next in M_0, an attempt is made to determine which one of the machines still to be scheduled would cause in one sense or another the severest disruption. To determine this, the original directed graph is modified by deleting *all* disjunctive arcs of the machines still to be scheduled (i.e., the machines in set $M - M_0$) and keeping only the relevant disjunctive arcs of the machines in set M_0 (one from every pair). Call this graph G'. Deleting all disjunctive arcs of a specific machine implies that all operations on this machine, which originally were supposed to be done on this machine one after another, now may be done in parallel (as if the machine has infinite capacity, or equivalently, each one of these operations has the machine for itself). The graph G' has one or more critical paths that determine the corresponding makespan. Call this makespan $C_{\max}(M_0)$.

Suppose that operation (i, j), $i \in \{M - M_0\}$, has to be processed in a time window of which the release date and due date are determined by the critical (longest) paths in G', i.e., the release date is equal to the longest path in G' from the source U to node (i, j) and the due date is equal to $C_{\max}(M_0)$, minus the longest path from node (i, j) to the sink, plus p_{ij}. Consider each of the machines in $M - M_0$ as a separate $1 \mid r_j \mid L_{\max}$ problem. As stated in the previous section this problem is strongly NP-hard, but procedures have been developed that perform reasonably well. The minimum $L_{\max}$ of the single machine problem corresponding to machine i is denoted by $L_{\max}(i)$ and is a measure of the criticality of machine i.

After solving all these single machine problems, the machine with the *largest* maximum lateness is chosen. Among the remaining machines, this machine is in a sense the most critical or the "bottleneck" and therefore the one to be included next in M_0. Label this machine k, call its maximum lateness $L_{\max}(k)$ and schedule it according to the optimal solution obtained for the single machine problem associated with this machine. If the disjunctive arcs that specify the sequence of operations on machine k are inserted in graph G', then the makespan of the current partial schedule increases by at least $L_{\max}(k)$, that is,

$$C_{\max}(M_0 \cup k) \geq C_{\max}(M_0) + L_{\max}(k).$$

Before starting the next iteration and determining the next machine to be scheduled, one additional step has to be done within the current iteration. In this additional step all the machines in the original set M_0 are resequenced in order to see if the makespan can be reduced. That is, a machine, say machine l, is taken out of set M_0 and a graph G'' is constructed by modifying graph G' through the inclusion of the disjunctive arcs that specify the sequence of oper-

ations on machine k and the exclusion of the disjunctive arcs associated with machine l. Machine l is resequenced by solving the corresponding $1 \mid r_j \mid L_{\max}$ problem with the release and due dates determined by the critical paths in graph G''. Resequencing each of the machines in the original set M_0 completes the iteration.

In the next iteration the entire procedure is repeated and another machine is added to the current set $M_0 \cup k$.

The shifting bottleneck heuristic can be summarized as follows.

Algorithm 7.2.1 (Shifting Bottleneck Heuristic)

Step 1. (Initial Conditions)

Set $M_0 = \emptyset$.
Graph G *is the graph with all the conjunctive arcs and no disjunctive arcs.*
Set $C_{\max}(M_0)$ *equal to the longest path in graph* G.

Step 2. (Analysis of machines still to be scheduled)

Do for each machine i *in set* $M - M_0$ *the following:*
generate an instance of $1 \mid r_j \mid L_{\max}$
(with the release date of operation (i, j) *determined by the longest path in graph* G *from the source node* U *to node* (i, j)*;*
and the due date of operation (i, j) *determined by* $C_{\max}(M_0)$ *minus the longest path in graph* G *from node* (i, j) *to the sink, plus* p_{ij}*).*
Minimize the $L_{\max}$ *in each one of these single machine subproblems.*
Let $L_{\max}(i)$ *denote the minimum* $L_{\max}$ *in the subproblem corresponding to machine* i.

Step 3. (Bottleneck selection and sequencing)

Let

$$L_{\max}(k) = \max_{i \in \{M - M_0\}} (L_{\max}(i))$$

Sequence machine k *according to the sequence obtained in Step 2 for that machine.*
Insert all the corresponding disjunctive arcs in graph G.
Insert machine k *in* M_0.

Step 4. (Resequencing of all machines scheduled earlier)

Do for each machine $i \in \{M_0 - k\}$ *the following:*
delete from G *the disjunctive arcs corresponding to machine* i*;*
formulate a single machine subproblem for machine i *with release dates and due dates of the operations determined by longest path calculations in* G.
Find the sequence that minimizes $L_{\max}(i)$ *and insert the corresponding disjunctive arcs in graph* G.

Step 5. (Stopping criterion)

If $M_0 = M$ *then STOP, otherwise go to Step 2.* ||

The structure of the shifting bottleneck heuristic shows the relationship between the bottleneck concept and more combinatorial concepts such as critical (longest) path and maximum lateness. A critical path indicates the location and the timing of a bottleneck. The maximum lateness gives an indication of the amount by which the makespan increases if a machine is added to the set of machines already scheduled.

The remainder of this section contains two examples that illustrate the use of the shifting bottleneck heuristic.

Example 7.2.2 (Application of Shifting Bottleneck Heuristic)

Consider the instance with four machines and three jobs described in Examples 7.1.1 and 7.1.4. The routes of the jobs, i.e., the machine sequences, and the processing times are given in the following table:

jobs	*machine sequence*	*processing times*
1	1,2,3	$p_{11} = 10, \quad p_{21} = 8, \quad p_{31} = 4$
2	2,1,4,3	$p_{22} = 8, \quad p_{12} = 3, \quad p_{42} = 5, \quad p_{32} = 6$
3	1,2,4	$p_{13} = 4, \quad p_{23} = 7, \quad p_{43} = 3$

Iteration 1: Initially, set M_0 is empty and graph G' contains only conjunctive arcs and no disjunctive arcs. The critical path and the makespan $C_{\max}(\emptyset)$ can be determined easily: this makespan is equal to the maximum total processing time required for any job. The maximum of 22 is achieved in this case by both jobs 1 and 2. To determine which machine to schedule first, each machine is considered as a $1 \mid r_j \mid L_{\max}$ problem with the release dates and due dates determined by the longest paths in G' (assuming a makespan of 22).

The data for the $1 \mid r_j \mid L_{\max}$ problem corresponding to machine 1 are presented in the following table.

jobs	1	2	3
p_{1j}	10	3	4
r_{1j}	0	8	0
d_{1j}	10	11	12

The optimal sequence turns out to be $1, 2, 3$ with $L_{\max}(1) = 5$.

The data for the subproblem regarding machine 2 are:

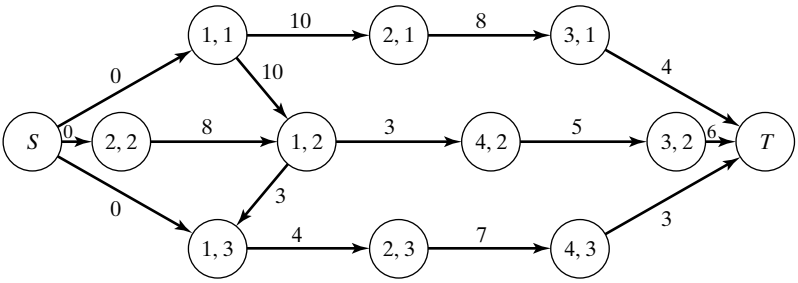


Fig. 7.6 Iteration 1 of shifting bottleneck heuristic (Example 7.2.2)

jobs	1	2	3
p_{2j}	8	8	7
r_{2j}	10	0	4
d_{2j}	18	8	19

The optimal sequence for this problem is $2, 3, 1$ with $L_{\max}(2) = 5$. Similarly, it can be shown that

$$L_{\max}(3) = 4$$

and

$$L_{\max}(4) = 0.$$

From this it follows that either machine 1 or machine 2 may be considered a bottleneck. Breaking the tie arbitrarily, machine 1 is selected to be included in M_0. The graph G'' is obtained by fixing the disjunctive arcs corresponding to the sequence of the jobs on machine 1 (see Figure 7.6). It is clear that

$$C_{\max}(\{1\}) = C_{\max}(\emptyset) + L_{\max}(1) = 22 + 5 = 27.$$

Iteration 2: Given that the makespan corresponding to G'' is 27, the critical paths in the graph can be determined. The three remaining machines have to be analyzed separately as $1 \mid r_j \mid L_{max}$ problems. The data for the instance concerning machine 2 are:

jobs	1	2	3
p_{2j}	8	8	7
r_{2j}	10	0	17
d_{2j}	23	10	24

The optimal schedule is $2, 1, 3$ and the resulting $L_{\max}(2) = 1$. The data for the instance corresponding to machine 3 are:

jobs	1	2
p_{3j}	4	6
r_{3j}	18	18
d_{3j}	27	27

Both sequences are optimal and $L_{\max}(3) = 1$. Machine 4 can be analyzed in the same way and the resulting $L_{\max}(4) = 0$. Again, there is a tie and machine 2 is selected to be included in M_0. So $M_0 = \{1, 2\}$ and

$$C_{\max}(\{1, 2\}) = C_{\max}(\{1\}) + L_{\max}(2) = 27 + 1 = 28.$$

The disjunctive arcs corresponding to the job sequence on machine 2 are added to G'' and graph G''' is obtained. At this point, still as a part of iteration 2, an attempt may be made to decrease $C_{\max}(\{1, 2\})$ by resequencing machine 1. It can be checked that resequencing machine 1 does not give any improvement.

Iteration 3: The critical path in G''' can be determined and machines 3 and 4 remain to be analyzed. These two problems turn out to be very simple with both having a zero maximum lateness. Neither one of the two machines constitutes a bottleneck in any way.

The final schedule is determined by the following job sequences on the four machines: job sequence $1, 2, 3$ on machine 1; job sequence $2, 1, 3$ on machine 2; job sequence $2, 1$ on machine 3 and job sequence $2, 3$ on machine 4. The makespan is 28. ||

The implementation of the shifting bottleneck technique in practice often tends to be more complicated than the heuristic described above. The solution procedure for the single machine subproblem must deal with some additional complications.

The single machine maximum lateness problem that has to be solved repeatedly within each iteration of the heuristic may at times be slightly different and more complicated than the $1 \mid r_j \mid L_{\max}$ problem described in Chapter 3 (which is also the problem used for determining the lower bounds in the previous section). In the single machine problem that has to be solved in the shifting bottleneck heuristic, the operations on a given machine may have to be subject to a special type of precedence constraints. It may be the case that an operation that has to be processed on a particular machine can only be processed on that machine after certain other operations have completed their processing on that machine. These precedence constraints may be imposed by the sequences

of the operations on the machines that already have been scheduled in earlier iterations.

It may even be the case that two operations that are subject to such constraints not only have to be processed in the given order, they may also have to be processed a given amount of time apart from one another. That is, in between the processing of two operations that are subject to these precedence constraints a certain minimum amount of time (i.e., a delay) may have to elapse.

The lengths of the delays are also determined by the sequences of the operations on the machines already scheduled. These precedence constraints are therefore referred to as *delayed* precedence constraints.

The next example illustrates the potential need for delayed precedence constraints in the single machine subproblem. Without these constraints the shifting bottleneck heuristic may end up in a situation where there is a cycle in the disjunctive graph and the corresponding schedule is infeasible. The following example illustrates how sequences on machines already scheduled (machines in M_0) impose constraints on machines still to be scheduled (machines in $M - M_0$).

Example 7.2.3 (Delayed Precedence Constraints)

Consider the following instance.

jobs	*machine sequence*	*processing times*
1	1,2	$p_{11} = 1, \quad p_{21} = 1$
2	2,1	$p_{22} = 1, \quad p_{12} = 1$
3	3	$p_{33} = 4$
4	3	$p_{34} = 4$

Applying the shifting bottleneck heuristic results in the following three iterations.

Iteration 1: The first iteration consists of the optimization of three subproblems. The data for the three subproblems associated with machines 1, 2, and 3 are tabulated below.

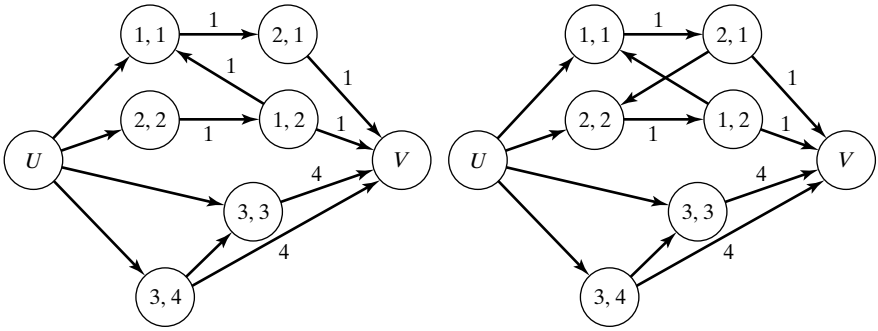
jobs	1	2
p_{1j}	1	1
r_{1j}	0	1
d_{1j}	3	4

jobs	1	2
p_{2j}	1	1
r_{2j}	1	0
d_{2j}	4	3

jobs	1	2
p_{3j}	4	4
r_{3j}	0	0
d_{3j}	4	4

The optimal solutions for machines 1 and 2 have $L_{\max} \leq 0$, while that for machine 3 has $L_{\max} = 4$. So machine 3 is scheduled first and arc $(3, 4) \rightarrow (3, 3)$ is inserted.

Iteration 2: The new set of subproblems are associated with machines 1 and 2.



(a) Second iteration (b) Final graph

Fig. 7.7 Application of shifting bottleneck heuristic in Example 7.2.3

jobs	1	2
p_{1j}	1	1
r_{1j}	0	1
d_{1j}	7	8

jobs	1	2
p_{2j}	1	1
r_{2j}	1	0
d_{2j}	8	7

The optimal solutions for machines 1 and 2 both have $L_{\max} = -6$, so we arbitrarily select machine 1 to be scheduled next. Arc $(1,2) \rightarrow (1,1)$ is inserted (see Figure 7.7.a).

Iteration 3: One subproblem remains, and it is associated with machine 2.

jobs	1	2
p_{2j}	1	1
r_{2j}	3	0
d_{2j}	8	5

Any schedule for machine 2 yields an $L_{\max} \leq 0$. If a schedule would be selected arbitrarily and arc $(2,1) \rightarrow (2,2)$ would be inserted, then a cycle is created in the graph, and the overall schedule is infeasible (see Figure 7.7.b).

This situation could have been prevented by imposing delayed precedence constraints. After scheduling machine 1 (in iteration 2) there is a path from (2,2) to (2,1) with length 3. After iteration 2 has been completed a delayed precedence constraint can be generated for subsequent iterations. Operation (2,2) must precede operation (2,1) and, furthermore, there must be a delay of 2 time units in between the completion of operation (2,2) and the start of operation (2,1). With this constraint iteration 3 generates a sequence for machine 2 that results in a feasible schedule. ||

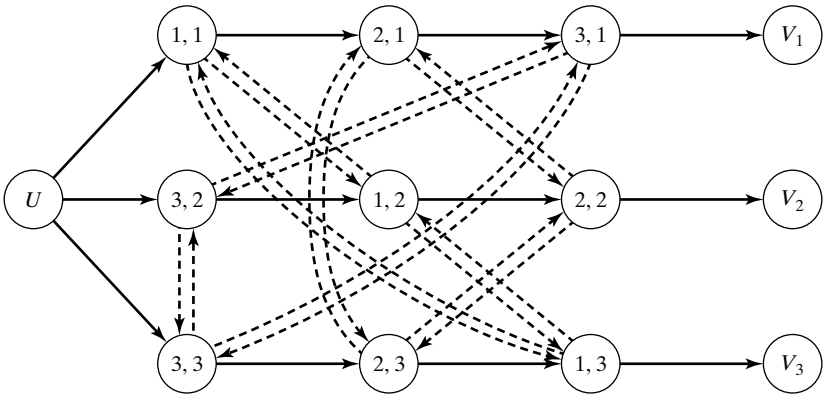


Fig. 7.8 Directed graph for job shop with total weighted tardiness objective

Extensive numerical research has shown that the Shifting Bottleneck heuristic is extremely effective. When applied to a standard test problem with 10 machines and 10 jobs that had remained unsolved for more than 20 years, the heuristic obtained a very good solution very fast. This solution turned out to be optimal after a branch-and-bound procedure found the same result and verified its optimality. The branch-and-bound approach, in contrast to the heuristic, needed many hours of CPU time. The disadvantage of the heuristic is, of course, that there is no guarantee that the solution it reaches is optimal.

The Shifting Bottleneck heuristic can be adapted in order to be applied to more general models than the job shop model considered above, i.e., it can be applied also to flexible job shops with recirculation.

7.3 The Shifting Bottleneck Heuristic and the Total Weighted Tardiness

This section describes an approach for $Jm \mid\mid \sum w_j T_j$ that combines a variant of the shifting bottleneck heuristic discussed in the previous section with a priority rule called the Apparent Tardiness Cost first (ATC) rule.

The disjunctive graph representation for $Jm \mid\mid \sum w_j T_j$ is different from that for $Jm \mid\mid C_{\max}$. In the makespan problem only the completion time of the last job to leave the system is of importance. There is therefore a single sink in the disjunctive graph. In the total weighted tardiness problem the completion times of all n jobs are of importance. Instead of a single sink, there are now n sinks, i.e., $V_1, \ldots, V_n$ (see Figure 7.8). The length of the longest path from the source U to the sink V_k represents the completion time of job k.

The approach can be described as follows. Machines are again scheduled one at a time. At the start of a given iteration all machines in set M_0 have already

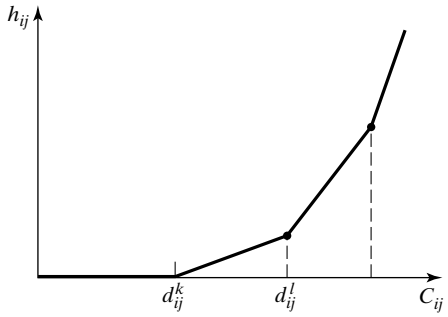


Fig. 7.9 Cost function h_{ij} of operation (i, j) in single machine subproblem

been scheduled (i.e., all disjunctive arcs have been selected for these machines) and in this iteration it has to be decided which machine should be scheduled next and how it should be scheduled. Each of the remaining machines has to be analyzed separately and for each of these machines a measure of criticality has to be computed. The steps to be done within an iteration can be described as follows.

In the disjunctive graph representation all disjunctive arcs belonging to the machines still to be scheduled are deleted and all disjunctive arcs selected for the machines already scheduled (set M_0) are kept in place. Given this directed graph, the completion times of all n jobs can be computed easily. Let C'_k denote the completion time of job k. Now consider a machine i that still has to be scheduled (machine i is an element of set $M - M_0$). To avoid an increase in the completion time C'_k, operation (i, j), $j = 1, \ldots, n$, must be completed on machine i by some local due date d_{ij}^k. This local due date can be computed by considering the longest path from operation (i, j) to the sink corresponding to job k, i.e., V_k. If there is no path from node (i, j) to sink V_k, then the local due date d_{ij}^k is infinity. So, because of job k, there may be a local due date d_{ij}^k for operation (i, j). That is, if operation (i, j) is completed after d_{ij}^k then job k's overall completion time is postponed, resulting in a penalty. If the completion of job k, C'_k, is already past the due date d_k of job k, any increase in the completion time increases the penalty at a rate w_k. Because operation (i, j) may cause a delay in the completion of any one of the n jobs, one can assume that operation (i, j) is subject to n local due dates. This implies that operation (i, j) is subject to a piece-wise linear cost function h_{ij} (see Figure 7.9).

Thus a measure of criticality of machine i can be obtained by solving a single machine problem with each operation subject to a piece-wise linear cost function, i.e., $1 \mid\mid \sum h_j(C_j)$, where h_j is a piece-wise linear cost function corresponding to job j. As in the previous section, the operations may be subject to delayed precedence constraints to ensure feasibility.

This single machine subproblem is a generalization of the $1 \mid\mid \sum w_j T_j$ problem (see Chapter 3). A well-known priority rule for $1 \mid\mid \sum w_j T_j$ is the so-called *Apparent Tardiness Cost (ATC)* rule. This ATC heuristic is a composite dispatching rule that combines the WSPT rule and the so-called *Minimum Slack first (MS)* rule (under the MS rule the slack of job j at time t, $\max(d_j - p_j - t, 0)$, is computed and the job with the minimum slack is scheduled). Under the ATC rule jobs are scheduled one at a time; that is, every time the machine becomes free a ranking index is computed for each remaining job. The job with the highest ranking index is then selected to be processed next. This ranking index is a function of the time t at which the machine became free as well as of the p_j, the w_j and the d_j of the remaining jobs. The index is defined as

$$I_j(t) = \frac{w_j}{p_j} \exp\Big(- \frac{\max(d_j - p_j - t, 0)}{K\bar{p}} \Big),$$

where K is a scaling parameter, that can be determined empirically, and $\bar{p}$ is the average of the processing times of the remaining jobs. The ATC rule is discussed in detail in Chapter 14.

The piece-wise linear and convex function h_{ij} in the subproblem $1 \mid\mid \sum h_j(C_j)$ may be regarded as a sum of linear penalty functions, for each of which an ATC priority index can be computed. One can think of several composite priority index functions for this more complicated cost function. A reasonably effective one assigns to operation (i, j) the priority value

$$I_{ij}(t) = \sum_{k=1}^{n} \frac{w_k}{p_{ij}} \exp\Big(- \frac{(d_{ij}^k - p_{ij} + (r_{ij} - t))^+}{K\bar{p}} \Big),$$

where t is the earliest time at which machine i can be used, K is a scaling parameter and $\bar{p}$ is the integer part of the average length of the operations to be processed on machine i. This composite dispatching rule yields a reasonably good schedule for machine i.

A measure for the criticality of machine i can now be computed in a number of ways. For example, consider the solutions of all the single machine subproblems and set the measure for the criticality of a machine equal to the corresponding value of the objective function. However, there are more involved and more effective methods for measuring machine criticality. For example, by selecting the disjunctive arcs implied by the schedule for machine i, one can easily compute in the new disjunctive graph the new (overall) completion times of all n jobs, say C_k''. Clearly, $C_k'' \geq C_k'$. The contribution of job k to the measure of criticality of machine i is computed as follows. If $C_k' > d_k$, then the contribution of job k to the measure of criticality of machine i is $w_k(C_k'' - C_k')$. However, if $C_k' < d_k$, then the penalty due to an increase of the completion of job k is more difficult to estimate. This penalty would then be a function of C_k', C_k'', and d_k. Several functions have been experimented with and appear to be promising. One such function is

$$w_k(C_k'' - C_k') \exp\Big(- \frac{(d_k - C_k'')^+}{K} \Big),$$

where K is a scaling parameter. Summing over all jobs, i.e.,

$$\sum_{k=1}^{n} w_k(C_k'' - C_k') \exp\Big(- \frac{(d_k - C_k'')^+}{K} \Big),$$

provides a measure of criticality for machine i. This last expression plays a role that is similar to the one of $L_{\max}(i)$ in Step 2 of Algorithm 7.2.1. After the criticality measures of all the machines in $M - M_0$ have been computed, the machine with the highest measure is selected as the next one to be included in set M_0.

However, this process does not yet complete an iteration. The original shifting bottleneck approach, as described in Algorithm 7.2.1, suggests that rescheduling all the machines in the original set M_0 is advantageous. This rescheduling may result in different and better schedules. After this step has been completed, the entire process repeats itself and the next iteration is started.

Example 7.3.1 (Shifting Bottleneck and Total Weighted Tardiness)
Consider the instance with three machines and three jobs depicted in Figure 7.8.

job	w_j	r_j	d_j	*machine sequence*	*processing times*
1	1	5	24	1,2,3	$p_{11} = 5$, $p_{21} = 10$, $p_{31} = 4$
2	2	0	18	3,1,2	$p_{32} = 4$, $p_{12} = 5$, $p_{22} = 6$
3	2	0	16	3,2,1	$p_{33} = 5$, $p_{23} = 3$, $p_{13} = 7$

The initial graph is depicted in Figure 7.10.a.
Iteration 1: The first iteration requires the optimization of three subproblems, one for each machine in the job shop. The data for these three subproblems, corresponding to machines 1, 2, and 3, are tabulated below.

jobs	1	2	3
p_{1j}	5	5	7
r_{1j}	5	4	8
$d_{1j}^1, d_{1j}^2, d_{1j}^3$	$10, -, -$	$-, 12, -$	$-, -, 16$

jobs	1	2	3
p_{2j}	10	6	3
r_{2j}	10	9	5
$d_{2j}^1, d_{2j}^2, d_{2j}^3$	$20, -, -$	$-, 18, -$	$-, -, 9$

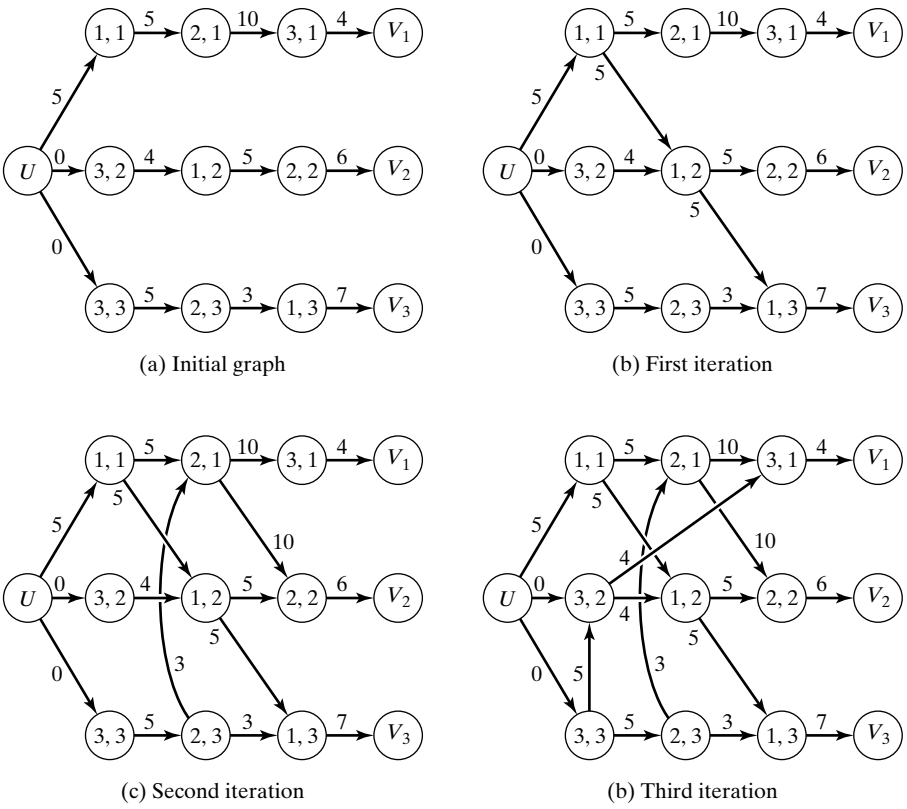


Fig. 7.10 Directed graphs in Example 7.3.1

jobs	1	2	3
p_{3j}	4	4	5
r_{3j}	20	0	0
$d^1_{3j}, d^2_{3j}, d^3_{3j}$	$24, -, -$	$-, 7, -$	$-, -, 6$

The entry "$-$" indicates that the corresponding due date d^k_{ij} is infinite, i.e., there is no path from operation (i, j) to the sink corresponding to job k. The subproblems are solved using a dispatching rule that is based on the priority index $I_{ij}(t)$ for operation (i, j), where t is the earliest time at which machine i can be used. Set the scaling parameter K equal to 0.1.

Since no operation has been scheduled yet, the priority indexes for the operations assigned to machine 1 are (assuming $t = 4$ and $\bar{p} = 5$) $I_{11}(4) = 1.23 \times 10^{-6}$, $I_{12}(4) = 3.3 \times 10^{-7}$ and $I_{13}(4) = 1.46 \times 10^{-12}$. The operation with the highest priority, i.e., operation $(1, 1)$, is put in the first position and the remaining indexes are recalculated in order to determine which operation

should be scheduled next. The solutions obtained for these three subproblems are:

machine i	*sequence*	*value*
1	(1,1),(1,2),(1,3)	18
2	(2,3),(2,1),(2,2)	16
3	(3,3),(3,2),(3,1)	4

Since the solution of subproblem 1 has the highest value, schedule machine 1 by inserting the disjunctive arcs $(1,1) \rightarrow (1,2)$ and $(1,2) \rightarrow (1,3)$, as shown in Figure 7.10.b.

Iteration 2: The data for the new subproblems, corresponding to machines 2 and 3, are tabulated below.

jobs	1	2	3
p_{2j}	10	6	3
r_{2j}	10	15	5
$d^1_{2j}, d^2_{2j}, d^3_{2j}$	$20, -, -$	$-, 21, -$	$-, -, 15$

jobs	1	2	3
p_{3j}	4	4	5
r_{3j}	20	0	0
$d^1_{3j}, d^2_{3j}, d^3_{3j}$	$24, -, -$	$-, 10, 10$	$-, -, 12$

In this iteration operation $(3,2)$ has two due dates because there is a (directed) path from node $(3,2)$ to V_2 and V_3. This makes its index equal to

$$I_{32}(0) = 1.53 \times 10^{-7} + 1.53 \times 10^{-7} = 3.06 \times 10^{-7},$$

since $t = 0$ and $\bar{p} = 4$. The solutions obtained for the two subproblems are:

machine i	*sequence*	*value*
2	(2,3),(2,1),(2,2)	10
3	(3,2),(3,3),(3,1)	0

The solution for subproblem 2 has the highest value (10). Schedule machine 2 by inserting the disjunctive arcs $(2,3) \rightarrow (2,1)$ and $(2,1) \rightarrow (2,2)$ as shown in Figure 7.10.c.

Iteration 3: The only subproblem that remains is the one for machine 3.

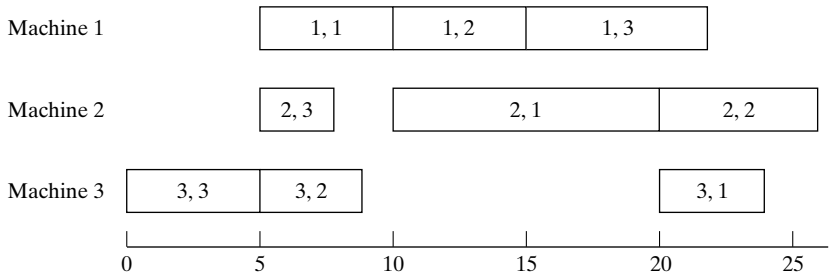


Fig. 7.11 Final schedule in Example 7.3.1

jobs	1	2	3
p_{3j}	4	4	5
r_{3j}	20	0	0
$d^1_{3j}, d^2_{3j}, d^3_{3j}$	$24, -, -$	$-, 15, 10$	$7, 7, 12$

Its optimal solution is $(3,3),(3,2),(3,1)$ with value equal to zero, so insert the arcs $(3,3) \rightarrow (3,2)$ and $(3,2) \rightarrow (3,1)$, as shown in Figure 7.10.d. The final solution is depicted in Figure 7.11, with objective function equal to

$$\sum_{j=1}^{3} w_j T_j = 1 \times (24-24)^+ + 2 \times (26-18)^+ + 2 \times (22-16)^+ = 28.$$

It happens that in this case the heuristic does not yield an optimal solution. The optimal solution with a value of 18 can be obtained with more elaborate versions of this heuristic. These versions make use of backtracking techniques as well as machine reoptimization (similar to Step 4 in Algorithm 7.2.1). ||

7.4 Constraint Programming and the Makespan

Constraint programming is a technique that originated in the Artificial Intelligence (AI) community. In recent years, it has often been implemented in combination with Operations Research (OR) techniques in order to improve its effectiveness.

Constraint programming, according to its original design, only tries to find a good solution that is feasible and that satisfies all the given constraints (which may include different release dates and due dates of jobs). The solutions obtained may not necessarily minimize the objective function. However, it is possible to embed a constraint programming technique in a framework that is designed to minimize any due date related objective function.

Constraint programming applied to $Jm \mid\mid C_{\max}$ works as follows. Suppose that in a job shop a schedule has to be found with a makespan $C_{\max}$ that is

less than or equal to a given deadline $\bar{d}$. The constraint satisfaction algorithm has to produce for each machine a sequence of operations such that the overall schedule has a makespan less than or equal to $\bar{d}$.

Before the actual procedure starts, an initialization step has to be done. For each operation a computation is done to determine its earliest possible starting time and latest possible completion time on the machine in question. After all the time windows have been computed, the time windows of all the operations on each machine are compared to one another. When the time windows of two operations on any given machine do not overlap, a precedence relationship between the two operations can be imposed; in any feasible schedule the operation with the earlier time window must precede the operation with the later time window. Actually, a precedence relationship may be inferred even when the time windows do overlap. Let S'_{ij} (S''_{ij}) denote the earliest (latest) possible starting time of operation (i, j) and C'_{ij} (C''_{ij}) the earliest (latest) possible completion time of operation (i, j) under the current set of precedence constraints. Note that the earliest possible starting time of operation (i, j), i.e., S'_{ij}, may be regarded as a local release date of the operation and may be denoted by r_{ij}, whereas the latest possible completion time, i.e., C''_{ij}, may be considered a local due date denoted by d_{ij}. Define the slack between the processing of operations (i, j) and (i, k) on machine i as

$$\sigma_{(i,j)\rightarrow(i,k)} = S''_{ik} - C'_{ij}$$

or

$$\sigma_{(i,j)\rightarrow(i,k)} = C''_{ik} - S'_{ij} - p_{ij} - p_{ik}$$

or

$$\sigma_{(i,j)\rightarrow(i,k)} = d_{ik} - r_{ij} - p_{ij} - p_{ik}.$$

If

$$\sigma_{(i,j)\rightarrow(i,k)} < 0$$

then there does not exist, under the current set of precedence constraints, a feasible schedule in which operation (i, j) precedes operation (i, k) on machine i; so a precedence relationship can be imposed that requires operation (i, k) to appear before operation (i, j). In the initialization step of the procedure all pairs of time windows are compared to one another and all implied precedence relationships are inserted in the disjunctive graph. Because of these additional precedence constraints the time windows of each one of the operations can be adjusted (narrowed) again, i.e., this involves a recomputation of the release date and the due date of each operation.

Constraint satisfaction techniques in general rely on constraint propagation. A constraint satisfaction technique typically attempts, in each step, to insert new precedence constraints (disjunctive arcs) that are implied by the precedence constraints inserted before and by the original constraints of the problem. With the new precedence constraints in place the technique recomputes the time windows of all operations. For each pair of operations that have to be processed

on the same machine it has to be verified which one of the following four cases applies.

Case 1:
If $\sigma_{(i,j)\rightarrow(i,k)} \geq 0$ *and* $\sigma_{(i,k)\rightarrow(i,j)} < 0$,
then the precedence constraint $(i,j) \rightarrow (i,k)$ *has to be imposed.*

Case 2:
If $\sigma_{(i,k)\rightarrow(i,j)} \geq 0$ *and* $\sigma_{(i,j)\rightarrow(i,k)} < 0$,
then the precedence constraint $(i,k) \rightarrow (i,j)$ *has to be imposed.*

Case 3:
If $\sigma_{(i,j)\rightarrow(i,k)} < 0$ *and* $\sigma_{(i,k)\rightarrow(i,j)} < 0$,
then there is no schedule that satisfies the precedence constraints already in place.

Case 4:
If $\sigma_{(i,j)\rightarrow(i,k)} \geq 0$ *and* $\sigma_{(i,k)\rightarrow(i,j)} \geq 0$,
then either ordering between the two operations is still possible.

In one of the steps of the algorithm that is described in this section a pair of operations has to be selected that satisfies Case 4, i.e., either ordering between the operations is still possible. In this step of the algorithm many pairs of operations may still satisfy Case 4. If there is more than one pair of operations that satisfies Case 4, then a search control heuristic has to be applied. The selection of a pair is based on the sequencing flexibility that this pair still provides. The pair with the lowest flexibility is selected. The reasoning behind this approach is straightforward. If a pair with low flexibility is not scheduled early on in the process, then it may be the case that later on in the process this pair cannot be scheduled at all. So it makes sense to give priority to those pairs with a low flexibility and postpone pairs with a high flexibility. Clearly, the flexibility depends on the amounts of slack under the two orderings. One simple estimate of the sequencing flexibility of a pair of operations, $\phi((i,j)(i,k))$, is the minimum of the two slacks, i.e.,

$$\phi((i,j)(i,k)) = \min(\sigma_{(i,j)\rightarrow(i,k)}, \sigma_{(i,k)\rightarrow(i,j)}).$$

However, relying on this minimum may lead to problems. For example, suppose one pair of operations has slack values 3 and 100, whereas another pair has slack values 4 and 4. In this case, there may be only limited possibilities for scheduling the second pair and postponing a decision with regard to the second pair may well eliminate them. A feasible ordering with regard to the first pair may not really be in jeopardy. Instead of using $\phi((i,j)(i,k))$ the following measure of sequencing flexibility has proven to be more effective:

$$\phi'((i,j)(i,k)) = \sqrt{\min(\sigma_{(i,j)\rightarrow(i,k)}, \sigma_{(i,k)\rightarrow(i,j)}) \times \max(\sigma_{(i,j)\rightarrow(i,k)}, \sigma_{(i,k)\rightarrow(i,j)})}.$$

So if the max is large, then the flexibility of a pair of operations increases and the urgency to order the pair goes down. After the pair of operations with the lowest sequencing flexibility $\phi'((i,j)(i,k))$ has been selected, the precedence constraint that retains the most flexibility is imposed, i.e., if

$$\sigma_{(i,j)\rightarrow(i,k)} \geq \sigma_{(i,k)\rightarrow(i,j)}$$

operation (i,j) must precede operation (i,k).

In one of the steps of the algorithm it also can happen that a pair of operations satisfies Case 3. When this is the case the partial schedule that is under construction cannot be completed and the algorithm has to backtrack. Backtracking typically implies that one or more of the ordering decisions made in earlier iterations has to be annulled (i.e., precedence constraints that had been imposed earlier have to be removed). Or, it may imply that there does not exist a feasible solution for the problem in the way it has been presented and formulated and that some of the original constraints of the problem have to be relaxed.

The constraint guided heuristic search procedure can be summarized as follows.

Algorithm 7.4.1 (Constraint Guided Heuristic Search)

Step 1.

Compute for each unordered pair of operations $\sigma_{(i,j)\rightarrow(i,k)}$ *and* $\sigma_{(i,k)\rightarrow(i,j)}$.

Step 2.

Check dominance conditions and classify remaining ordering decisions.
If any ordering decision is either of Case 1 or Case 2 go to Step 3.
If any ordering decision is of Case 3, then backtrack;
otherwise go to Step 4.

Step 3.

Insert new precedence constraint and go to Step 1.

Step 4.

If no ordering decision is of Case 4, then solution is found. STOP.
Otherwise go to Step 5.

Step 5.

Compute $\phi'((i,j)(i,k))$ *for each pair of operations not yet ordered.*
Select the pair with the minimum $\phi'((i,j)(i,k))$.
If $\sigma_{(i,j)\rightarrow(i,k)} \geq \sigma_{(i,k)\rightarrow(i,j)}$, *then operation* (i,k) *must follow operation* (i,j)*;*
otherwise operation (i,j) *must follow operation* (i,k).
Go to Step 3. ||

In order to apply the constraint guided heuristic search procedure to $Jm \mid\mid C_{\max}$, it has to be embedded in the following framework. First, an upper bound d_u and a lower bound d_l have to be found for the makespan.

Algorithm 7.4.2 (Framework for Constraint Guided Heuristic Search)

Step 1.

Set $d = (d_l + d_u)/2$.
Apply Algorithm 7.4.1.

Step 2.

If $C_{\max} < d$, *set* $d_u = d$.
If $C_{\max} > d$, *set* $d_l = d$.

Step 3.

If $d_u - d_l > 1$ *return to Step 1.*
Otherwise STOP. ||

The following example illustrates the use of the constraint satisfaction technique.

Example 7.4.3 (Application of Constraint Programming to the Job Shop)

Consider the instance of the job shop problem described in Example 7.1.1.

jobs	*machine sequence*	*processing times*
1	1, 2, 3	$p_{11} = 10, \quad p_{21} = 8, \quad p_{31} = 4$
2	2, 1, 4, 3	$p_{22} = 8, \quad p_{12} = 3, \quad p_{42} = 5, \quad p_{32} = 6$
3	1, 2, 4	$p_{13} = 4, \quad p_{23} = 7, \quad p_{43} = 3$

Consider a due date $d = 32$ by when all jobs have to be completed. Consider again the disjunctive graph but disregard all disjunctive arcs (see Figure 7.12). By doing all longest path computations, the local release dates and local due dates for all operations can be established (see Table 7.1(a)).

Given these time windows for all the operations, it has to be verified whether these constraints already imply any additional precedence constraints. Consider, for example, the pair of operations (2,2) and (2,3) which both have to go on machine 2. Computing the slack yields

$$\begin{aligned} \sigma_{(2,3)\rightarrow(2,2)} &= d_{22} - r_{23} - p_{22} - p_{23} \\ &= 18 - 4 - 8 - 7 \\ &= -1, \end{aligned}$$

which implies that the ordering $(2,3) \rightarrow (2,2)$ is not feasible. So the disjunctive arc $(2,2) \rightarrow (2,3)$ has to be inserted. In the same way, it can be

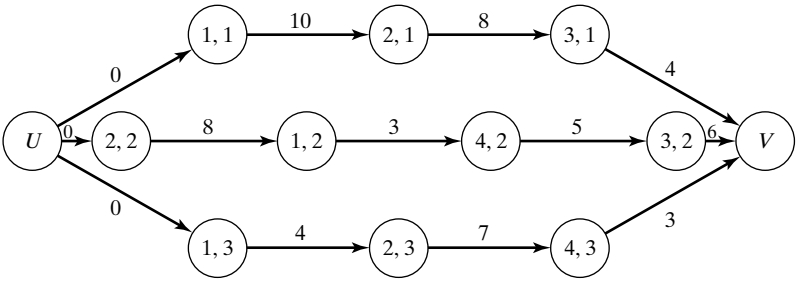


Fig. 7.12 Disjunctive graph without disjunctive arcs

(a)

operations	r_{ij}	d_{ij}
(1,1)	0	20
(2,1)	10	28
(3,1)	18	32
(2,2)	0	18
(1,2)	8	21
(4,2)	11	26
(3,2)	16	32
(1,3)	0	22
(2,3)	4	29
(4,3)	11	32

(b)

operations	r_{ij}	d_{ij}
(1,1)	0	18
(2,1)	10	28
(3,1)	18	32
(2,2)	0	18
(1,2)	10	21
(4,2)	13	26
(3,2)	18	32
(1,3)	0	22
(2,3)	8	29
(4,3)	15	32

(c)

pair	$\phi'((i,j)(i,k))$
(1,1)(1,3)	$\sqrt{4 \times 8} = 5.65$
(1,2)(1,3)	$\sqrt{5 \times 14} = 8.36$
(2,1)(2,3)	$\sqrt{4 \times 5} = 4.47$
(3,1)(3,2)	$\sqrt{4 \times 4} = 4.00$
(4,2)(4,3)	$\sqrt{3 \times 11} = 5.74$

Table 7.1 (a) Local release and due dates. (b) Local release and due dates after update. (c) Computing $\phi'((i,j)(i,k))$.

shown that the disjunctive arcs $(2,2) \rightarrow (2,1)$ and $(1,1) \rightarrow (1,2)$ have to be inserted as well.

These additional precedence constraints require an update of the release dates and due dates of all operations. The adjusted release and due dates are presented in Table 7.1(b).

These updated release and due dates do not imply any additional precedence constraints. Going through Step 5 of the algorithm requires the computation of the factor $\phi'((i,j)(i,k))$ for every unordered pair of operations on each machine (see Table 7.1(c)).

The pair with the least flexibility is $(3,1)(3,2)$. Since the slacks are such that

$$\sigma_{(3,2)\rightarrow(3,1)} = \sigma_{(3,1)\rightarrow(3,2)} = 4,$$

either precedence constraint can be inserted. Suppose the precedence constraint $(3,2) \rightarrow (3,1)$ is inserted. This precedence constraint causes major changes in the time windows during which the operations have to be processed (see Table 7.2(a)).

(a)

operations	r_{ij}	d_{ij}
(1,1)	0	14
(2,1)	10	28
(3,1)	24	32
(2,2)	0	14
(1,2)	10	17
(4,2)	13	22
(3,2)	18	28
(1,3)	0	22
(2,3)	8	29
(4,3)	15	32

(b)

operations	r_{ij}	d_{ij}
(1,1)	0	14
(2,1)	10	28
(3,1)	24	32
(2,2)	0	14
(1,2)	10	17
(4,2)	13	22
(3,2)	18	28
(1,3)	0	22
(2,3)	8	29
(4,3)	18	32

(c)

pair	$\phi'((i,j)(i,k))$
(1,1)(1,3)	$\sqrt{0 \times 8} = 0.00$
(1,2)(1,3)	$\sqrt{5 \times 10} = 7.07$
(2,1)(2,3)	$\sqrt{4 \times 5} = 4.47$

Table 7.2 (a) Local release and due dates. (b) Local release and due dates after update. (c) Computing $\phi'((i,j)(i,k))$.

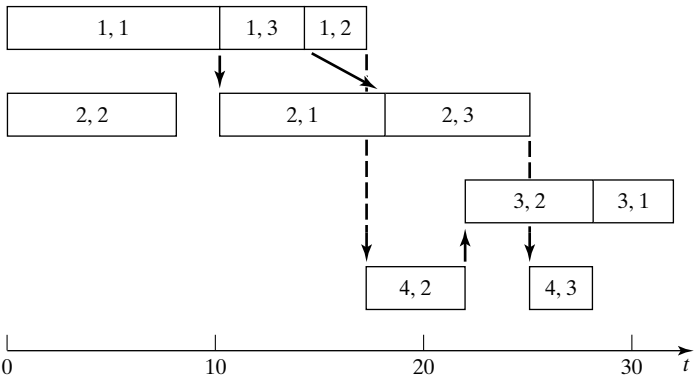


Fig. 7.13 Final schedule in Example 7.4.3

However, this new set of time windows imposes an additional precedence constraint, namely $(4,2) \rightarrow (4,3)$. This new precedence constraint causes changes in the release dates and due dates of the operations shown in Table 7.2(b).

These updated release and due dates do not imply additional precedence constraints. Step 5 of the algorithm now computes for every unordered pair of operations on each machine the factor $\phi'((i,j)(i,k))$ (see Table 7.2(c)).

The pair with the least flexibility is $(1,1)(1,3)$ and the precedence constraint $(1,1) \rightarrow (1,3)$ has to be inserted.

Inserting this last precedence constraint enforces one more constraint, namely $(2,1) \rightarrow (2,3)$. Now only one unordered pair of operations remains, namely pair $(1,3)(1,2)$. These two operations can be ordered in either way without violating any due dates. A feasible ordering is $(1,3) \rightarrow (1,2)$. The resulting schedule with a makespan of 32 is depicted in Figure 7.13. This schedule meets the due date originally set but is not optimal.

When the pair $(3,1)(3,2)$ had to be ordered the first time around, it could have been ordered in either direction because the two slack values were equal. Suppose at that point the opposite ordering was selected, i.e., $(3,1) \rightarrow (3,2)$. Restarting the process at that point yields the release and due dates shown in Table 7.3(a).

These release and due dates enforce a precedence constraint on the pair of operations $(2,1)(2,3)$ and the constraint is $(2,1) \rightarrow (2,3)$. This additional constraint changes the release dates and due dates (see Table 7.3(b)).

These new release dates and due dates have an effect on the pair $(4,2)(4,3)$ and the arc $(4,2) \rightarrow (4,3)$ has to be included. This additional arc does not cause any additional changes in the release and due dates. At this point only two pairs of operations remain unordered, namely the pair $(1,1)(1,3)$ and the pair $(1,2)(1,3)$ (see Table 7.3(c)).

(a)

operations	r_{ij}	d_{ij}
(1,1)	0	14
(2,1)	10	22
(3,1)	18	26
(2,2)	0	18
(1,2)	10	21
(4,2)	13	26
(3,2)	18	32
(1,3)	0	22
(2,3)	8	29
(4,3)	15	32

(b)

operations	r_{ij}	d_{ij}
(1,1)	0	14
(2,1)	10	22
(3,1)	18	26
(2,2)	0	18
(1,2)	10	21
(4,2)	13	26
(3,2)	22	32
(1,3)	0	22
(2,3)	18	29
(4,3)	25	32

(c)

pair	$\phi'((i,j)(i,k))$
(1,1)(1,3)	$\sqrt{0 \times 8} = 0.00$
(1,2)(1,3)	$\sqrt{5 \times 14} = 8.36$

Table 7.3 (a) Local release and due dates. (b) Local release and due dates after update. (c) Computing $\phi'((i,j)(i,k))$.

So the pair $(1,1)(1,3)$ is more critical and has to be ordered $(1,1) \rightarrow (1,3)$. It turns out that the last pair to be ordered, $(1,2)(1,3)$, can be ordered either way.

The resulting schedule turns out to be optimal and has a makespan of 28. ||

As stated before, constraint satisfaction is not only suitable for makespan minimization. It can also be applied to problems with due date related objectives and with each job having its own release date.

7.5 Discussion

The disjunctive graph formulation for $Jm \mid\mid C_{\max}$ extends to $Jm \mid rcrc \mid C_{\max}$. The set of disjunctive arcs for a machine may now not be a clique. If two operations of the same job have to be performed on the same machine a precedence relationship is given. These two operations are not connected by a pair of disjunctive arcs, since they are already connected by conjunctive arcs. The branch-and-bound approach described in Section 7.1 still applies. However,

the bounding mechanism is now not based on the solution of a $1 \mid r_j \mid L_{\max}$ problem, but rather on the solution of a $1 \mid r_j, prec \mid L_{\max}$ problem. The precedence constraints are the routing constraints on the different operations of the same job to be processed on the machine.

In the same way that a flow shop can be generalized into a flexible flow shop, a job shop can be generalized into a flexible job shop. The fact that the flexible flow shop allows for few structural results gives already an indication that it is hard to obtain results for the flexible job shop. Even the proportionate cases, i.e., $p_{ij} = p_j$ for all i, are hard to analyze.

The Shifting Bottleneck heuristic can be adapted in such a way that it can be applied to more general models than $Jm \mid\mid C_{\max}$. These more general models include recirculation as well as multiple machines at every stage. One such variation of the Shifting Bottleneck heuristic is based on decomposition principles. This variation is especially suitable for the scheduling of flexible job shops. The following five phase approach can be applied to flexible job shops.

Phase 1: The shop is divided into a number of workcenters that have to be scheduled. A workcenter may consist of a single machine or a bank of machines in parallel.

Phase 2: The entire job shop is represented through a disjunctive graph.

Phase 3: A performance measure is computed in order to rank the workcenters in order of criticality. The schedule of the most critical workcenter, among the workcenters of which the sequences still have to be determined, is fixed.

Phase 4: The disjunctive graph representation is used to capture the interactions between the workcenters already scheduled and those not yet scheduled.

Phase 5: Those workcenters that already have been sequenced are rescheduled using the new information obtained in Phase 4. If all workcenters have been scheduled, stop. Otherwise go to Phase 3.

The subproblem now becomes a nonpreemptive parallel machine scheduling problem with the jobs subject to different release dates and the maximum lateness as objective. A significant amount of computational research has been done on this parallel machine problem.

This chapter describes an application of constraint programming to minimize the makespan in job shops. A fair amount of research and development has been done in recent years with regard to constraint programming techniques. These techniques have now also been used for minimizing the total weighted tardiness in job shops.

This chapter has not shown the use of local search in job shop scheduling. An enormous amount of work has been done on this front. Chapter 14 discusses the applications of local search to job shops.

It is clear from this chapter that there are a number of completely different techniques for dealing with job shops, namely disjunctive programming, shifting bottleneck, constraint programming and also local search techniques.

Future research on job shop scheduling may focus on the development of hybrid techniques incorporating two or more of these techniques in a single framework that can be adapted easily to any given job shop instance.

Exercises (Computational)

7.1. Consider the following heuristic for $Jm \mid\mid C_{\max}$. Each time a machine is freed, select the job (among the ones immediately available for processing on the machine) with the largest *total* remaining processing (including the processing on the machine freed). If at any point in time more than one machine is freed, consider first the machine with the largest remaining workload. Apply this heuristic to the instance in Example 7.1.1.

7.2. Consider the following instance of $Jm \mid\mid C_{\max}$.

jobs	*machine sequence*	*processing times*
1	1,2,3	$p_{11} = 9, \quad p_{21} = 8, \quad p_{31} = 4$
2	1,2,4	$p_{12} = 5, \quad p_{22} = 6, \quad p_{42} = 3$
3	3,1,2	$p_{33} = 10, \quad p_{13} = 4, \quad p_{23} = 9$

Give the disjunctive programming formulation of this instance.

7.3. Consider the following instance of the $Jm \mid\mid C_{\max}$ problem.

jobs	*machine sequence*	*processing times*
1	1,2,3,4	$p_{11} = 9, \quad p_{21} = 8, \quad p_{31} = 4, \quad p_{41} = 4$
2	1,2,4,3	$p_{12} = 5, \quad p_{22} = 6, \quad p_{42} = 3, \quad p_{32} = 6$
3	3,1,2,4	$p_{33} = 10, \quad p_{13} = 4, \quad p_{23} = 9, \quad p_{43} = 2$

Give the disjunctive programming formulation of this instance.

7.4. Apply the heuristic described in Exercise 7.1 to the to the instance in Exercise 7.3.

7.5. Consider the instance in Exercise 7.2.

(a) Apply the Shifting Bottleneck heuristic to this instance (doing the computation by hand).

(b) Compare your result with the result of the shifting bottleneck routine in the LEKIN system.

7.6. Consider again the instance in Exercise 7.2.

(a) Apply the branch-and-bound algorithm to this instance of job shop problem.

(b) Compare your result with the result of the local search routine in the LEKIN system.

7.7. Consider the following instance of the two machine flow shop with the makespan as objective (i.e., an instance of $F2 \mid\mid C_{\max}$, which is a special case of $J2 \mid\mid C_{\max}$).

jobs	1	2	3	4	5	6	7	8	9	10	11
p_{1j}	3	6	4	3	4	2	7	5	5	6	12
p_{2j}	4	5	5	2	3	3	6	6	4	7	2

(a) Apply the heuristic described in Exercise 7.1 to this two machine flow shop.

(b) Apply the shifting bottleneck heuristic to this two machine flow shop.

(c) Construct a schedule using Johnson's rule (see Chapter 6).

(d) Compare the schedules found under (a), (b), and (c).

7.8. Consider the instance of the job shop with the total weighted tardiness objective described in Example 7.3.1. Apply the Shifting Bottleneck heuristic again, but now use as scaling parameter $K = 5$. Compare the resulting schedule with the schedule obtained in Example 7.3.1.

7.9. Consider the following instance of $Jm \mid\mid \sum w_j T_j$.

job	w_j	r_j	d_j	*machine sequence*	*processing times*
1	1	3	23	1,2,3	$p_{11} = 4$, $p_{21} = 9$, $p_{31} = 5$
2	2	2	17	3,1,2	$p_{32} = 4$, $p_{12} = 5$, $p_{22} = 5$
3	2	0	15	3,2,1	$p_{33} = 6$, $p_{23} = 4$, $p_{13} = 6$

(a) Apply the Shifting Bottleneck heuristic for the total weighted tardiness.

(b) Compare your result with the result of the shifting bottleneck routine in the LEKIN system.

(c) Compare your result with the result of the local search routine in the LEKIN system.

7.10. Consider the following instance of $F2 \mid\mid \sum w_j T_j$.

jobs	1	2	3	4	5
p_{1j}	12	4	6	8	2
p_{2j}	10	5	4	6	3
d_j	12	32	21	14	28
w_j	3	2	4	3	2

Apply the shifting bottleneck heuristic to minimize the total weighted tardiness.

Exercises (Theory)

7.11. Design a branching scheme for a branch-and-bound approach that is based on the insertion of disjunctive arcs. The root node of the tree corresponds to a disjunctive graph without any disjunctive arcs. Each node in the branching tree corresponds to a particular selection of a subset of the disjunctive arcs. That is, for any particular node in the tree a subset of the disjunctive arcs has been fixed in certain directions, while the remaining set of disjunctive arcs has not been fixed yet. From every node there are two arcs emanating to two nodes at the next level. One of the two nodes at the next level corresponds to an additional disjunctive arc being fixed in a given direction while the other node corresponds to the reverse arc being selected. Develop an algorithm that generates the nodes of such a branching tree and show that your algorithm generates every possible schedule.

7.12. Determine an upper and a lower bound for the makespan in an m machine job shop when preemptions are not allowed. The processing time of job j on machine i is p_{ij} (i.e., no restrictions on the processing times).

7.13. Show that when preemptions are allowed there always exists an optimal schedule for the job shop that is non-delay.

7.14. Consider $J2 \mid rcrc, p_{ij} = 1 \mid C_{\max}$. Each job has to be processed a number of times on each one of the two machines. A job always has to alternate between the two machines, i.e., after a job has completed one operation on one of the machines it has to go to the other machine for the next operation. The processing time of each operation is 1. Determine the schedule that minimizes the makespan and prove its optimality.

Comments and References

Job shop scheduling has received an enormous amount of attention in the research literature as well as in books.

The algorithm for minimizing the makespan in a two machine job shop without recirculation is due to Jackson (1956) and the disjunctive programming formulation described in Section 7.1 is from Roy and Sussmann (1964).

Branch-and-bound techniques have often been applied in order to minimize the makespan in job shops; see, for example, Lomnicki (1965), Brown and Lomnicki (1966), Barker and McMahon (1985), Carlier and Pinson (1989), Applegate and Cook (1991), Hoitomt, Luh and Pattipati (1993), Brucker, Jurisch and Sievers (1994) and Brucker, Jurisch and Krämer (1994). For an overview of branch-and-bound techniques applied to the job shop problem, see Pinson (1995). Some of the branching schemes of these branch-and-bound approaches are based on the generation of active schedules (the concept of an active schedule was first introduced by Giffler and Thompson (1960)), while other branching schemes are based on the directions of the disjunctive arcs to be selected.

The famous shifting bottleneck heuristic is due to Adams, Balas and Zawack (1988). Their algorithm makes use of a single machine scheduling algorithm developed by Carlier (1982). Earlier work on this particular single machine subproblem was done by McMahon and Florian (1975). Nowicki and Zdrzalka (1986), Dauzère-Pérès and Lasserre (1993, 1994) and Balas, Lenstra and Vazacopoulos (1995) all developed more refined versions of the Carlier algorithm. The monograph by Ovacik and Uzsoy (1997) presents an excellent treatise of the application of decomposition methods and shifting bottleneck techniques to large scale job shops with various objectives, e.g., the makespan and the maximum lateness. This monograph is based on a number of papers by the authors; see, for example, Uzsoy (1993) for the application of decomposition methods to flexible job shops.

Job shops with the total weighted tardiness as objective have been the focus of a number of studies. Vepsalainen and Morton (1987) developed heuristics based on priority rules. Singer and Pinedo (1998) developed a branch-and-bound approach and Pinedo and Singer (1999) developed the shifting bottleneck approach described in Section 7.3.

For some basic examples of constraint programming applications to job shops, see the books by Baptiste, Le Pape, and Nuijten (2001) and Van Hentenryck and Michel (2005). For an application of constraint programming for minimizing the total weighted tardiness, see Van Hentenryck and Michel (2004).

In addition to the procedures discussed in this chapter, job shop problems have also been tackled with local search procedures; see, for example, Matsuo, Suh, and Sullivan (1988), Dell'Amico and Trubian (1991), Della Croce, Tadei and Volta (1992), Storer, Wu and Vaccari (1992), Nowicki and Smutnicki (1996), and Kreipl (2000). Examples of such local search procedures are presented in Chapter 14.

For a broader view of the job shop scheduling problem, see Wein and Chevelier (1992). For an interesting special case of the job shop, i.e., a flow shop with reentry, see Graves, Meal, Stefek and Zeghmi (1983).

Chapter 8

Open Shops (Deterministic)

This chapter deals with multi-operation models that are different from the job shop models considered in the previous chapter. In a job shop each job has a fixed route that is predetermined. In practice, it often occurs that the route of the job is immaterial and up to the scheduler to decide. When the routes of the jobs are open, the model is referred to as an *open* shop.

The first section covers nonpreemptive open shop models with the makespan as objective. The second section deals with preemptive open shop models with the makespan as objective. The third and fourth section focus on nonpreemptive and preemptive models with the maximum lateness as objective. The fifth section considers nonpreemptive models with the number of tardy jobs as objective.

8.1 The Makespan without Preemptions

Consider $O2 \,||\, C_{\max}$; that is, there are two machines and n jobs. Job j may be processed first on machine 1 and then on machine 2 or vice versa; the decision-maker may determine the routes. The makespan has to be minimized. It is clear that

$$C_{\max} \geq \max\Big(\sum_{j=1}^{n} p_{1j}, \sum_{j=1}^{n} p_{2j}\Big),$$

since the makespan cannot be less than the workload on either machine. One would typically expect the makespan to be *equal* to the RHS of the inequality;

M.L. Pinedo, *Scheduling*, DOI: 10.1007/978-0-387-78935-4_8,

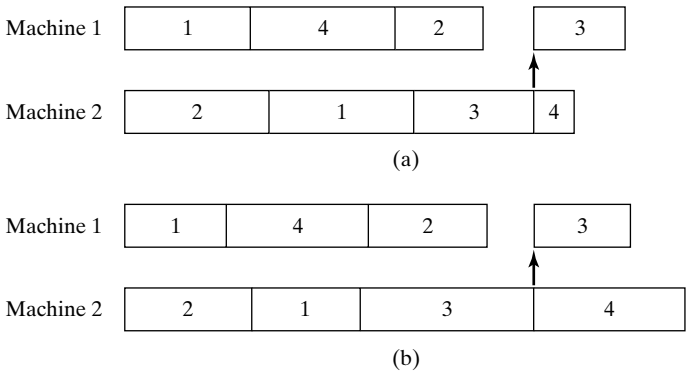


Fig. 8.1 Idle periods in two machine open shops: (a) idle period causes unnecessary increase in makespan (b) idle period does not cause an unnecessary increase in makespan

only in very special cases one would expect the makespan to be larger than the RHS. It is worthwhile to investigate the special cases where the makespan is strictly greater than the maximum of the two workloads.

This section considers only non-delay schedules. That is, if there is a job waiting for processing when a machine is free, then that machine is not allowed to remain idle. It immediately follows that an idle period can occur on a machine if and only if one job remains to be processed on that machine and, when that machine is available, this last job is just then being processed on the other machine. It can be shown that at most one such idle period can occur on at most one of the two machines (see Figure 8.1). Such an idle period *may* cause an unnecessary increase in the makespan; if this last job turns out to be the very last job to complete all its processing, then the idle period does cause an increase in the makespan (see Figure 8.1.a). If this last job, after having completed its processing on the machine that was idle, is *not* the very last job to leave the system, then the makespan is still equal to the maximum of the two workloads (see Figure 8.1.b).

Consider the following rule: whenever a machine is freed, start processing among the jobs that have not yet received processing on either machine the one with the longest processing time on the *other* machine. This rule is in what follows referred to as the *Longest Alternate Processing Time first (LAPT)* rule. At time zero, when both machines are idle, it may occur that the same job qualifies to be first on both machines. If that is the case, then it does not matter on which machine this job is processed first. According to this LAPT rule, whenever a machine is freed, jobs that already have completed their processing on the other machine have the lowest, that is, zero, priority on the machine just freed. There is therefore no distinction between the priorities of two jobs that both already have been processed on the other machine.

Theorem 8.1.1. *The LAPT rule results in an optimal schedule for* $O2 \mid\mid C_{\max}$ *with makespan*

$$C_{\max} = \max\Big(\max_{j\in\{1,\ldots,n\}} (p_{1j} + p_{2j}),\ \sum_{j=1}^{n} p_{1j}, \sum_{j=1}^{n} p_{2j} \Big).$$

Proof. Actually, a more general (and less restrictive) scheduling rule already guarantees a minimum makespan. This more general rule may result in many different schedules that are all optimal. This class of optimal schedules includes the LAPT schedule. This general rule also assumes that unforced idleness is not allowed.

Assume, without loss of generality, that the longest processing time among the $2n$ processing times belongs to operation $(1, k)$, that is,

$$p_{ij} \le p_{1k}, \qquad i = 1, 2, \ \ j = 1, \ldots, n.$$

The more general rule can be described as follows. If operation $(1, k)$ is the longest operation, then job k must be started at time 0 on machine 2. After job k has completed its processing on machine 2, its operation $(1, k)$ has the lowest possible priority with regard to processing on machine 1. Since its priority is then at all times lower than the priority of any other operation available for processing on machine 1, the processing of operation $(1, k)$ will be postponed as much as possible. It can only be processed on machine 1 if no other job is available for processing on machine 1 (this can happen either if it is the last operation to be done on machine 1 or if it is the second last operation and the last operation is not available because it is just then being processed on machine 2). The $2(n-1)$ operations of the remaining $n-1$ jobs can be processed on the two machines in any order; however, unforced idleness is not allowed.

That this rule results in a schedule with a minimum makespan can be shown as follows. If the resulting schedule has no idle period on either machine, then, of course, it is optimal. However, an idle period may occur either on machine 1 or on machine 2. So two cases have to be considered.

Case 1: Suppose an idle period occurs on machine 2. If this is the case, then only one more operation needs processing on machine 2 but this operation still has to complete its processing on machine 1. Assume this operation belongs to job l. When job l starts on machine 2, job k starts on machine 1 and $p_{1k} > p_{2l}$. So the makespan is determined by the completion of job k on machine 1 and no idle period has occurred on machine 1. So the schedule is optimal.

Case 2: Suppose an idle period occurs on machine 1. An idle period on machine 1 can occur only when machine 1 is freed after completing all its operations with the exception of operation $(1, k)$ and operation $(2, k)$ of job k is at that point still being processed on machine 2. In this case, the makespan is equal to $p_{2k} + p_{1k}$ and the schedule is optimal. □

Another rule that may seem appealing at first sight is the rule that gives, whenever a machine is freed, the highest priority to the job with the largest *total* remaining processing time on *both* machines. It turns out that there are instances, even with two machines, when this rule results in a schedule that is not optimal (see Exercise 8.12). The fact that the priority level of a job on one machine depends only on the amount of processing remaining to be done on the *other* machine is key.

The LAPT rule described above may be regarded as a special case of a more general rule that can be applied to open shops with more than two machines. This more general rule may be referred to as the *Longest Total Remaining Processing on Other Machines first* rule. According to this rule, again, the processing required on the machine currently available does not affect the priority level of a job. However, this rule does not always result in an optimal schedule since the $Om \mid\mid C_{\max}$ problem is NP-hard when $m \geq 3$.

Theorem 8.1.2. *The problem* $O3 \mid\mid C_{\max}$ *is NP-hard.*

Proof. The proof is based on a reduction of PARTITION to $O3 \mid\mid C_{\max}$. The PARTITION problem can be formulated as follows. Given positive integers $a_1, \ldots, a_t$ and

$$b = \frac{1}{2} \sum_{j=1}^{t} a_j,$$

do there exist 2 disjoint subsets S_1 and S_2 such that

$$\sum_{j \in S_1} a_j = \sum_{j \in S_2} a_j = b \, ?$$

The reduction is based on the following transformation. Consider $3t+1$ jobs. Of these $3t+1$ jobs there are $3t$ jobs that have only one nonzero operation and one job that has to be processed on each one of the three machines.

$$\begin{aligned}
&p_{1j} = a_j, \quad p_{2j} = p_{3j} = 0, && \text{for } 1 \leq j \leq t, \\
&p_{2j} = a_j, \quad p_{1j} = p_{3j} = 0, && \text{for } t+1 \leq j \leq 2t, \\
&p_{3j} = a_j, \quad p_{1j} = p_{2j} = 0, && \text{for } 2t+1 \leq j \leq 3t, \\
&p_{1,3t+1} = p_{2,3t+1} = p_{3,3t+1} = b,
\end{aligned}$$

where

$$\sum_{j=1}^{t} a_j = \sum_{j=t+1}^{2t} a_j = \sum_{j=2t+1}^{3t} a_j = 2b$$

and $z = 3b$. The open shop problem now has a schedule with a makespan equal to z if and only if there exists a partition. It is clear that to have a makespan equal to $3b$ job $3t+1$ has to be processed on the three machines without interruption. Consider the machine on which job $3t+1$ is processed second, that is, during the interval $(b, 2b)$. Without loss of generality it may

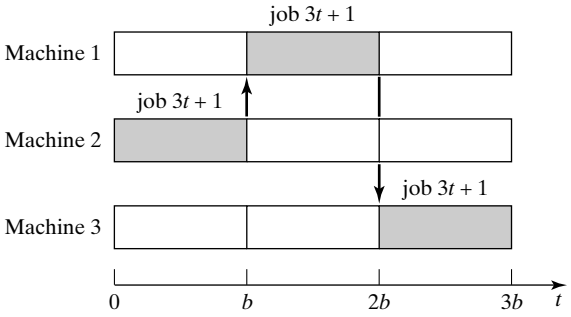


Fig. 8.2 Reduction of PARTITION to $O3 \mid\mid C_{\max}$

be assumed that this is machine 1. Jobs $1, \ldots, t$ have to be processed only on machine 1. If there exists a partition of these t jobs in such a way that one set can be processed during the interval $(0, b)$ and the other set can be processed during the interval $(2b, 3b)$, then the makespan is $3b$ (see Figure 8.2). If there does not exist such a partition, then the makespan has to be larger than $3b$. □

The LAPT rule for $O2 \mid\mid C_{\max}$ is one of the few polynomial time algorithms for nonpreemptive open shop problems. Most of the more general open shop models within the framework of Chapter 2 are NP-hard, for example, $O2 \mid r_j \mid C_{\max}$. However, the problem $Om \mid r_j, p_{ij} = 1 \mid C_{\max}$ can be solved in polynomial time. This problem is discussed in a more general setting in Section 8.3.

8.2 The Makespan with Preemptions

Preemptive open shop problems tend to be somewhat easier. In contrast to $Om \mid\mid C_{\max}$ the $Om \mid prmp \mid C_{\max}$ problem is solvable in polynomial time.

From the fact that the value of the makespan under LAPT is a lower bound for the makespan with two machines even when preemptions are allowed, it follows that the nonpreemptive LAPT rule is also optimal for $O2 \mid prmp \mid C_{\max}$.

It is easy to establish a lower bound for the makespan with m ($m \geq 3$) machines when preemptions are allowed:

$$C_{\max} \geq \max\Big(\max_{j \in \{1,\ldots,n\}} \sum_{i=1}^{m} p_{ij}, \max_{i \in \{1,\ldots,m\}} \sum_{j=1}^{n} p_{ij} \Big).$$

That is, the makespan is at least as large as the maximum workload on each of the m machines and at least as large as the total amount of processing to be done on each of the n jobs. It turns out that it is rather easy to obtain a schedule with a makespan that is equal to this lower bound.

In order to see how the algorithm works, consider the $m \times n$ matrix $\mathbf{P}$ of the processing times p_{ij}. Row i or column j is called *tight* if its sum is equal

to the lower bound and *slack* otherwise. Suppose it is possible to find in this matrix a subset of nonzero entries with exactly one entry in each tight row and one entry in each tight column and at most one entry in each slack row and slack column. Such a subset would be called a *decrementing* set. This subset is used to construct a partial schedule of length Δ, for some appropriately chosen Δ. In this partial schedule machine i works on job j for an amount of time that is equal to $\min(p_{ij}, \Delta)$ for each element p_{ij} in the decrementing set. In the original matrix $\mathbf{P}$ the entries corresponding to the decrementing set are then reduced to $\max(0, p_{ij} - \Delta)$ and the resulting matrix is then called $\mathbf{P'}$. If Δ is chosen appropriately, the makespan $C'_{\max}$ that corresponds to the new matrix $\mathbf{P'}$ is equal to $C_{\max} - \Delta$. This value for Δ has to be chosen carefully. First, it is clear that the Δ has to be smaller than every p_{ij} in the decrementing set that is in a tight row or column, otherwise there will be a row or column in $\mathbf{P'}$ that is strictly larger than $C'_{\max}$. For the same reason, if p_{ij} is an element in the decrementing set in a slack row, say row i, it is necessary that

$$\Delta \leq p_{ij} + C_{\max} - \sum_k p_{ik},$$

where $C_{\max} - \sum p_{ik}$ is the amount of slack time in row i. Similarly, if p_{ij} is an entry in the slack column j, then

$$\Delta \leq p_{ij} + C_{\max} - \sum_k p_{kj},$$

where $C_{\max} - \sum p_{kj}$ is the amount of slack time in column j. If row i or column j does not contain an element in the decrementing set, then

$$\Delta \leq C_{\max} - \sum_j p_{ij}$$

or

$$\Delta \leq C_{\max} - \sum_i p_{ij}.$$

If Δ is chosen to be as large as possible subject to these conditions, then either $\mathbf{P'}$ will contain at least one less strictly positive element than $\mathbf{P}$ or $\mathbf{P'}$ will contain at least one more tight row or column than $\mathbf{P}$. It is then clear that there cannot be more than $r + m + n$ iterations where r is the number of strictly positive elements in the original matrix.

It turns out that it is always possible to find a decrementing set for a nonnegative matrix $\mathbf{P}$. This property is the result of a basic theorem due to Birkhoff and von Neumann regarding stochastic matrices and permutation matrices. However, the proof of this theorem is beyond the scope of this book.

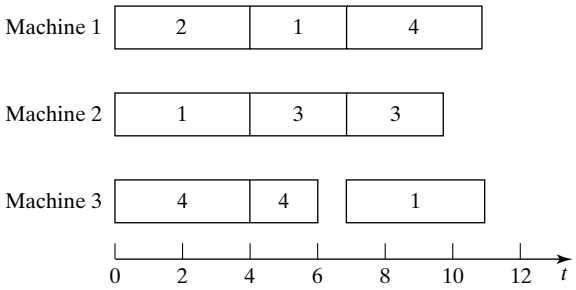


Fig. 8.3 Optimal Schedule for $O3 \mid prmp \mid C_{\max}$ with four jobs (Example 8.2.1)

Example 8.2.1 (Minimizing Makespan with Preemptions)
Consider 3 machines and 4 jobs with the processing times being the entries in the matrix

$$\mathbf{P} = \begin{bmatrix} 3\,4\,0\,4 \\ 4\,0\,6\,0 \\ 4\,0\,0\,6 \end{bmatrix}$$

It is easily verified that $C_{\max} = 11$ and that the first row and first column are tight. A possible decrementing set comprises the processing times $p_{12} = 4$, $p_{21} = 4$ and $p_{34} = 6$. If Δ is set equal to 4, then $C'_{\max} = 7$. A partial schedule is constructed by scheduling job 2 on machine 1 for 4 time units; job 1 on machine 2 for 4 time units and job 4 on machine 3 for 4 time units. The matrix is now

$$\mathbf{P}' = \begin{bmatrix} 3\,0\,0\,4 \\ 0\,0\,6\,0 \\ 4\,0\,0\,2 \end{bmatrix}$$

Again, the first row and the first column are tight. A decrementing set is obtained with the processing times $p_{11} = 3$, $p_{23} = 6$ and $p_{34} = 2$. Choosing $\Delta = 3$, the partial schedule can be augmented by assigning job 1 to machine 1 for 3 time units, job 3 to machine 2 for 3 time units and job 4 again to machine 3 but now only for 2 time units. The matrix is

$$\mathbf{P}'' = \begin{bmatrix} 0\,0\,0\,4 \\ 0\,0\,3\,0 \\ 4\,0\,0\,0 \end{bmatrix}$$

The last decrementing set is obtained with the remaining three positive processing times. The final schedule is obtained by augmenting the partial schedule by assigning job 4 on machine 1 for 4 time units, job 3 to machine 2 for 3 time units and job 1 to machine 3 for 4 time units (see Figure 8.3). ||

8.3 The Maximum Lateness without Preemptions

The $Om \mid\mid L_{\max}$ problem is a generalization of the $Om \mid\mid C_{\max}$ problem and is therefore at least as hard.

Theorem 8.3.1. *The problem $O2 \mid\mid L_{\max}$ is strongly NP-Hard.*

Proof. The proof is done by reducing 3-PARTITION to $O2 \mid\mid L_{\max}$. The 3-PARTITION problem is formulated as follows. Given positive integers $a_1, \ldots, a_{3t}$ and b, such that

$$\frac{b}{4} < a_j < \frac{b}{2}$$

and

$$\sum_{j=1}^{3t} a_j = tb,$$

do there exist t pairwise disjoint three element subsets $S_i \subset \{1, \ldots, 3t\}$ such that

$$\sum_{j \in S_i} a_j = b$$

for $i = 1, \ldots, t$?

The following instance of $O2 \mid\mid L_{\max}$ can be constructed. The number of jobs, n, is equal to $4t$ and

$$\begin{array}{llll} p_{1j} = 0 & p_{2j} = a_j & d_j = 3tb & j = 1, \ldots, 3t \\ p_{1j} = 0 & p_{2j} = 2b & d_j = 2b & j = 3t+1 \\ p_{1j} = 3b & p_{2j} = 2b & d_j = (3(j-3t)-1)b & j = 3t+2, \ldots, 4t \end{array}$$

There exists a schedule with $L_{\max} \leq 0$ if and only if jobs $1, \ldots, 3t$ can be divided into t groups, each containing 3 jobs and requiring b units of processing time on machine 2, i.e., if and only if 3-PARTITION has a solution. □

It can be shown that $O2 \mid\mid L_{\max}$ is equivalent to $O2 \mid r_j \mid C_{\max}$. Consider the $O2 \mid\mid L_{\max}$ problem with deadlines $\bar{d}_j$ rather than due dates d_j. Let

$$\bar{d}_{\max} = \max(\bar{d}_1, \ldots, \bar{d}_n).$$

Apply a time reversal to $O2 \mid\mid L_{\max}$. Finding a feasible schedule with $L_{\max} = 0$ is now equivalent to finding a schedule for $O2 \mid r_j \mid C_{\max}$ with

$$r_j = \bar{d}_{\max} - \bar{d}_j$$

and a makespan that is less than $\bar{d}_{\max}$. So the $O2 \mid r_j \mid C_{\max}$ problem is therefore also strongly NP-hard.

Consider now the special case $Om \mid r_j, p_{ij} = 1 \mid L_{\max}$. The fact that all processing times are equal to 1 makes the problem considerably easier. The polynomial time solution procedure consists of three phases, namely

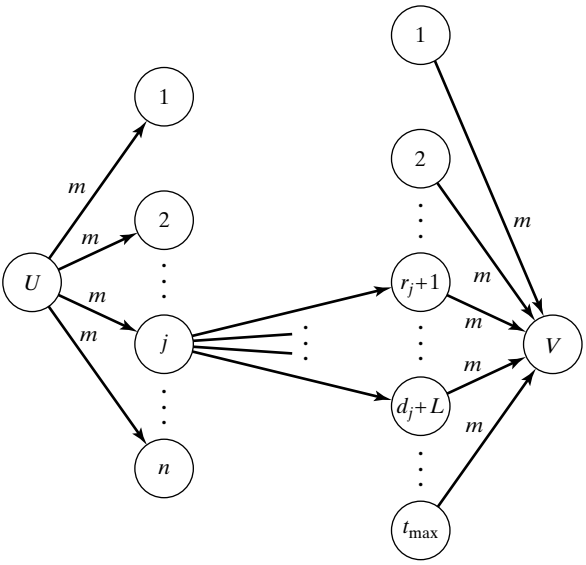


Fig. 8.4 Network flow problem of Phase 2

Phase 1: Parametrizing and a binary search.

Phase 2: Solving a network flow problem.

Phase 3: Coloring a bipartite graph.

The first phase of the procedure involves a parametrization. Let L be a free parameter and assume that each job has a deadline $d_j + L$. The objective is to find a schedule in which each job is completed before or at its deadline, ensuring that $L_{\max} \leq L$. Let

$$t_{\max} = \max(d_1, \ldots, d_n) + L,$$

that is, no job should receive any processing after time $t_{\max}$.

The second phase focuses on the following network flow problem: There is a source node U that has n arcs emanating to nodes $1, \ldots, n$. Node j corresponds to job j. The arc from the source node U to node j has capacity m (equal to the number of machines and to the number of operations of each job). There is a second set of $t_{\max}$ nodes, each node corresponding to one time unit. Node t, $t = 1, \ldots, t_{\max}$, corresponds to the time slot $[t-1, t]$. Node j has arcs emanating to nodes $r_j + 1, r_j + 2, \ldots, d_j + L$. Each one of these arcs has unit capacity. Each node of the set of $t_{\max}$ nodes has an arc with capacity m going to sink V (see Figure 8.4). The capacity limit on each one of these arcs is necessary to ensure that no more than m operations are processed in any given time period. The solution of this network flow problem indicates in which time slots the m operations of job j are to be processed.

However, the network flow solution cannot be translated immediately into a feasible schedule for the open shop, because in the network flow formulation no distinction is made between the different machines (i.e., in this solution it may be possible that two different operations of the same job are processed in two different time slots on the same machine). However, it turns out that the assignment of operations to time slots prescribed by the network flow solution can be transformed into a feasible schedule in such a way that each operation of job j is processed on a different machine.

The third phase of the algorithm generates a feasible schedule. Consider a graph coloring problem with a bipartite graph that consists of two sets of nodes N_1 and N_2 and a set of undirected arcs. Set N_1 has n nodes and set N_2 has $t_{\max}$ nodes. Each node in N_1 is connected to m nodes in N_2; a node in N_1 is connected to those m nodes in N_2 that correspond to the time slots in which its operations are supposed to be processed (according to the solution of the network flow problem in the second phase). So each one of the nodes in N_1 is connected to exactly m nodes in N_2, while each node in N_2 is connected to at most m nodes in N_1. A result in graph theory states that if each node in a bipartite graph has at most m arcs, then the arcs can be colored with m different colors in such a way that no node has two arcs of the same color. Each color then corresponds to a given machine.

The coloring algorithm that achieves this can be described as follows. Let g_j, $j = 1, \ldots, n$ denote the degree of a node from set N_1, and let h_t, $t = 1, \ldots, t_{\max}$ denote the degree of a node from set N_2. Let

$$\Delta = \max(g_1, \ldots, g_n, h_1, \ldots, h_{t_{\max}})$$

In order to describe the algorithm that yields a coloring with Δ colors, let $a_{jt} = 1$ if node j from N_1 is connected to node t from N_2, and let $a_{jt} = 0$ otherwise. The a_{jt} are elements of a matrix with n rows and $t_{\max}$ columns. Clearly,

$$\sum_{j=1}^{n} a_{jt} \leq \Delta \qquad t = 1, \ldots, t_{\max}$$

and

$$\sum_{t=1}^{t_{\max}} a_{jt} \leq \Delta \qquad j = 1, \ldots, n$$

The entries (j, t) in the matrix with $a_{jt} = 1$ are referred to as occupied cells. Each occupied cell in the matrix has to be assigned one of the Δ colors in such a way that in no row or column the same color is assigned twice.

The assignment of colors to occupied cells is done by visiting the occupied cells of the matrix row by row from left to right. When visiting occupied cell (j, t) a color c, not yet assigned in column t, is selected. If c is assigned to another cell in row j, say (j, t^*), then there exists a color c' not yet assigned in row j that can be used to replace the assignment of c to (j, t^*). If another cell

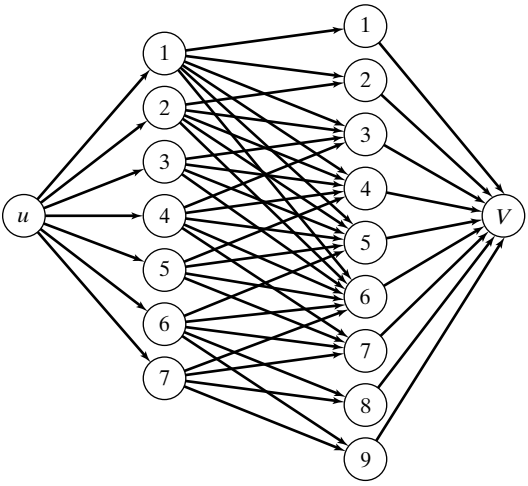


Fig. 8.5 Network flow problem of Phase 2 (Example 8.3.2)

(j^*, t^*) in column j^* already has assignment c', then this assignment is replaced by c. This conflict resolution process stops when there is no remaining conflict. If the partial assignment before coloring (j, t) was feasible, then the conflict resolution procedure yields a feasible coloring in at most n steps.

Example 8.3.2 (Minimizing the Maximum Lateness without Preemptions)

Consider the following instance of $O3 \mid r_j, p_{ij} = 1 \mid L_{\max}$ with 3 machines and 7 jobs.

jobs	1	2	3	4	5	6	7
r_j	0	1	2	2	3	4	5
d_j	5	5	5	6	6	8	8

Assume that $L = 1$. Each job has a deadline $\bar{d}_j = d_j + 1$. So $t_{\max} = 9$. Phase 2 results in the network flow problem described in Figure 8.5. On the left there are 7 nodes that correspond to the 7 jobs and on the right there are 9 nodes that correspond to the 9 time units.

The result of the network flow problem is that the jobs are processed during the time units given in the table below.

jobs	1	2	3	4	5	6	7
time units	1,2,3	2,3,4	4,5,6	4,5,6	5,6,7	7,8,9	7,8,9

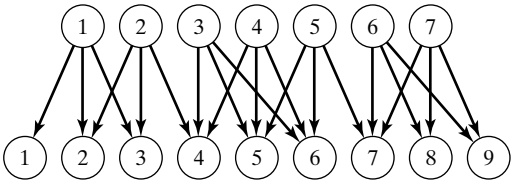


Fig. 8.6 Bipartite graph coloring in Phase 3 (Example 8.3.2)

It can be verified easily that at no point in time more than three jobs are processed simultaneously.

Phase 3 leads to the graph coloring problem. The graph is depicted in Figure 8.6 and the matrix with the appropriate coloring is

$$\left\|\begin{array}{ccccccccc} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \end{array}\right\|$$

It is easy to find a red (r), blue (b) and white (w) coloring that corresponds to a feasible schedule.

$$\left\|\begin{array}{ccccccccc} r & b & w & - & - & - & - & - & - \\ - & r & b & w & - & - & - & - & - \\ - & - & - & r & b & w & - & - & - \\ - & - & - & b & w & r & - & - & - \\ - & - & - & - & r & b & w & - & - \\ - & - & - & - & - & - & r & b & w \\ - & - & - & - & - & - & b & w & r \end{array}\right\|$$

Since there is a feasible schedule for $L = 1$, it has to be verified at this point whether or not there is a feasible schedule for $L = 0$. It can be shown easily that there does not exist a schedule in which every job is completed on time. ||

8.4 The Maximum Lateness with Preemptions

In scheduling it is often the case that the preemptive version of a problem is easier than its nonpreemptive counterpart. That is also the case with $Om \mid prmp \mid L_{\max}$ and $Om \mid\mid L_{\max}$.

Consider $O2 \mid prmp \mid L_{\max}$ and assume that $d_1 \leq \cdots \leq d_n$. Let

$$A_k = \sum_{j=1}^{k} p_{1j}$$

and

$$B_k = \sum_{j=1}^{k} p_{2j}.$$

The procedure to minimize the maximum lateness first considers the due dates as absolute deadlines and then tries to generate a feasible solution. The jobs are scheduled in increasing order of their deadlines, i.e., first job 1, then job 2, and so on. Suppose that jobs $1, \ldots, j-1$ have been scheduled successfully and that job j has to be scheduled next. Let x_j (y_j) denote the total amount of time prior to d_j that machine 1 (2) is idle while machine 2 (1) is busy. Let z_j denote the total amount of time prior to d_j that machines 1 and 2 are idle simultaneously. Note that x_j, y_j, and z_j are not independent, since

$$x_j + z_j = d_j - A_{j-1}$$

and

$$y_j + z_j = d_j - B_{j-1}.$$

The minimum amount of processing that must be done on operation $(1, j)$ while both machines are available is $\max(0, p_{1j} - x_j)$ and the minimum amount of processing on operation $(2, j)$ while both machines are available is $\max(0, p_{2j} - y_j)$. It follows that job j can be scheduled successfully if and only if

$$\max(0, p_{1j} - x_j) + \max(0, p_{2j} - y_j) \leq z_j$$

This inequality is equivalent to the following three inequalities:

$$\begin{aligned} p_{1j} - x_j &\leq z_j \\ p_{2j} - y_j &\leq z_j \\ p_{1j} - x_j + p_{2j} - y_j &\leq z_j \end{aligned}$$

So job j can be scheduled successfully if and only if each one of the following three feasibility conditions holds:

$$A_j \leq d_j$$
$$B_j \leq d_j$$
$$A_j + B_j \leq 2d_j - z_j$$

These inequalities indicate that in order to obtain a feasible schedule an attempt has to be made in each iteration to minimize the value of z_j. The smallest possible values of $z_1, \ldots, z_n$ are defined recursively by

$$z_1 = d_1$$
$$z_j = d_j - d_{j-1} + \max(0, z_{j-1} - p_{1,j-1} - p_{2,j-1}), \qquad j = 2, \ldots, n.$$

In order to verify the existence of a feasible schedule, the values of $z_1, \ldots, z_n$ have to be computed recursively and for each z_j it has to be checked whether it satisfies the third one of the feasibility conditions. There exists a feasible schedule if all the z_j satisfy the conditions.

In order to minimize $L_{\max}$ a parametrized version of the preceding computation has to be done. Replace each d_j by $d_j + L$, where L is a free parameter. The smallest value of L for which there exists a feasible schedule is equal to the minimum value of $L_{\max}$ that can be achieved with the original due dates d_j.

It turns out that there exists also a polynomial time algorithm for the more general open shop with m machines, even when the jobs have different release dates, that is, $Om \mid r_j, prmp \mid L_{\max}$. Again, as in the case with 2 machines, the due dates d_j are considered deadlines $\bar{d}_j$, and an attempt is made to find a feasible schedule where each job is completed before or at its due date. Let

$$a_1 < a_2 < \cdots < a_{p+1}$$

denote the ordered collection of all distinct release dates r_j and deadlines $\bar{d}_j$. So there are p intervals $[a_k, a_{k+1}]$. Let I_k denote the length of interval k, that is,

$$I_k = a_{k+1} - a_k.$$

Let the decision variable x_{ijk} denote the amount of time that operation (i, j) is processed during interval k. Consider the following linear program:

$$\max \quad \sum_{k=1}^{p} \sum_{i=1}^{m} \sum_{j=1}^{n} x_{ijk}$$

subject to

$$\sum_{i=1}^{m} x_{ijk} \leq I_k \qquad \text{for all } 1 \leq j \leq n,\ \ 1 \leq k \leq p$$

$$\sum_{j=1}^{n} x_{ijk} \leq I_k \qquad \text{for all } 1 \leq i \leq m,\ \ 1 \leq k \leq p$$

$$\sum_{k=1}^{p} x_{ijk} \leq p_{ij} \qquad \text{for all } 1 \leq j \leq n,\ \ 1 \leq i \leq m$$

$$x_{ijk} \geq 0 \qquad \text{if } r_j \leq a_k \text{ and } d_j \geq a_{k+1}$$

$$x_{ijk} = 0 \qquad \text{if } r_j \geq a_{k+1} \text{ or } d_j \leq a_k$$

The first inequality requires that no job is scheduled for more than I_k units of time in interval k. The second inequality requires that the amount of processing assigned to any machine is not more than the length of the interval. The third inequality requires that each job is not processed longer than necessary. The constraints on x_{ijk} ensure that no job is assigned to a machine either before its release date or after its due date. An initial feasible solution for this problem is clearly $x_{ijk} = 0$. However, since the objective is to maximize the sum of the x_{ijk}, the third inequality is tight under the optimal solution assuming there exists a feasible solution for the scheduling problem.

If there exists a feasible solution for the linear program, then there exists a schedule with all jobs completed on time. However, the solution of the linear program only gives the optimal values for the decision variables x_{ijk}. It does not specify how the operations should be scheduled within the interval $[a_k, a_{k+1}]$. This scheduling problem within each interval can be solved as follows: consider interval k as an independent open shop problem with the processing time of operation (i, j) being the value x_{ijk} that came out of the linear program. The objective for the open shop scheduling problem for interval k is equivalent to the minimization of the makespan, i.e., $Om \mid prmp \mid C_{\max}$. The polynomial algorithm described in the previous section can then be applied to each interval separately.

If the outcome of the linear program indicates that no feasible solution exists, then (similar to the $m = 2$ case) a parametrized version of the entire procedure has to be carried out. Replace each $\bar{d}_j$ by $\bar{d}_j + L$, where L is a free parameter. The smallest value of L for which there exists a feasible schedule is equal to the minimum value of $L_{\max}$ that can be achieved with the original due dates d_j.

Example 8.4.1 (Minimizing Maximum Lateness with Preemptions)

Consider the following instance of $O3 \mid r_j, prmp \mid L_{\max}$ with 3 machines and 5 jobs.

jobs	1	2	3	4	5
p_{1j}	1	2	2	2	3
p_{2j}	3	1	2	2	1
p_{3j}	2	1	1	2	1
r_j	1	1	3	3	3
d_j	9	7	6	7	9

There are 4 intervals that are determined by $a_1 = 1$, $a_2 = 3$, $a_3 = 6$, $a_4 = 7$, $a_5 = 9$. The lengths of the four intervals are $I_1 = 2$, $I_2 = 3$, $I_3 = 1$, and $I_4 = 2$. There are $4 \times 3 \times 5 = 60$ decision variables x_{ijk}.

The first set of constraints of the linear program has 20 constraints. The first one of this set, i.e., $j = 1, k = 1$, is

$$x_{111} + x_{211} + x_{311} = 2.$$

The second set of constraints has 12 constraints. The first one of this set, i.e., $i = 1, k = 1$, is

$$x_{111} + x_{121} + x_{131} + x_{141} + x_{151} = 2.$$

The third set of constraints has 15 constraints. The first one of this set, i.e., $i = 1, j = 1$, is

$$x_{111} + x_{112} + x_{113} + x_{114} + x_{115} = 1.$$

It turns out that this linear program has no feasible solution. Replacing d_j by $d_j + 1$ yields another linear program that also does not have a feasible solution. Replacing the original d_j by $d_j + 2$ results in the following data set:

jobs	1	2	3	4	5
p_{1j}	1	2	2	2	3
p_{2j}	3	1	2	2	1
p_{3j}	2	1	1	2	1
r_j	1	1	3	3	3
d_j	11	9	8	9	11

There are 4 intervals that are determined by $a_1 = 1$, $a_2 = 3$, $a_3 = 8$, $a_4 = 9$, $a_5 = 11$. The lengths of the four intervals are $I_1 = 2$, $I_2 = 5$, $I_3 = 1$, and $I_4 = 2$. The resulting linear program has feasible solutions and the optimal solution is the following:

$x_{111} = 1$	$x_{211} = 0$	$x_{311} = 1$
$x_{121} = 1$	$x_{221} = 1$	$x_{321} = 0$
$x_{131} = 0$	$x_{231} = 0$	$x_{331} = 0$
$x_{141} = 0$	$x_{241} = 0$	$x_{341} = 0$
$x_{151} = 0$	$x_{251} = 0$	$x_{351} = 0$
$x_{112} = 0$	$x_{212} = 0$	$x_{312} = 1$
$x_{122} = 1$	$x_{222} = 0$	$x_{322} = 0$
$x_{132} = 2$	$x_{232} = 2$	$x_{332} = 1$
$x_{142} = 1$	$x_{242} = 2$	$x_{342} = 2$
$x_{152} = 1$	$x_{252} = 1$	$x_{352} = 1$
$x_{113} = 0$	$x_{213} = 1$	$x_{313} = 0$
$x_{123} = 0$	$x_{223} = 0$	$x_{323} = 1$
$x_{133} = 0$	$x_{233} = 0$	$x_{333} = 0$
$x_{143} = 1$	$x_{243} = 0$	$x_{343} = 0$
$x_{153} = 0$	$x_{253} = 0$	$x_{353} = 0$
$x_{114} = 0$	$x_{214} = 2$	$x_{314} = 0$
$x_{124} = 0$	$x_{224} = 0$	$x_{324} = 0$
$x_{134} = 0$	$x_{234} = 0$	$x_{334} = 0$
$x_{144} = 0$	$x_{244} = 0$	$x_{344} = 0$
$x_{154} = 2$	$x_{254} = 0$	$x_{354} = 0$

Each one of the four intervals has to be analyzed now as a separate $O3 \mid prmp \mid C_{\max}$ problem. Consider, for example, the second interval $[\,3, 8\,]$, i.e., x_{ij2}. The $O3 \mid prmp \mid C_{\max}$ problem for this interval contains the following data.

jobs	1	2	3	4	5
p_{1j}	0	1	2	1	1
p_{2j}	0	0	2	2	1
p_{3j}	1	0	1	2	1

Applying the algorithm described in Section 8.2 results in the schedule presented in Figure 8.7 (which turns out to be nonpreemptive). The schedules in the other three intervals can be determined very easily. ||

8.5 The Number of Tardy Jobs

The $Om \mid p_{ij} = 1 \mid \sum U_j$ problem is strongly related to the problems discussed in the previous sections. In this problem again each job consists of m operations

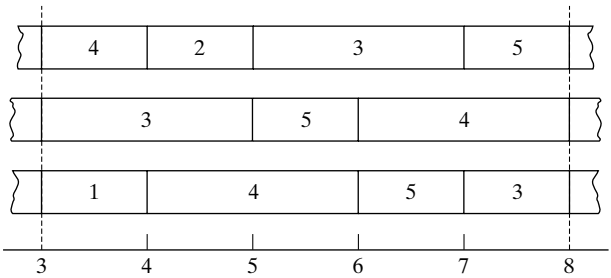


Fig. 8.7 Schedule for interval $[3, 8]$ in Example 8.4.1

and each operation requires one time unit. Assume, without loss of generality, that $d_1 \leq d_2 \leq \cdots \leq d_n$.

It can be shown easily that the set of jobs that are completed on time in an optimal schedule belong to a set $k^*, k^* + 1, \ldots, n$. So the search for an optimal schedule has two aspects. First, it has to be determined what the optimal value of k^* is, and second, given k^*, a schedule has to be constructed in which each job of this set finishes on time.

The value of k^* can be determined via binary search. Given a specific set of jobs that have to be completed on time, a schedule can be generated as follows: Consider the problem $Om \mid r_j, p_{ij} = 1 \mid C_{\max}$, which is a special case of the $Om \mid r_j, p_{ij} = 1 \mid L_{\max}$ problem that is solvable by the polynomial time algorithm described in Section 8.3. Set r_j in this corresponding problem equal to $d_{\max} - d_j$ in the original problem. In essence, the $Om \mid r_j, p_{ij} = 1 \mid C_{\max}$ problem is a time reversed version of the original $Om \mid p_{ij} = 1 \mid \sum U_j$ problem. If for the makespan minimization problem a schedule can be found with a makespan less than $d_{\max}$, then the reverse schedule is applicable to the $Om \mid p_{ij} = 1 \mid \sum U_j$ problem with all jobs completing their processing on time.

8.6 Discussion

This chapter, as several other chapters in this book, focuses mainly on models that are polynomial time solvable. Most open shop models tend to be NP-hard.

For example, very little can be said about the total completion time objective. The $Om \mid\mid \sum C_j$ problem is strongly NP-hard when $m \geq 2$. The $Om \mid prmp \mid \sum C_j$ problem is known to be strongly NP-hard when $m \geq 3$. When $m = 2$ the $Om \mid prmp \mid \sum C_j$ problem is NP-hard in the ordinary sense.

In the same way that a flow shop can be generalized to a flexible flow shop, an open shop can be generalized to a flexible open shop. The fact that the flexible flow shop allows for few structural results gives already an indication that it may be hard to obtain results for the flexible open shop. Even the proportionate cases, i.e., $p_{ij} = p_j$ for all i or $p_{ij} = p_i$ for all j, are hard to analyze.

Another class of models that are closely related to open shops have received recently a considerable amount of attention in the literature. This class of models are typically referred to as concurrent open shops or open shops with job overlap. In these open shops the processing times of any given job on the different machines are allowed to overlap in time (in contrast to the conventional open shops where they are not allowed to overlap). This class of models are at times also referred to as the class of order scheduling models. The motivation is based on the following: Consider a facility with m different machines in parallel and each machine being able to produce a specific type of product. A customer places an order requesting a certain quantity of each product type. After all the items for a given customer have been produced, the entire order can be shipped to the customer.

Exercises (Computational)

8.1. Consider the following instance of $O2 \mid\mid C_{\max}$ and determine the number of optimal schedules that are non-delay.

jobs	1	2	3	4
p_{1j}	9	7	5	13
p_{2j}	5	10	11	7

8.2. Consider the following instance of $O5 \mid\mid C_{\max}$ with 6 jobs and all processing times either 0 or 1. Find an optimal schedule.

jobs	1	2	3	4	5	6
p_{1j}	1	0	0	1	1	1
p_{2j}	1	1	1	0	1	0
p_{3j}	0	1	0	1	1	1
p_{4j}	1	1	1	0	0	1
p_{5j}	1	1	1	1	0	0

8.3. Consider the proportionate open shop $O4 \mid p_{ij} = p_j \mid C_{\max}$ with 6 jobs. Compute the makespan under the optimal schedule.

jobs	1	2	3	4	5	6
p_j	3	5	6	6	8	9

8.4. Consider the problem $O4 \mid\mid C_{\max}$ and consider the *Longest Total Remaining Processing on Other Machines* (*LTRPOM*) rule. Every time a machine is freed the job with the longest total remaining processing time on all *other* machines, among available jobs, is selected for processing. Unforced idleness is not allowed. Consider the following processing times.

jobs	1	2	3	4
p_{1j}	5	5	13	0
p_{2j}	5	7	3	8
p_{3j}	12	5	7	0
p_{4j}	0	5	0	15

(a) Apply the LTRPOM rule. Consider at time 0 first machine 1, then machine 2, followed by machines 3 and 4. Compute the makespan.

(b) Apply the LTRPOM rule. Consider at time 0 first machine 4, then machine 3, followed by machines 2 and 1. Compute the makespan.

(c) Find the optimal schedule and the minimum makespan.

8.5. Find an optimal schedule for the instance of $O4 \mid prmp \mid C_{\max}$ with 4 jobs and with the same processing times as in Exercise 8.4.

8.6. Consider the following instance of $O4 \mid r_j, p_{ij} = 1 \mid L_{\max}$ with 4 machines and 7 jobs.

jobs	1	2	3	4	5	6	7
r_j	0	1	2	2	3	4	5
d_j	6	6	6	7	7	9	9

(a) Find the optimal schedule and the minimum $L_{\max}$.

(b) Compare your result with the result in Example 8.3.2.

8.7. Solve the following instance of the $O2 \mid prmp \mid L_{\max}$ problem.

jobs	1	2	3	4	5	6	7
p_{1j}	7	3	2	5	3	2	3
p_{2j}	3	4	2	4	3	4	5
d_j	5	6	6	11	14	17	20

8.8. Solve the following instance of the proportionate $O2 \mid prmp \mid L_{\max}$ problem.

jobs	1	2	3	4	5	6	7
p_{1j}	7	3	2	5	3	2	3
p_{2j}	7	3	2	5	3	2	3
d_j	5	6	6	11	14	17	20

Can the algorithm described in Section 8.4 be simplified when the processing times are proportionate?

8.9. Consider the Linear Programming formulation of the instance in Exercise 8.7. Write out the objective function. How many constraints are there?

8.10. Consider the following instance of $Om \mid p_{ij} = 1 \mid \sum U_j$ with 3 machines and 8 jobs.

jobs	1	2	3	4	5	6	7	8
d_j	3	3	4	4	4	4	5	5

Find the optimal schedule and the maximum number of jobs completed on time.

Exercises (Theory)

8.11. Show that non-delay schedules for $Om \mid\mid C_{\max}$ have at most $m-1$ idle times on one machine. Show also that if there are $m-1$ idle times on one machine there can be at most $m-2$ idle times on any other machine.

8.12. Consider the following rule for $O2 \mid\mid C_{\max}$. Whenever a machine is freed, start processing the job with the largest sum of remaining processing times on the two machines. Show, through a counterexample, that this rule does not necessarily minimize the makespan.

8.13. Give an example of $Om \mid\mid C_{\max}$ where the optimal schedule is not non-delay.

8.14. Consider $O2 \mid\mid \sum C_j$. Show that the rule which always gives priority to the job with the smallest total remaining processing is not necessarily optimal.

8.15. Consider $O2 \mid prmp \mid \sum C_j$. Show that the rule which always gives preemptive priority to the job with the smallest total remaining processing time is not necessarily optimal.

8.16. Consider $Om \mid\mid C_{\max}$. The processing time of job j on machine i is either 0 or 1. Consider the following rule: At each point in time select, from the machines that have not been assigned a job yet, the machine that still has the largest number of jobs to do. Assign to that machine the job that still has

to undergo processing on the largest number of machines (ties may be broken arbitrarily). Show through a counterexample that this rule does not necessarily minimize the makespan.

8.17. Consider a flexible open shop with two workcenters. Workcenter 1 consists of a single machine and workcenter 2 consists of two identical machines. Determine whether or not LAPT minimizes the makespan.

8.18. Consider the proportionate open shop $Om \mid p_{ij} = p_j \mid C_{\max}$. Find the optimal schedule and prove its optimality.

8.19. Consider the proportionate open shop $Om \mid prmp, p_{ij} = p_j \mid \sum C_j$. Find the optimal schedule and prove its optimality.

8.20. Consider the following two machine hybrid of an open shop and a job shop. Job j has processing time p_{1j} on machine 1 and p_{2j} on machine 2. Some jobs have to be processed first on machine 1 and then on machine 2. Other jobs have to be processed first on machine 2 and then on machine 1. The routing of the remaining jobs may be determined by the scheduler. Describe a schedule that minimizes the makespan.

8.21. Find an upper and a lower bound for the makespan in an m machine open shop when preemptions are not allowed. The processing time of job j on machine i is p_{ij} (i.e., no restrictions on the processing times).

8.22. Compare $Om \mid p_j = 1 \mid \gamma$ with $Pm \mid p_j = 1, chains \mid \gamma$ in which there are n chains consisting of m jobs each. Let Z_1 denote the value of the objective function in the open shop problem and let Z_2 denote the value of the objective function in the parallel machine problem. Find conditions under which $Z_1 = Z_2$ and give examples where $Z_1 > Z_2$.

Comments and References

The LAPT rule for $O2 \mid\mid C_{\max}$ appears to be new. Different algorithms have been introduced before for $O2 \mid\mid C_{\max}$, see for example, Gonzalez and Sahni (1976). Gonzalez and Sahni (1976) provide an NP-hardness proof for $O3 \mid\mid C_{\max}$ and Sevastianov and Woeginger (1998) present a Polynomial Time Approximation Scheme (PTAS) for $Om \mid\mid C_{\max}$.

The polynomial time algorithm for $Om \mid prmp \mid C_{\max}$ is from Lawler and Labetoulle (1978). This algorithm is based on a property of stochastic matrices that is due to Birkhoff and Von Neumann; for an organized proof of this property, see Marshall and Olkin (1979), Chapter 2, Theorems 2.A.2 and 2.C.1. For more work on $Om \mid prmp \mid C_{\max}$, see Gonzalez (1979).

Lawler, Lenstra and Rinnooy Kan (1981) provide a polynomial time algorithm for $O2 \mid prmp \mid L_{\max}$ and show that $O2 \mid\mid L_{\max}$ is NP-Hard in the strong

sense. Cho and Sahni (1981) analyze preemptive open shops with more than two machines and present the linear programming formulation for $Om \mid prmp \mid L_{\max}$.

For results on the minimization of the (weighted) number of late jobs in open shops with unit processing times, see Brucker, Jurisch and Jurisch (1993), Galambos and Woeginger (1995) and Kravchenko (2000).

Achugbue and Chin (1982) present an NP-hardness proof for $O2 \mid\mid \sum C_j$ and Liu and Bulfin (1985) provide an NP-hardness proof for $O3 \mid prmp \mid \sum C_j$. Tautenhahn and Woeginger (1997) analyze the total completion time when all the jobs have unit processing times.

Vairaktarakis and Sahni (1995) obtain results for flexible open shop models.

Concurrent open shops and order scheduling models have received a considerable amount of attention recently, see Wagneur and Sriskandarajah (1993), Sung and Yoon (1998), Ng, Cheng and Yuan (2003), Leung, Li, Pinedo and Sriskandarajah (2005), Yang and Posner (2005), Leung, Li and Pinedo (2005a, 2005b, 2006), and Roemer (2006).

Part II

Stochastic Models

Chapter 9

Stochastic Models: Preliminaries

Production environments in the real world are subject to many sources of uncertainty or randomness. Sources of uncertainty that may have a major impact include machine breakdowns and unexpected releases of high priority jobs. Another source of uncertainty lies in the processing times, which are often not precisely known in advance. A good model for a scheduling problem should address these forms of uncertainty.

There are several ways in which such forms of randomness can be modeled. For example, one could model the possibility of machine breakdowns as an integral part of the processing times. This can be done by modifying the distribution of the processing times to take into account the possibility of breakdowns. Alternatively, one may model breakdowns as a separate stochastic process, that determines when a machine is available and when it is not.

The first section of this chapter describes the framework and notation. The second section deals with distributions and classes of distributions. The third section goes over various forms of stochastic dominance. The fourth section discusses the effect of randomness on the expected value of the objective function given a fixed schedule. The fifth section describes several classes of scheduling policies.

9.1 Framework and Notation

In what follows, it is assumed that the *distributions* of the processing times, the release dates and the due dates are all known in advance, that is, at time zero. The actual *outcome* or *realization* of a random processing time only becomes

M.L. Pinedo, *Scheduling*, DOI: 10.1007/978-0-387-78935-4_9,

known upon the completion of the processing; the realization of a release date or due date becomes known only at that point in time when it actually occurs.

In this part of the book the following notation is used. Random variables are capitalized, while the actual realized values are in lower case. Job j has the following quantities of interest associated with it.

X_{ij} = the random processing time of job j on machine i; if job j is only to be processed on one machine, or if it has the same processing times on each one of the machines it may visit, the subscript i is omitted.

$1/\lambda_{ij}$ = the mean or expected value of the random variable X_{ij}.

R_j = the random release date of job j.

D_j = the random due date of job j.

w_j = the weight (or importance factor) of job j.

This notation is not completely analogous to the notation used for the deterministic scheduling models. The reason why X_{ij} is used as the processing time in stochastic scheduling is because of the fact that P usually refers to a probability. The weight w_j, similar to that in the deterministic models, is basically equivalent to the cost of keeping job j in the system for one unit of time. In the queueing theory literature, which is closely related to stochastic scheduling, c_j is often used for the weight or cost of job j. The c_j and the w_j are equivalent.

9.2 Distributions and Classes of Distributions

Distributions and density functions may take many forms. In what follows, for obvious reasons, only distributions of nonnegative random variables are considered. A density function may be continuous over given intervals and may have mass concentrated at given discrete points. This implies that the distribution function may not be differentiable everywhere (see Figure 9.1). In what follows a distinction is made between continuous time distributions and discrete time distributions.

A random variable from a continuous time distribution may assume any real nonnegative value within one or more intervals. The distribution function of a continuous time distribution is usually denoted by $F(t)$ and its density function by $f(t)$, i.e.,

$$F(t) = P(X \leq t) = \int_0^t f(t)\, dt,$$

where

$$f(t) = \frac{dF(t)}{dt}$$

provided the derivative exists. Furthermore,

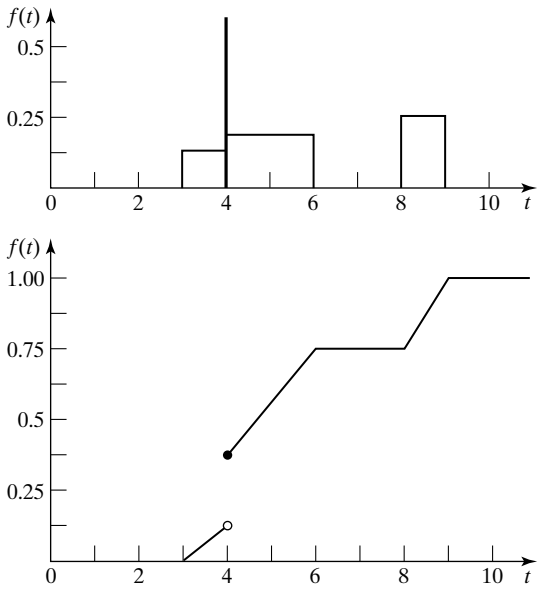


Fig. 9.1 Example of a density function and a distribution function

$$\bar{F}(t) = 1 - F(t) = P(X \geq t).$$

An important example of a continuous time distribution is the *exponential* distribution. The density function of an exponentially distributed random variable X is

$$f(t) = \lambda e^{-\lambda t},$$

and the corresponding distribution function is

$$F(t) = 1 - e^{-\lambda t},$$

which is equal to the probability that X is smaller than t (see Figure 9.2). The mean or expected value of X is

$$E(X) = \int_0^\infty t f(t)\, dt = \int_0^\infty t\, dF(t) = \frac{1}{\lambda}.$$

The parameter λ is referred the *rate* of the exponential distribution.

A random variable from a discrete time distribution may assume only values on the nonnegative integers, i.e., $P(X = t) \geq 0$ for $t = 0, 1, 2, \ldots$ and $P(X = t) = 0$ otherwise. An important discrete time distribution is the *deterministic* distribution. A deterministic random variable assumes a given value with probability one.

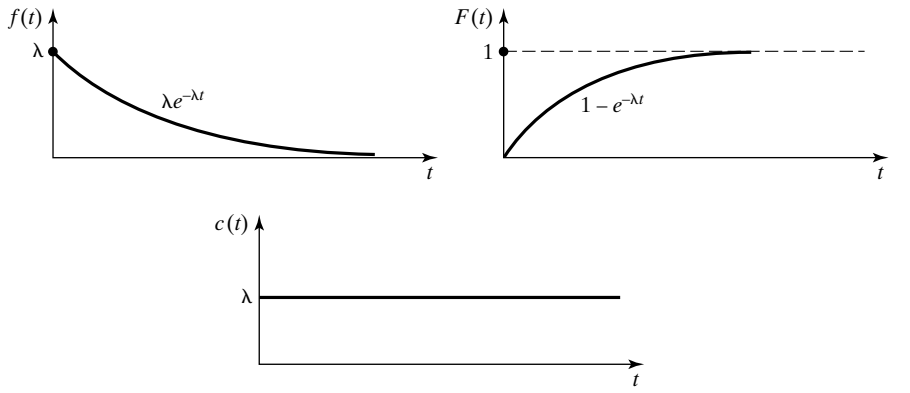


Fig. 9.2 The exponential distribution

Another important example of a discrete time distribution is the *geometric* distribution. The probability that a geometrically distributed random variable X assumes the value t, $t = 0, 1, 2, \ldots$, is

$$P(X = t) = (1 - q)q^t.$$

Its distribution function is

$$P(X \leq t) = \sum_{s=0}^{t} (1 - q)q^s = 1 - \sum_{s=t+1}^{\infty} (1 - q)q^s = 1 - q^{t+1}$$

and its mean is

$$E(X) = \frac{q}{1 - q}.$$

The *completion rate* $c(t)$ of a continuous time random variable X with density function $f(t)$ and distribution function $F(t)$ is defined as follows:

$$c(t) = \frac{f(t)}{1 - F(t)}.$$

This completion rate is equivalent to the failure rate or hazard rate in reliability theory. For an exponentially distributed random variable $c(t) = \lambda$ for all t. That the completion rate is independent of t is one of the reasons why the exponential distribution plays an important role in stochastic scheduling. This property is closely related to the so-called *memoryless* property of the exponential distribution, which implies that the distribution of the *remaining* processing time of a job that already has received processing for an amount of time t, is exponentially distributed with rate λ and therefore identical to its processing time distribution at the very start of its processing.

The completion rate of a discrete time random variable is defined as

$$c(t) = \frac{P(X = t)}{P(X \geq t)}.$$

The discrete time completion rate of the geometric distribution is

$$c(t) = \frac{P(X = t)}{P(X \geq t)} = \frac{(1-q)q^t}{q^t} = 1 - q, \qquad t = 0, 1, 2, \ldots,$$

which is a constant independent of t. This implies that the probability a job is completed at t, given it has not been completed before t, is $1-q$. So the geometric distribution has the memoryless property as well. The geometric distribution is, in effect, the discrete time counterpart of the exponential distribution.

Distributions, either discrete time or continuous time, can be classified based on the completion rate. An *Increasing Completion Rate (ICR)* distribution is defined as a distribution whose completion rate $c(t)$ is increasing in t, while a *Decreasing Completion Rate (DCR)* distribution is defined as a distribution whose completion rate is decreasing in t.

A subclass of the class of continuous time ICR distributions is the class of *Erlang(k,λ)* distributions. The Erlang(k, λ) distribution is defined as

$$F(t) = 1 - \sum_{r=0}^{k-1} \frac{(\lambda t)^r e^{-\lambda t}}{r!}.$$

The Erlang(k, λ) is a k-fold convolution of the same exponential distribution with rate λ. The mean of the Erlang(k, λ) distribution is therefore k/λ. If k equals one, then the distribution is the exponential. If both k and λ go to ∞ while $k/\lambda = 1$, then the Erlang(k, λ) approaches the deterministic distribution with mean 1. The exponential as well as the deterministic distribution are ICR distributions.

A subclass of the class of continuous time DCR distributions is the class of *mixtures of exponentials*. A random variable X is distributed according to a mixture of exponentials if it is exponentially distributed with rate λ_j with probability p_j, $j = 1, \ldots, n$, and

$$\sum_{j=1}^{n} p_j = 1.$$

The exponential distribution is DCR as well as ICR. The class of DCR distributions contains other special distributions. For example, let X with probability p be exponentially distributed with mean $1/p$ and with probability $1 - p$ be zero. The mean and variance of this distribution are $E(X) = 1$ and $Var(X) = 2/p - 1$. When p is very close to zero this distribution is in what follows referred to as an *Extreme Mixture of Exponentials (EME)* distribution. Of course, similar distributions can be constructed for the discrete time case as well.

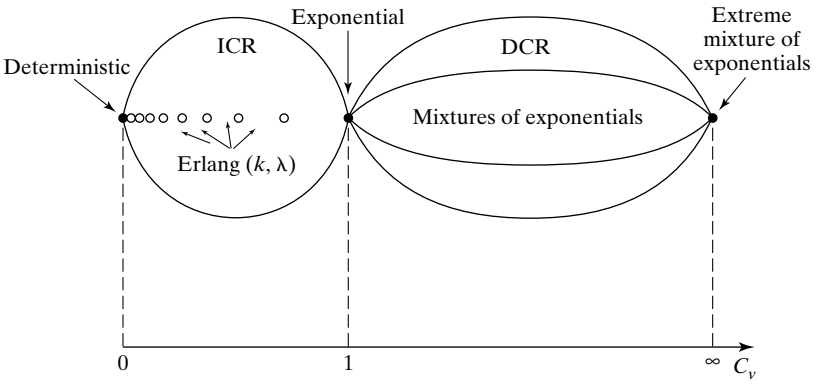


Fig. 9.3 Classes of distributions

One way of measuring the variability of a distribution is through its coefficient of variation $C_v(X)$, which is defined as the square root of the variance (i.e., the standard deviation) divided by the mean, i.e.,

$$C_v(X) = \frac{\sqrt{Var(X)}}{E(X)} = \frac{\sqrt{E(X^2) - (E(X))^2}}{E(X)}.$$

It can be verified easily that the $C_v(X)$ of the deterministic distribution is zero and the $C_v(X)$ of the exponential distribution is 1 (see Figure 9.3). The $C_v(X)$ of an extreme mixture of exponentials may be arbitrarily large (it goes to ∞ when p goes to 0). One may expect the $C_v(X)$ of the geometric to be 1, since the geometric is a discrete time counterpart of the exponential distribution. However, the $C_v(X)$ of the geometric, as it is defined above, is $1/\sqrt{q}$ (see Exercise 9.16).

9.3 Stochastic Dominance

It occurs often in stochastic scheduling that two random variables have to be compared to one another. There are many ways in which one can compare random variables to one another. Comparisons are based on properties referred to as *stochastic dominance*, i.e., a random variable *dominates* another with respect to some stochastic property. All forms of stochastic dominance presented in this section apply to continuous random variables as well as to discrete random variables. The discrete time and continuous time definitions are only in a few cases presented separately. Most forms of stochastic dominance can also be applied in comparisons between a continuous random variable and a discrete random variable.

Definition 9.3.1 (Stochastic Dominance Based on Expectation). *(i) The random variable X_1 is said to be larger in expectation than the random variable X_2 if $E(X_1) \geq E(X_2)$.*

(ii) The random variable X_1 is said to be stochastically larger than the random variable X_2 if

$$P(X_1 > t) \geq P(X_2 > t)$$

or

$$1 - F_1(t) \geq 1 - F_2(t)$$

for all t. This ordering is usually referred to as stochastic ordering and is denoted by $X_1 \geq_{st} X_2$.

(iii) The continuous time random variable X_1 is larger than the continuous time random variable X_2 in the likelihood ratio sense if $f_1(t)/f_2(t)$ is nondecreasing in t, $t \geq 0$. The discrete time random variable X_1 is larger than the discrete time random variable X_2 in the likelihood ratio sense if $P(X_1 = t)/P(X_2 = t)$ is nondecreasing in t, $t = 0, 1, 2, \ldots$ This form of stochastic dominance is denoted by $X_1 \geq_{lr} X_2$.

(iv) The random variable X_1 is almost surely larger than or equal to the random variable X_2 if $P(X_1 \geq X_2) = 1$. This ordering implies that the density functions f_1 and f_2 may overlap at most on one point and is denoted by $X_1 \geq_{a.s.} X_2$.

Ordering in expectation is the crudest form of stochastic dominance. Stochastic ordering implies ordering in expectation since

$$E(X_1) = \int_0^\infty t f_1(t) dt = \int_0^\infty (1 - F_1(t)) dt = \int_0^\infty \bar{F}_1(t) dt$$

(see Exercise 9.11). It can easily be shown that likelihood ratio ordering implies stochastic ordering and that the reverse does not hold.

Example 9.3.2 (Stochastically Ordered Random Variables)

Consider two discrete time random variables X_1 and X_2. Both take values on 1, 2 and 3:

$$P(X_1 = 1) = \frac{1}{8}, \quad P(X_1 = 2) = \frac{3}{8}, \quad P(X_1 = 3) = \frac{1}{2};$$

$$P(X_2 = 1) = \frac{1}{8}, \quad P(X_2 = 2) = \frac{5}{8}, \quad P(X_2 = 3) = \frac{1}{4}.$$

Note that X_1 and X_2 are stochastically ordered but not likelihood ratio ordered as the ratio $P(X_1 = t)/P(X_2 = t)$ is not monotone. ||

It is also easy to find an example of a pair of random variables that are monotone likelihood ratio ordered but not almost surely ordered.

Example 9.3.3 (Likelihood Ratio Ordered Random Variables)
Consider two exponential distributions with rates λ_1 and λ_2. These two distributions are likelihood ratio ordered as

$$\frac{f_1(t)}{f_2(t)} = \frac{\lambda_1 e^{-\lambda_1 t}}{\lambda_2 e^{-\lambda_2 t}} = \frac{\lambda_1}{\lambda_2} e^{-(\lambda_1 - \lambda_2)t},$$

which is monotone in t. Of course, the two exponentials are not almost surely ordered as their density functions overlap everywhere. ||

The four forms of stochastic dominance described above all imply that the random variables being compared, in general, have different means. They lead to the following chain of implications.

almost surely larger $\Longrightarrow$ larger in likelihood ratio sense $\Longrightarrow$
stochastically larger $\Longrightarrow$ larger in expectation

There are several other important forms of stochastic dominance that are based on the *variability* of the random variables under the assumption that the *means are equal*. In the subsequent definitions three such forms are presented. One of these is defined for density functions that are symmetric around the mean, i.e.,

$$f(E(X) + t) = f(E(X) - t)$$

for all $0 \leq t \leq E(X)$. Such a density function then has an upper bound of $2E(X)$. A Normal (Gaussian) density function with mean μ that is truncated at 0 and at 2μ is a symmetric density function.

Definition 9.3.4 (Stochastic Dominance Based on Variance).
(i) The random variable X_1 is said to be larger than the random variable X_2 in the variance sense if the variance of X_1 is larger than the variance of X_2.
(ii) The continuous random variable X_1 is said to be more variable than the continuous random variable X_2 if

$$\int_0^\infty h(t) dF_1(t) \geq \int_0^\infty h(t) dF_2(t)$$

for all convex functions h. The discrete random variable X_1 is said to be more variable than the discrete random variable X_2 if

$$\sum_{t=0}^{\infty} h(t) P(X_1 = t) \geq \sum_{t=0}^{\infty} h(t) P(X_2 = t)$$

for all convex functions h. This ordering is denoted by $X_1 \geq_{cx} X_2$.

(iii) The random variable X_1 is said to be symmetrically more variable than the random variable X_2 if the density functions $f_1(t)$ and $f_2(t)$ are symmetric around the same mean $1/\lambda$ and $F_1(t) \geq F_2(t)$ for $0 \leq t \leq 1/\lambda$ and $F_1(t) \leq F_2(t)$ for $1/\lambda \leq t \leq 2/\lambda$.

Again, the first form of stochastic dominance is somewhat crude. However, any two random variables with equal means can be compared to one another in this way.

From the fact that the functions $h(t) = t$ and $h(t) = -t$ are convex, it follows that if X_1 is "more variable" than X_2 then $E(X_1) \geq E(X_2)$ and $E(X_1) \leq E(X_2)$. So $E(X_1)$ has to be equal to $E(X_2)$. From the fact that $h(t) = t^2$ is convex it follows that $Var(X_1)$ is larger than $Var(X_2)$. Variability ordering is a partial ordering, i.e., not every pair of random variables with equal means can be ordered in this way. At times, variability ordering is also referred to as ordering *in the convex sense*.

It can be shown easily that symmetrically more variable implies more variable in the convex sense but not vice versa.

Example 9.3.5 (Variability Ordered Random Variables)

Consider a deterministic random variable X_1 that always assumes the value $1/\lambda$ and an exponentially distributed random variable X_2 with mean $1/\lambda$. It can be verified easily that X_2 is more variable, but not symmetrically more variable than X_1. ||

Example 9.3.6 (Symmetric Variability Ordered Random Variables)

Consider X_1 with a uniform distribution over the interval $[1, 3]$, i.e., $f_1(t) = 0.5$ for $1 \leq t \leq 3$, and X_2 with a uniform distribution over the interval $[0, 4]$, i.e., $f_2(t) = 0.25$ for $0 \leq t \leq 4$. It is easily verified that X_2 is symmetrically more variable than X_1. ||

The forms of stochastic dominance described in Definition 9.3.4 lead to the following chain of implications:

symmetrically more variable $\Longrightarrow$ more variable $\Longrightarrow$ larger in variance

9.4 Impact of Randomness on Fixed Schedules

The stochastic ordering ($\geq_{st}$) as well as the variability ordering ($\geq_{cx}$) described in the previous section are restricted versions of another form of dominance known as *increasing convex* ordering.

Definition 9.4.1 (Increasing Convex Ordering). *A continuous time random variable X_1 is said to be larger than a continuous time random variable X_2 in the increasing convex sense if*

$$\int_0^\infty h(t)dF_1(t) \geq \int_0^\infty h(t)dF_2(t)$$

for all increasing convex functions h. The discrete time random variable X_1 is said to be larger than the discrete time random variable X_2 in the increasing convex sense if

$$\sum_{t=0}^{\infty} h(t)P(X_1 = t) \geq \sum_{t=0}^{\infty} h(t)P(X_2 = t)$$

for all increasing convex functions h. This ordering is denoted by $X_1 \geq_{icx} X_2$.

Clearly, $E(X_1)$ is larger than, but not necessarily equal to, $E(X_2)$ and $Var(X_1)$ is not necessarily larger than $Var(X_2)$. However, if $E(X_1) = E(X_2)$ then indeed $Var(X_1) \geq Var(X_2)$. From Definition 9.4.1, it immediately follows that

stochastically larger $\Longrightarrow$ larger in the increasing convex sense
more variable $\Longrightarrow$ larger in the increasing convex sense

To see the importance of this form of stochastic dominance consider two vectors of independent random variables, namely $X_1^{(1)}, \ldots, X_n^{(1)}$ and $X_1^{(2)}, \ldots, X_n^{(2)}$. All $2n$ random variables are independent. Let

$$Z_1 = g(X_1^{(1)}, \ldots, X_n^{(1)})$$

and

$$Z_2 = g(X_1^{(2)}, \ldots, X_n^{(2)}),$$

where the function g is increasing convex in each one of the n arguments.

Lemma 9.4.2. *If $X_j^{(1)} \geq_{icx} X_j^{(2)}, j = 1, \ldots, n$, then $Z_1 \geq_{icx} Z_2$.*

Proof. The proof is by induction on n. When $n = 1$ it has to be shown that

$$E\Big(h(g(X_1^{(1)}))\Big) \geq E\Big(h(g(X_1^{(2)}))\Big)$$

with both g and h increasing convex and $X_1^{(1)} \geq_{icx} X_1^{(2)}$. This follows from the definition of variability ordering as the function $h(g(t))$ is increasing and convex in t since

$$\frac{d}{dt}h(g(t)) = h'(g(t))g'(t) \geq 0$$

and

$$\frac{d^2}{dt^2}h(g(t)) = h''(g(t))(g'(t))^2 + h'(g(t))g''(t) \geq 0.$$

Assume as induction hypothesis that the lemma holds for vectors of size $n-1$. Now

$$\begin{aligned} E\Big(h(g(X_1^{(1)}, X_2^{(1)}, \ldots, X_n^{(1)})) \mid X_1^{(1)} = t\Big) &= E\Big(h(g(t, X_2^{(1)}, \ldots, X_n^{(1)})) \mid X_1^{(1)} = t\Big) \\ &= E\Big(h(g(t, X_2^{(1)}, \ldots, X_n^{(1)}))\Big) \\ &\geq E\Big(h(g(t, X_2^{(2)}, \ldots, X_n^{(2)}))\Big) \\ &= E\Big(h(g(t, X_2^{(2)}, \ldots, X_n^{(2)})) \mid X_1^{(1)} = t\Big). \end{aligned}$$

Taking expectations yields

$$E\Big(h(g(X_1^{(1)}, X_2^{(1)}, \ldots, X_n^{(1)}))\Big) \geq E\Big(h(g(X_1^{(1)}, X_2^{(2)}, \ldots, X_n^{(2)}))\Big).$$

Conditioning on on $X_2^{(2)}, \ldots, X_n^{(2)}$ and using the result for $n = 1$ shows that

$$E\Big(h(g(X_1^{(1)}, X_2^{(2)}, \ldots, X_n^{(2)}))\Big) \geq E\Big(h(g(X_1^{(2)}, X_2^{(2)}, \ldots, X_n^{(2)}))\Big),$$

which completes the proof. □

The following two examples illustrate the significance of the previous lemma.

Example 9.4.3 (Stochastic Comparison of Makespans)
Consider two scenarios, each with two machines in parallel and two jobs. The makespan in the first scenario is

$$C_{\max}^{(1)} = \max(X_1^{(1)}, X_2^{(1)})$$

and the makespan in the second scenario is

$$C_{\max}^{(2)} = \max(X_1^{(2)}, X_2^{(2)}).$$

The "max" function is increasing convex in both arguments. From Lemma 9.4.2 it immediately follows that if $X_j^{(1)} \geq_{cx} X_j^{(2)}$, then $X_j^{(1)} \geq_{icx} X_j^{(2)}$ and therefore $C_{\max}^{(1)} \geq_{icx} C_{\max}^{(2)}$. This implies $E(C_{\max}^{(1)}) \geq E(C_{\max}^{(2)})$. ||

Example 9.4.4 (Stochastic Comparison of Total Completion Times)
Consider the problem $1 \mid\mid \sum h(C_j)$ with the function h increasing convex. Consider two scenarios, each one with a single machine and n jobs. The processing time of job j in the first (second) scenario is $X_j^{(1)}$ ($X_j^{(2)}$). Assume

that in both cases the jobs are scheduled in the sequence $1, 2, \ldots, n$. The objective function in scenario $i, i = 1, 2$, is therefore

$$\sum_{j=1}^{n} h(C_j^{(i)}) = h(X_1^{(i)}) + h(X_1^{(i)} + X_2^{(i)}) + \cdots + h(X_1^{(i)} + X_2^{(i)} + \cdots + X_n^{(i)}).$$

The objective function is increasing convex in each one of the n arguments. From Lemma 9.4.2 it follows that if $X_j^{(1)} \geq_{cx} X_j^{(2)}$, then $X_j^{(1)} \geq_{icx} X_j^{(2)}$ and therefore $E(\sum_{j=1}^{n} h(C_j^{(1)})) \geq E(\sum_{j=1}^{n} h(C_j^{(2)}))$. Note that if the function h is linear, the values of the two objectives are equal (since $E(X_j^{(1)}) = E(X_j^{(2)})$). ||

Lemma 9.4.2 turns out to be very useful for determining bounds on performance measures of given schedules when the processing time distributions satisfy certain properties. In the next lemma four distributions, F_1, F_2, F_3 and F_4, are considered, all with mean 1.

Lemma 9.4.5. *If F_1 is deterministic, F_2 is ICR, F_3 is exponential and F_4 is DCR, then*

$$F_1 \leq_{cx} F_2 \leq_{cx} F_3 \leq_{cx} F_4.$$

Proof. For a proof of this result the reader is referred to Barlow and Proschan (1975). The result shown by Barlow and Proschan is actually more general than the result stated here: the distributions F_2 and F_4 do not necessarily have to be ICR and DCR respectively. These distributions may belong to larger classes of distributions. □

The result in Lemma 9.4.5 can also be extended in another direction. It can be shown that for any DCR distribution with mean 1, there exists an Extreme Mixture of Exponentials (EME) distribution with mean 1 that is more variable.

These orderings make it easy to obtain upper and lower bounds on performance measures when processing times are either all ICR or all DCR.

Example 9.4.6 (Bounds on the Expected Makespan)

Consider the scenario of two machines in parallel and two jobs (see Example 9.4.3). Suppose X_1 and X_2 are independent and identically distributed (i.i.d.) according to F with mean 1. If F is deterministic, then the makespan is 1. If F is exponential, then the makespan is 3/2. If F is an EME distribution as defined in Section 9.1, then the makespan is $2-(p/2)$ (that is, if p goes to 0 the makespan goes to 2). Combining the conclusion of Example 9.4.3 with Lemma 9.4.5 yields, when F is ICR, the inequalities

$$1 \leq E(C_{\max}) \leq \frac{3}{2}$$

and, when F is DCR, the inequalities

$$\frac{3}{2} \leq E(C_{\max}) < 2.$$

It is easy to see that the makespan never can be larger than 2. If both jobs are processed on the same machine one after another the expected makespan is equal to 2. ||

9.5 Classes of Policies

In stochastic scheduling, certain conventions have to be made that are not needed in deterministic scheduling. During the evolution of a stochastic process new information becomes available continuously. Job completions and occurrences of random release dates and due dates represent additional information that the decision-maker may wish to take into account when scheduling the remaining part of the process. The amount of freedom the decision maker has in using this additional information is the basis for the various classes of decision making policies. In this section four classes of policies are defined.

The first class of policies is, in what follows, only used in scenarios where all the jobs are available for processing at time zero; the machine environments considered are the single machine, parallel machines and permutation flow shops.

Definition 9.5.1 (Nonpreemptive Static List Policy). *Under a nonpreemptive static list policy the decision maker orders the jobs at time zero according to a priority list. This priority list does not change during the evolution of the process and every time a machine is freed the next job on the list is selected for processing.*

Under this class of policies the decision maker puts at time zero the n jobs in a list (permutation) and the list does not change during the evolution of the process. In the case of machines in parallel, every time a machine is freed, the job at the top of the list is selected as the next one for processing. In the case of a permutation flow shop the jobs are also put in a list in front of the first machine at time zero; every time the first machine is freed the next job on the list is scheduled for processing. This class of nonpreemptive static list policies is in what follows also referred to as the class of *permutation* policies. This class of policies is in a sense similar to the static priority rules usually considered in deterministic models.

Example 9.5.2 (Application of a Nonpreemptive Static List Policy)

Consider a single machine and three jobs. All three jobs are available at time zero. All three jobs have the same processing time distributions, which is 2 with probability .5 and 8 with probability .5. The due date distributions are the same, too. The due date is 1 with probability .5 and 5 with probability .5. If a job is completed at the same time as its due date, it is considered to

be on time. It would be of interest to know the expected number of jobs completed on time under a permutation policy.

Under a permutation policy the first job is completed in time with probability .25 (its processing time has to be 2 and its due date has to be 5); the second job is completed in time with probability .125 (the processing times of the first and second job have to be 2 and the due date of the second job has to be 5); the third job never will be completed in time. The expected number of on-time completions is therefore .375 and the expected number of tardy jobs is $3 - 0.375 = 2.625$. ||

The second class of policies is a preemptive version of the first class and is in what follows only used in scenarios where jobs are released at *different* points in time.

Definition 9.5.3 (Preemptive Static List Policy). *Under a preemptive static list policy the decision maker orders the jobs at time zero according to a priority list. This list includes jobs with nonzero release dates, i.e., jobs that are to be released later. This priority list does not change during the evolution of the process and at any point in time the job at the top of the list of available jobs is the one to be processed on the machine.*

Under this class of policies the following may occur. When there is a job release at some time point and the job released is higher on the static list than the job currently being processed, then the job being processed is preempted and the job released is put on the machine instead.

Under the third and fourth class of policies, the decision-maker is allowed to make his decisions during the evolution of the process. That is, every time he makes a decision, he may take all the information that has become available up to that point in time into account. The third class of policies does not allow preemptions.

Definition 9.5.4 (Nonpreemptive Dynamic Policy). *Under a nonpreemptive dynamic policy, every time a machine is freed, the decision maker is allowed to determine which job goes next. His decision at such a point in time may depend on all the information available, e.g., the current time, the jobs waiting for processing, the jobs currently being processed on other machines and the amount of processing these jobs already have received on these machines. However, the decision maker is not allowed to preempt; once a job has begun its processing, it has to be completed without interruption.*

Example 9.5.5 (Application of a Nonpreemptive Dynamic Policy)

Consider the same problem as in Example 9.5.2. It is of interest to know the expected number of jobs completed on time under a nonpreemptive dynamic policy. Under a nonpreemptive dynamic policy the probability the first job is completed on time is again .25. With probability .5 the first job is completed at time 2. With probability .25 the due dates of both remaining jobs already occurred at time 1 and there will be no more on-time completions. With

probability .75 at least one of the remaining two jobs has a due date at time 5. The probability that the second job put on the machine is completed on time is 3/16 (the probability that the first job has completion time 2 times the probability at least one of the two remaining jobs has due date 5 times the probability that the second job has processing time 2). The expected number of on-time completions is therefore .4375 and the expected number of tardy jobs is 2.5625. ||

The last class of policies is a preemptive version of the third class.

Definition 9.5.6 (Preemptive Dynamic Policy). *Under a preemptive dynamic policy the decision maker may decide at any point in time which jobs should be processed on the machines. His decision at any given point in time may take into account all information available at that point and may involve preemptions.*

Example 9.5.7 (Application of a Preemptive Dynamic Policy)

Consider again the problem of Example 9.5.2. It is of interest to know the expected number of jobs completed on time under a preemptive dynamic policy. Under a preemptive dynamic policy, the probability that the first job is completed on time is again .25. This first job is either taken off the machine at time 1 (with probability .5) or at time 2 (with probability .5). The probability the second job put on the machine is completed on time is 3/8, since the second job enters the machine either at time 1 or at time 2 and the probability of being completed on time is 0.75 times the probability it has processing time 2, which equals 3/8 (regardless of when the first job was taken off the machine). However, unlike under the nonpreemptive dynamic policy, the second job put on the machine is taken off with probability .5 at time 3 and with probability 0.5 at time 4. So there is actually a chance that the third job that goes on the machine will be completed on time. The probability the third job is completed on time is 1/16 (the probability that the due date of the first job is 1 (=.5) times the probability that the due dates of both remaining jobs are 5 (=.25) times the probability that the processing time of the third job is 2 (=.5)). The total expected number of on-time completions is therefore 11/16 = 0.6875 and the expected number of tardy jobs is 2.3125. ||

It is clear that the optimal preemptive dynamic policy leads to the best possible value of the objective as in this class of policies the decision maker has the most information available and the largest amount of freedom. No general statement can be made with regard to a comparison between the optimal preemptive static list policy and the optimal nonpreemptive dynamic policy. The static list policy has the advantage that preemptions are allowed while the nonpreemptive dynamic policy has the advantage that all current information can be taken into account during the process. However, if all jobs are present at time zero and the environment is either a bank of machines in parallel or a

permutation flow shop, then the optimal nonpreemptive dynamic policy is at least as good as the optimal nonpreemptive static list policy (see Examples 9.5.2 and 9.5.5).

There are various forms of minimization in stochastic scheduling. Whenever an objective function has to be minimized, it has to be specified in what *sense* the objective has to be minimized. The crudest form of optimization is in the *expectation* sense, e.g., one wishes to minimize the *expected* makespan, that is $E(C_{\max})$, and find a policy under which the expected makespan is smaller than the expected makespan under any other policy. A stronger form of optimization is optimization in the *stochastic* sense. If a schedule or policy minimizes $C_{\max}$ stochastically, then the makespan under the optimal schedule or policy is *stochastically* less than the makespan under any other schedule or policy. Stochastic minimization, of course, implies minimization in expectation. In the subsequent chapters the objective is usually minimized in expectation. Frequently, however, the policies that minimize the objective in expectation minimize the objective stochastically as well.

Exercises (Computational)

9.1. Determine the completion rate of the discrete Uniform distribution, where $P(X = i) = 0.1$, for $i = 1, \ldots, 10$.

9.2. Determine the completion rate of the continuous Uniform distribution, with density function $f(t) = 0.1$, for $0 \leq t \leq 10$, and 0 otherwise.

9.3. Consider two discrete time random variables X_1 and X_2. Both take values on 1, 2 and 3:

$$P(X_1 = 1) = .2, \quad P(X_1 = 2) = .3, \quad P(X_1 = 3) = .5;$$

$$P(X_2 = 1) = .2, \quad P(X_2 = 2) = .6, \quad P(X_2 = 3) = .2.$$

(a) Are X_1 and X_2 likelihood ratio ordered?
(b) Are X_1 and X_2 stochastically ordered?

9.4. Consider two discrete time random variables X_1 and X_2. Both take values on 1, 2, 3 and 4:

$$P(X_1 = 1) = .125, \;\; P(X_1 = 2) = .375, \;\; P(X_1 = 3) = .375, \;\; P(X_1 = 4) = .125;$$

$$P(X_2 = 1) = .150, \;\; P(X_2 = 2) = .400, \;\; P(X_2 = 3) = .250, \;\; P(X_2 = 4) = .200.$$

(a) Are X_1 and X_2 symmetrically variability ordered?
(b) Are X_1 and X_2 variability ordered?

9.5. Consider three jobs on two machines. The processing times of the three jobs are independent and identically distributed (i.i.d.) according to the discrete distribution $P(X_j = 0.5) = P(X_j = 1.5) = 0.5$.

(a) Compute the expected makespan.

(b) Compute the total expected completion time (*Hint:* note that the sum of the expected completion times is equal to the sum of the expected starting times plus the sum of the expected processing times.)

(c) Compare the results of (a) and (b) with the results in case all three jobs have deterministic processing times equal to 1.

9.6. Do the same as in the previous exercise, but now with two machines and four jobs.

9.7. Consider a flow shop of two machines with unlimited intermediate storage. There are two jobs and the four processing times are i.i.d. according to the discrete distribution

$$P(X_j = 0.5) = P(X_j = 1.5) = 0.5.$$

Compute the expected makespan and the total expected completion time.

9.8. Consider a single machine and three jobs. The three jobs have i.i.d. processing times. The distribution is exponential with mean 1. The due dates of the three jobs are also i.i.d. exponential with mean 1. The objective is to minimize the expected number of tardy jobs, i.e., $E(\sum_{j=1}^{n} U_j)$. Consider a nonpreemptive static list policy.

(a) Compute the probability of the first job being completed on time.

(b) Compute the probability of the second job being completed on time.

(c) Compute the probability of the third job being completed on time.

9.9. Do the same as in the previous exercise but now for a nonpreemptive dynamic policy.

9.10. Do the same as in Exercise 9.8 but now for a preemptive dynamic policy.

Exercises (Theory)

9.11. Show that

$$E(X) = \int_0^\infty \bar{F}(t)dt$$

(recall that $\bar{F}(t) = 1 - F(t)$).

9.12. Show that

$$E(\min(X_1, X_2)) = \int_0^\infty \bar{F}_1(t)\bar{F}_2(t)dt,$$

with X_1 and X_2 being independent.

9.13. Show that

$$E(\min(X_1, X_2)) = \frac{1}{\lambda + \mu},$$

when X_1 is exponentially distributed with rate λ and X_2 exponentially distributed with rate μ.

9.14. Show that

$$E(\max(X_1, X_2)) = E(X_1) + E(X_2) - E(\min(X_1, X_2)),$$

with X_1 and X_2 arbitrarily distributed.

9.15. Compute the coefficient of variation of the Erlang(k, λ) distribution (recall that the Erlang(k, λ) distribution is a convolution of k i.i.d. exponential random variables with rate λ).

9.16. Consider the following variant of the geometric distribution:

$$P(X = t + a) = (1 - q)q^t, \qquad t = 0, 1, 2, \ldots,$$

where $a = (\sqrt{q} - q)/(1 - q)$. Show that the coefficient of variation $C_v(X)$ of this "shifted" geometric is equal to 1.

9.17. Consider the following partial ordering between random variables X_1 and X_2. The random variable X_1 is said to be smaller than the random variable X_2 *in the completion rate sense* if the completion rate of X_1 at time t, say $\lambda_1(t)$, is larger than or equal to the completion rate of X_2, say $\lambda_2(t)$, for any t.

(a) Show that this ordering is equivalent to the ratio $(1 - F_1(t))/(1 - F_2(t))$ being monotone decreasing in t.

(b) Show that

monotone likelihood ratio ordering $\Rightarrow$ completion rate ordering $\Rightarrow$ stochastic ordering.

9.18. Consider m machines in parallel and n jobs with i.i.d. processing times from distribution F with mean 1. Show that

$$\frac{n}{m} \leq E(C_{\max}) < n.$$

Are there distributions for which these bounds are attained?

9.19. Consider a permutation flow shop with m machines in series and n jobs. The processing time of job j on machine i is X_{ij}, distributed according to F with mean 1. Show that

$$n + m - 1 \leq E(C_{\max}) \leq mn.$$

Are there distributions for which these bounds are attained?

9.20. Consider a single machine and n jobs. The processing time of job j is X_j with mean $E(X_j)$ and variance $Var(X_j)$. Find the schedule that minimizes $E(\sum C_j)$ and the schedule that minimizes $Var(\sum C_j)$. Prove your results.

9.21. Assume $X_1 \geq_{st} X_2$. Show through a counterexample that

$$Z_1 = 2X_1 + X_2 \geq_{st} 2X_2 + X_1 = Z_2.$$

is not necessarily true.

Comments and References

For a general overview of stochastic scheduling problems, see Dempster, Lenstra and Rinnooy Kan (1982), Möhring and Radermacher (1985b) and Righter (1994).

For an easy to read and rather comprehensive treatment of distributions and classes of distributions based on completion (failure) rates, see Barlow and Proschan (1975) (Chapter 3).

For a lucid and comprehensive treatment of the several forms of stochastic dominance, see Ross (1995). A definition of the form of stochastic dominance based on symmetric variability appears in Pinedo (1982). For a scheduling application of monotone likelihood ratio ordering see Brown and Solomon (1973). For a scheduling application of completion rate ordering (Exercise 9.17), see Pinedo and Ross (1980). For an overview of the different forms of stochastic dominance and their importance with respect to scheduling, see Chang and Yao (1993) and Righter (1994).

For the impact of randomness on fixed schedules, see Pinedo and Weber (1984), Pinedo and Schechner (1985), Pinedo and Wie (1986), Shanthikumar and Yao (1991) and Chang and Yao (1993).

Many classes of scheduling policies have been introduced over the years; see, for example, Glazebrook (1981a, 1981b, 1982), Pinedo (1983) and Möhring, Radermacher and Weiss (1984, 1985).

Chapter 10

Single Machine Models (Stochastic)

Stochastic models, especially with exponential processing times, may often contain more structure than their deterministic counterparts and may lead to results which, at first sight, seem surprising. Models that are NP-hard in a deterministic setting often allow a simple priority policy to be optimal in a stochastic setting.

In this chapter single machine models with arbitrary processing times in a nonpreemptive setting are discussed first. Then the preemptive cases are considered, followed by models where the processing times are likelihood ratio ordered. Finally, models with exponentially distributed processing times are analyzed.

10.1 Arbitrary Distributions without Preemptions

For a number of stochastic problems, finding the optimal policy is equivalent to solving a deterministic scheduling problem. Usually, when such an equivalence relationship exists, the deterministic counterpart can be obtained by replacing all random variables with their means. The optimal schedule for the deterministic problem then minimizes the objective of the stochastic version in expectation.

One such case is when the objective in the deterministic counterpart is linear in $p_{(j)}$ and $w_{(j)}$, where $p_{(j)}$ and $w_{(j)}$ denote the processing time and weight of the job in the j-th position in the sequence.

M.L. Pinedo, *Scheduling*, DOI: 10.1007/978-0-387-78935-4_10,

This observation implies that it is easy to find the optimal permutation schedule for the stochastic counterpart of $1 \mid\mid \sum w_j C_j$, when the processing time of job j is X_j, from an arbitrary distribution F_j, and the objective is $E(\sum w_j C_j)$. This problem leads to the stochastic version of the WSPT rule, which sequences the jobs in decreasing order of the ratio $w_j/E(X_j)$ or $\lambda_j w_j$. In what follows this rule is referred to either as the *Weighted Shortest Expected Processing Time first (WSEPT)* rule or as the $''\lambda w''$ rule.

Theorem 10.1.1. *The WSEPT rule minimizes the expected sum of the weighted completion times in the class of nonpreemptive static list policies as well as in the class of nonpreemptive dynamic policies.*

Proof. The proof for nonpreemptive static list policies is similar to the proof for the deterministic counterpart of this problem. The proof is based on an adjacent pairwise interchange argument identical to the one used in the proof of Theorem 3.1.1. The only difference is that the p_j's in that proof have to be replaced by $E(X_j)$'s.

The proof for nonpreemptive dynamic policies needs an additional argument. It is easy to show that it is true for $n = 2$ (again an adjacent pairwise interchange argument). Now consider three jobs. It is clear that the last two jobs have to be sequenced according to the λw rule. These last two jobs will be sequenced in this order independent of what occurs during the processing of the first job and of the completion time of the first job. There are then three sequences that may occur: each of the three jobs starting first and the remaining two jobs sequenced according to the λw rule. A simple interchange argument between the first job and the second shows that all three jobs have to sequenced according to the λw rule. It can be shown by induction that all n jobs have to be sequenced according to the λw rule in the class of nonpreemptive dynamic policies: suppose it is true for $n-1$ jobs. If there are n jobs, then it follows from the induction hypothesis that the last $n-1$ jobs have to be sequenced according to the λw rule. Suppose the first job is not the job with the highest $\lambda_j w_j$. Interchanging this job with the second job in the sequence, i.e., the job with the highest $\lambda_j w_j$, results in a decrease in the expected value of the objective function. This completes the proof of the theorem. □

It can be shown that the nonpreemptive WSEPT rule is also optimal in the class of *preemptive* dynamic policies when all n processing time distributions are ICR. This follows from the fact that at any time when a preemption is contemplated, the λw ratio of the job currently on the machine is actually higher than when it was put on the machine (the expected remaining processing time of an ICR job decreases as processing goes on). If the ratio of the job was the highest among the remaining jobs when it was put on the machine, it remains the highest while it is being processed.

The same cannot be said about jobs with DCR distributions. The expected remaining processing time then *increases* while a job is being processed. So the weight divided by the expected remaining processing time of a job, while it is

being processed, *decreases* with time. Preemptions may therefore be advantageous when processing times are DCR.

Example 10.1.2 (Optimal Policy with Random Variables that are DCR)

Consider n jobs with the processing time X_j distributed as follows. The processing time X_j is 0 with probability p_j and it is distributed according to an exponential with rate λ_j with probability $1-p_j$. Clearly, this distribution is DCR as it is a mixture of two exponentials with rates ∞ and λ_j. The objective to be minimized is the expected sum of the weighted completion times. The optimal preemptive dynamic policy is clear. All n jobs have to be tried out for a split second at time zero, in order to determine which jobs have zero processing times. If a job does not have zero processing time, it is taken immediately off the machine. After having determined in this way which jobs have nonzero processing times, these remaining jobs are sequenced in decreasing order of $\lambda_j w_j$. ||

Consider the following generalization of the stochastic counterpart of $1 \mid\mid \sum w_j C_j$ described above. The machine is subject to breakdowns. The "up" times, i.e., the times that the machine is in operation, are exponentially distributed with rate ν. The "down" times are independent and identically distributed (i.i.d.) according to distribution G with mean $1/\mu$. It can be shown that even in this case the λw rule is optimal. Actually, it can be shown that this stochastic problem with breakdowns is equivalent to a similar deterministic problem without breakdowns. The processing time of job j in the equivalent deterministic problem is determined as follows. Let X_j denote the original random processing time of job j when there are no breakdowns and let Y_j denote the time job j occupies the machine, including the time that the machine is not in operation. The following relationship can be determined easily (see Exercise 10.11).

$$E(Y_j) = E(X_j)(1 + \frac{\nu}{\mu}).$$

This relationship holds because of the exponential uptimes of the machines and the fact that all the breakdowns have the same mean. So, even with the breakdown process described above, the problem is still equivalent to the deterministic problem $1 \mid\mid \sum w_j C_j$ without breakdowns.

The equivalences between the single machine stochastic models and their deterministic counterparts go even further. Consider the stochastic counterpart of $1 \mid \mathit{chains} \mid \sum w_j C_j$. If in the stochastic counterpart the jobs are subject to precedence constraints that take the form of chains, then Algorithm 3.1.4 can be used for minimizing the expected sum of the weighted completion times (in the definition of the ρ-factor the p_j is again replaced by the $E(X_j)$).

Consider now the stochastic version of $1 \mid\mid \sum w_j(1 - e^{-rC_j})$ with arbitrarily distributed processing times. This problem leads to the stochastic version of

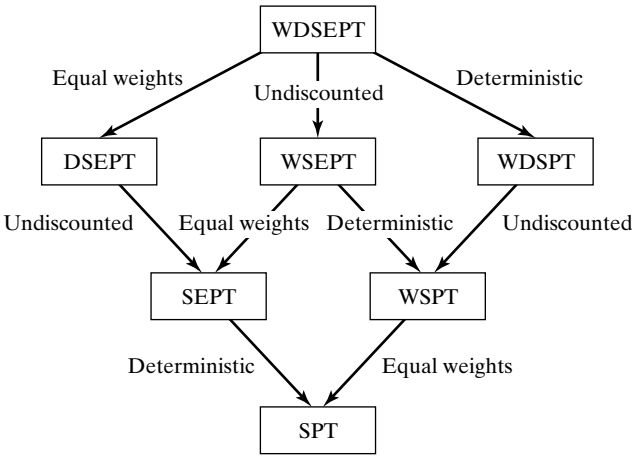


Fig. 10.1 Hierarchy of scheduling rules

the WDSPT rule which sequences the jobs in decreasing order of the ratio

$$\frac{w_j E(e^{-rX_j})}{1 - E(e^{-rX_j})}.$$

This rule is referred to as the *Weighted Discounted Shortest Expected Processing Time first (WDSEPT)* rule. This rule is, in a sense, a generalization of a number of rules considered before (see Figure 10.1).

Theorem 10.1.3. *The WDSEPT rule minimizes the expected weighted sum of the discounted completion times in the class of nonpreemptive static list policies as well as in the class of nonpreemptive dynamic policies.*

Proof. The optimality of this rule can be shown again through a straightforward pairwise interchange argument similar to the one used in the proof of Theorem 3.1.6. The $w_j e^{-r(t+p_j)}$ is replaced by the $w_j E(e^{-r(t+X_j)})$. Optimality in the class of nonpreemptive dynamic policies follows from an induction argument similar to the one presented in Theorem 10.1.1. □

Example 10.1.4 (Application of the WDSEPT Rule)

Consider two jobs with equal weights, say 1. The processing time distribution of the first job is a continuous time uniform distribution over the interval $[0, 2]$, i.e., $f_1(t) = .5$ for $0 \leq t \leq 2$. The processing time distribution of the second job is a discrete time uniform distribution with 0, 1 and 2 as possible outcomes, i.e.,

$$P(X_2 = 0) = P(X_2 = 1) = P(X_2 = 2) = \frac{1}{3}.$$

Clearly, $E(X_1) = E(X_2) = 1$. The discount factor r is 1/10. In order to determine the priority indices of the two jobs $E(e^{-rX_1})$ and $E(e^{-rX_2})$ have to be computed:

$$E(e^{-rX_1}) = \int_0^\infty e^{-rt} f_1(t)dt = \int_0^2 \frac{1}{2} e^{-0.1t} dt = .9063$$

and

$$E(e^{-rX_2}) = \sum_{t=0}^{\infty} e^{-rt} P(X_2 = t) = \frac{1}{3} + \frac{1}{3}e^{-0.1} + \frac{1}{3}e^{-0.2} = .9078.$$

The priority index of job 1 is therefore 9.678 and the priority index of job 2 is 9.852. This implies that job 2 has to be processed first and job 1 second. If the discount factor would have been zero any one of the two sequences would have been optimal. Observe that $Var(X_1) = 1/3$ and $Var(X_2) = 2/3$. So in this case it is optimal to process the job with the larger variance first. ||

In the theorem above the optimality of the WDSEPT rule is shown for the class of nonpreemptive static list policies as well as for the class of nonpreemptive dynamic policies. Precedence constraints can be handled in the same way as they are handled in the deterministic counterpart (see Exercise 3.22). The model considered in Theorem 10.1.3, without precedence constraints, will be considered again in the next section in an environment that allows preemptions.

The remaining part of this section focuses on due date related problems. Consider the stochastic counterpart of $1 \mid\mid L_{\max}$ with processing times that have arbitrary distributions and deterministic due dates. The objective is to minimize the expected maximum lateness.

Theorem 10.1.5. *The EDD rule minimizes expected maximum lateness for arbitrarily distributed processing times and deterministic due dates in the class of nonpreemptive static list policies, the class of nonpreemptive dynamic policies and the class of preemptive dynamic policies.*

Proof. It is clear that the EDD rule minimizes the maximum lateness for any realization of processing times (after conditioning on the processing times, the problem is basically a deterministic problem and Algorithm 3.2.1 applies). If the EDD rule minimizes the maximum lateness for any realization of processing times then it minimizes the maximum lateness also in expectation (it actually minimizes the maximum lateness almost surely). □

Example 10.1.6 (Application of the EDD Rule)

Consider two jobs with deterministic due dates. The processing time distributions of the jobs are discrete:

$$P(X_1 = 1) = P(X_1 = 2) = P(X_1 = 4) = \frac{1}{3}$$

and

$$P(X_2 = 2) = P(X_2 = 4) = \frac{1}{2}.$$

The due date of the first job is $D_1 = 1$ and the due date of the second job is $D_2 = 4$. Now

$$\begin{aligned}
E(\max(L_1, L_2)) = {} & E\Big(\max(L_1, L_2) \mid X_1 = 1, X_2 = 2\Big) \times P(X_1 = 1, X_2 = 2) \\
& + E\Big(\max(L_1, L_2) \mid X_1 = 1, X_2 = 4\Big) \times P(X_1 = 1, X_2 = 4) \\
& + E\Big(\max(L_1, L_2) \mid X_1 = 2, X_2 = 2\Big) \times P(X_1 = 2, X_2 = 2) \\
& + E\Big(\max(L_1, L_2) \mid X_1 = 2, X_2 = 4\Big) \times P(X_1 = 2, X_2 = 4) \\
& + E\Big(\max(L_1, L_2) \mid X_1 = 4, X_2 = 2\Big) \times P(X_1 = 4, X_2 = 2) \\
& + E\Big(\max(L_1, L_2) \mid X_1 = 4, X_2 = 4\Big) \times P(X_1 = 4, X_2 = 4) \\
= {} & (0 + 1 + 1 + 2 + 3 + 4)\frac{1}{6} \\
= {} & \frac{11}{6}.
\end{aligned}$$

It can easily be verified that scheduling job 2 first and job 1 second results in a larger $E(\max(L_1, L_2))$.

Note, however, that in any given sequence $E(L_{\max}) = E(\max(L_1, L_2))$ does not necessarily have to be equal to $\max(E(L_1), E(L_2))$. Under sequence $1, 2$,

$$E(L_1) = 0 \times \frac{1}{3} + 1 \times \frac{1}{3} + 3 \times \frac{1}{3} = \frac{4}{3},$$

while

$$E(L_2) = \frac{1}{3}\Big(\frac{1}{2} \times 0 + \frac{1}{2} \times 1\Big) + \frac{1}{3}\Big(\frac{1}{2} \times 0 + \frac{1}{2} \times 2\Big) + \frac{1}{3}\Big(\frac{1}{2} \times 2 + \frac{1}{2} \times 4\Big) = \frac{3}{2}.$$

So $\max(E(L_1), E(L_2)) = 3/2$, which is smaller than $E(\max(L_1, L_2))$. ||

It can be shown that the EDD rule not only minimizes

$$E(L_{\max}) = E(\max(L_1, \ldots, L_n)),$$

but also $\max(E(L_1), \ldots, E(L_n))$.

It is even possible to develop an algorithm for a stochastic counterpart of the more general $1 \mid prec \mid h_{\max}$ problem considered in Chapter 3. In this problem the objective is to minimize the maximum of the n expected costs incurred by the n jobs, i.e., the objective is to minimize

$$\max\Big(E(h_1(C_1)), \ldots, E(h_n(C_n))\Big),$$

where $h_j(C_j)$ is the cost incurred by job j being completed at C_j. The cost function h_j is nondecreasing in the completion time C_j. The algorithm is a modified version of Algorithm 3.2.1. The version here is also a backward procedure. Whenever one has to select a schedulable job for processing, it is clear that the distribution of its completion time is the convolution of the processing times of the jobs that have not yet been scheduled. Let f_{J^c} denote the density function of the convolution of the processing times of the set of unscheduled jobs J^c. Job j^* is then selected to be processed last among the set of jobs J^c if

$$\int_0^\infty h_{j^*}(t) f_{J^c}(t) dt = \min_{j \in J^c} \int_0^\infty h_j(t) f_{J^c}(t) dt.$$

The L.H.S. denotes the expected value of the penalty for job j^* if it is the last job to be scheduled among the jobs in J^c. This rule replaces the first part of Step 2 in Algorithm 3.2.1. The proof of optimality is similar to the proof of optimality for the deterministic case. However, implementation of the algorithm may be significantly more cumbersome because of the evaluation of the integrals.

Example 10.1.7 (Minimizing Maximum Expected Cost)

Consider three jobs with random processing times X_1, X_2 and X_3 from distributions F_1, F_2 and F_3. Particulars of the processing times and cost functions are given in the table below.

jobs	1	2	3
$h_j(C_j)$	$1 + 2\,C_1$	38	$4\,C_3$
$E(X_j)$	6	18	12

Note that all three cost functions are linear. This makes the evaluations of all the necessary integrals very easy, since the integrals are a linear function of the means of the processing times. If job 1 is the last one to be completed, the expected penalty with regard to job 1 is $1 + 2(6 + 18 + 12) = 73$; the expected penalty with regard to job 2 is 38 and with regard to job 3 is $4(6 + 18 + 12) = 144$. The procedure selects job 2 for the last position. If job 1 would go second the expected penalty would be $1 + 2(6 + 12) = 37$ and if job 3 would go second its expected penalty would be $4(6 + 12) = 72$. So job 1 is selected to go second and job 3 goes first. If job 3 goes first its expected penalty is 48. The maximum of the three expected penalties under sequence $3, 1, 2$ is $\max(48, 37, 38) = 48$. ||

Note that the analysis in Example 10.1.7 is particularly easy since all three cost functions are linear. The only information needed with regard to the processing time of a job is its mean. If any one of the cost functions is nonlinear, the expected penalty of the corresponding job is more difficult to compute; its entire distribution has to be taken into account. The integrals may have to be evaluated through approximation methods.

10.2 Arbitrary Distributions with Preemptions: the Gittins Index

Consider the problem of scheduling n jobs with random processing times $X_1, \ldots, X_n$ from discrete time distributions. The scheduler is allowed to preempt the machine at discrete times 0, 1, 2, ... If job j is completed at the integer completion time C_j a reward $w_j \beta^{C_j}$ is received, where β is a discount factor between 0 and 1. The value of β is typically close to 1. It is of interest to determine the policy that maximizes the total expected reward.

Before proceeding with the analysis it may be useful to relate this problem to another described earlier. It can be argued that this problem is a discrete-time version of the continuous-time problem with the objective

$$E\Big(\sum_{j=1}^{n} w_j (1 - e^{-rC_j})\Big).$$

The argument goes as follows. Maximizing $\sum w_j \beta^{C_j}$ is equivalent to minimizing $\sum w_j (1 - \beta^{C_j})$. Consider the limiting case where the size of the time unit is decreased and the number of time units is increased accordingly. If the time unit is changed from 1 to $1/k$ with $(k > 1)$, the discount factor β has to be adjusted as well. The appropriate discount factor, which corresponds to the new time unit $1/k$, is then $\sqrt[k]{\beta}$. If β is relatively close to one, then

$$\sqrt[k]{\beta} \approx 1 - \frac{1-\beta}{k}.$$

So

$$\beta^{C_j} \approx \Big(1 - \frac{1-\beta}{k}\Big)^{kC_j}$$

and

$$\lim_{k \to \infty} \Big(1 - \frac{1-\beta}{k}\Big)^{kC_j} = e^{-(1-\beta)C_j}.$$

This implies that when β is relatively close to one, it is similar to the $1 - r$ used in earlier models. The current model is then a discrete time stochastic version of the deterministic problem $1 \mid prmp \mid \sum w_j (1 - \exp(-rC_j))$ discussed in Chapter 3. The stochastic version can be used to model the problem described in Example 1.1.4 (the problem described in Example 1.1.4 is actually slightly more general as it assumes that jobs have different release dates).

In order to characterize the optimal policy for this discrete time scheduling problem with preemptions, it is actually easier to consider a more general reward process. Let $x_j(t)$ denote the state of job j at time t. This state is basically determined by the amount of processing job j has received prior to time t. If the decision is made to process job j during the interval $[t, t+1]$ a random reward $W_j(x_j(t))$ is received. This random reward is a function of the state $x_j(t)$ of job j. Clearly, this reward process is more general than the one described at

the beginning of this section under which a (fixed) reward w_j is received only if job j is *completed* during the interval $[t, t+1]$. In what follows the optimal policy is determined for the more general reward process.

The decision to be made at any point in time has two elements. First, a decision has to be made with regard to the job selection. Second, if a particular job is selected, a decision has to be made with regard to the amount of time the job should remain on the machine. The first point in time at which another job is considered for processing is referred to as the *stopping time* and is denoted by τ.

It is shown in what follows that the solution of this problem can be characterized by functions G_j, $j = 1, \ldots, n$, with the property that processing job j^* at time t is optimal if and only if

$$G_{j^*}(x_{j^*}) = \max_{1 \le j \le n} G_j(x_j),$$

where x_j is the amount of processing job j already has received by time t, i.e., $x_j(t) = x_j$. The function G_j is called the *Gittins index* and is defined as

$$G_j(x_j) = \max_{\pi} \frac{E_\pi\Big(\sum_{s=0}^{\tau-1} \beta^{s+1} W_j(x_j(s)) \mid x_j(0) = x_j \Big)}{E_\pi\Big(\sum_{s=0}^{\tau-1} \beta^{s+1} \mid x_j(0) = x_j \Big)},$$

with the stopping time τ being determined by the policy π. However, the value τ is not necessarily the time at which the processing of job j stops. Job j may actually be completed *before* the stopping time.

This is one of the more popular forms in which the Gittins index is presented in the literature. The Gittins index may be described in words as the largest value that is obtainable by dividing the total discounted expected reward over a given time period (determined by the stopping time) by the discounted time itself.

The next theorem focuses on the optimal policy under the more general reward process described above.

Theorem 10.2.1. *The policy that maximizes the total discounted expected reward in the class of preemptive dynamic policies prescribes, at each point in time, the processing of the job with the largest Gittins index.*

Proof. Assume that the scheduler has to pay a fixed charge to the machine if he decides to process job j. Call this charge the prevailing charge. Suppose job j is the only one in the system and the scheduler has to decide whether or not to process it. The scheduler has to decide to process the job for a number of time units, observe the state as it evolves and then stop processing the moment the prevailing charge does not justify any further processing. If the prevailing charge is very small, it is advantageous to continue processing whereas if the prevailing charge is too high, any further processing leads to losses.

As a function of the state of job j, say x_j, the so-called fair charge is defined as $\gamma_j(x_j)$, the level of prevailing charge for which the optimal action will neither be profitable nor cause losses. That is, the fair charge is the level of the prevailing charge at which the costs to the scheduler are exactly in balance with the expected outside rewards to be obtained by processing the jobs according to the optimal policy. So

$$\gamma_j(x_j) = \max\Big(\gamma : \max_{\pi} E_{\pi}\Big(\sum_{s=0}^{\tau-1} \beta^{s+1}\Big(W_j(x_j(s)) - \gamma\Big) \mid x_j(0) = x_j\Big) \geq 0\Big),$$

where the policy π determines the stopping time τ. Thus the fair charge is determined by the optimal action which prescribes processing the job for exactly τ time units or until completion, whichever comes first. Processing the job for less time causes losses and processing the job for more than τ time units causes losses also. Suppose the prevailing charge is reduced to the fair charge whenever the scheduler would have decided to stop processing the job due to the prevailing charge being too high. Then the scheduler would keep the job on the machine until completion as the process now becomes a fair game. In this case, the sequence of prevailing charges for the job is nonincreasing in the number of time units the job already has undergone processing.

Suppose now that there are n different jobs and at each point in time the scheduler has to decide which one to process during the next time period. Assume that initially the prevailing charge for each job is set equal to its fair charge and the prevailing charges are reduced periodically afterwards, as described above, every time a stopping time is reached. Thus the scheduler never pays more than the fair charge and can make sure that his expected total discounted profit is nonnegative. However, it is also clear that his total profit cannot be positive, because it would have to be positive for at least one job and this cannot happen as the prevailing charges consistently are set equal to the fair charges. The scheduler only can break even if, whenever he selects a job, he processes the job according to the optimal policy. That is, he has to continue processing this job until the optimal stopping time, that determines the level of the fair charge. If he does not act accordingly, he acts suboptimally and incurs an expected discounted loss. So the scheduler acts optimally if, whenever he starts processing a job, he continues to do so as long as the job's fair charge remains greater than its prevailing charge.

The sequence of prevailing charges for each job is a nonincreasing function of the number of time units the job has undergone processing. By definition it is a sequence that is independent of the policy adopted. If for each job the sequence of prevailing charges is nonincreasing and if the scheduler adopts the policy of always processing the job with the largest prevailing charge, then he incurs charges in a nonincreasing sequence. This intertwining of sequences of prevailing charges into a single nonincreasing sequence is unique (in terms of charges, not necessarily in terms of jobs). Thus the policy of processing the job with the largest prevailing charge maximizes the expected total discounted

charge paid by the scheduler. This maximum quantity is an upper bound for the expected total discounted reward obtained by the scheduler. This upper bound is achieved by the proposed policy, since the policy forces the scheduler to process a job, without interruption, for the time that its fair charge exceeds its prevailing charge (this leads to a fair game in which the scheduler's total expected discounted profit is zero). This completes the proof of the theorem. □

From the expression for the fair charge $\gamma_j(x_j)$ the expression for the Gittins index immediately follows. For the special case in which a fixed reward w_j is received only upon completion of job j, the Gittins index becomes

$$G_j(x_j) = \max_{\tau>0} \frac{\sum_{s=0}^{\tau-1} \beta^{s+1} w_j P(X_j = x_j + 1 + s \mid X_j > x_j)}{\sum_{s=0}^{\tau-1} \beta^{s+1} P(X_j \geq x_j + 1 + s \mid X_j > x_j)}$$
$$= \max_{\tau>0} \frac{\sum_{s=0}^{\tau-1} \beta^{s+1} w_j P(X_j = x_j + 1 + s)}{\sum_{s=0}^{\tau-1} \beta^{s+1} P(X_j \geq x_j + 1 + s)},$$

The Gittins index is determined by the maximum of the ratio of the R.H.S. over all possible stopping times. As the expectations of the sums in the numerator and denominator must take into account that the scheduler does not keep the job on the machine for τ time units in case the job is completed early, each element in one of the sums has to be multiplied with the appropriate probability.

As the computation of Gittins indices at first sight may seem somewhat involved, an example is in order. The following example considers the reward process where a fixed reward w_j is obtained upon completion of job j.

Example 10.2.2 (Application of the Gittins Index)

Consider three jobs with $w_1 = 60, w_2 = 30$ and $w_3 = 40$. Let p_{jk} denote the probability that the processing time of job j takes k time units, i.e.,

$$p_{jk} = P(X_j = k)$$

The processing times of the three jobs take only values on the integers 1, 2 and 3.

$$p_{11} = \frac{1}{6}, \quad p_{12} = \frac{1}{2}, \quad p_{13} = \frac{1}{3};$$
$$p_{21} = \frac{2}{3}, \quad p_{22} = \frac{1}{6}, \quad p_{23} = \frac{1}{6};$$
$$p_{31} = \frac{1}{2}, \quad p_{32} = \frac{1}{4}, \quad p_{33} = \frac{1}{4};$$

Assume the discount rate β to be 0.5. If job 1 is put on the machine at time 0, the discounted expected reward at time 1 is $w_1 p_{11} \beta$ which is 5. The discounted expected reward obtained at time 2 is $w_1 p_{12} \beta^2$ which is 7.5. The discounted expected reward obtained at time 3 is $w_1 p_{13} \beta^3$ which is 2.5. The

Gittins index for job 1 at time 0 can now be computed easily.

$$G_1(x_1(0)) = G_1(0) = \max\Big(\frac{5}{0.5},\ \frac{5+7.5}{0.5+0.208},\ \frac{5+7.5+2.5}{0.5+0.208+0.042}\Big) = 20.$$

Thus, if job 1 is selected as the one to go on the machine at time 0, it will be processed until it is completed. In the same way the Gittins indices for jobs 2 and 3 at time zero can be computed.

$$G_2(0) = \max\Big(\frac{10}{0.5}, \frac{11.25}{0.5+0.083}, \frac{11.875}{0.5+0.083+0.021}\Big) = 20$$

The computation of the Gittins index of job 2 indicates that job 2, if selected to go on the machine at time 0, may be preempted before being completed; processing is only guaranteed for one time unit.

$$G_3(0) = \max\Big(\frac{10}{0.5}, \frac{12.5}{0.5+0.125}, \frac{13.75}{0.5+0.125+0.031}\Big) = 20.96$$

If job 3 would be selected for processing at time 0, it would be processed up to completion.

After comparing the three Gittins indices for the three jobs at time zero a decision can be made with regard to the job to be selected for processing. The maximum of the three Gittins indices is 20.96. So job 3 is put on the machine at time 0 and is kept on the machine until completion. At the completion of job 3 either job 1 or job 2 may be selected. The values of their Gittins indices are the same. If job 1 is selected for processing it remains on the machine until it is completed. If job 2 is selected, it is guaranteed processing for only one time unit; if it is not completed after one time unit it is preempted and job 1 is selected for processing. ||

What would happen if the processing times have ICR distributions? It can be shown that in this case the scheduler never will preempt. The Gittins index of the job being processed increases continuously, while the indices of the jobs waiting for processing remain the same. Consider the limiting case where the length of the time unit goes to 0 as the number of timesteps increases accordingly. The problem becomes a continuous time problem. When the processing times are ICR, the result in Theorem 10.2.1 is equivalent to the result in Theorem 10.1.3. So, in one sense Theorem 10.1.3 is more general as it covers the nonpreemptive setting with arbitrary processing time distributions (not just ICR distributions), while Theorem 10.2.1 does not give any indication of the form of the optimal policy in a *nonpreemptive* setting when the processing times are *not* ICR. In another sense Theorem 10.2.1 is more general, since Theorem 10.1.3 does not give any indication of the form of the optimal policy in a *preemptive* setting when the processing times are *not* ICR.

The result in Theorem 10.2.1 can be generalized to include jobs arriving according to a Poisson process. In a discrete time framework this implies that

the interarrival times are geometrically distributed with a fixed parameter. The job selected at any point in time is the job with the largest Gittins index among the jobs present. The proof of this result lies beyond the scope of this book.

The result can also be generalized to include breakdowns with up times that are i.i.d. geometrically distributed and down times that are i.i.d. arbitrarily distributed. For the proof of this result the reader is also referred to the literature.

10.3 Likelihood Ratio Ordered Distributions

Section 10.1 discussed examples of nonpreemptive stochastic models that are basically equivalent to their deterministic counterparts. In a number of cases the distributions of the random variables did not matter at all; only their expectations played a role. In this subsection, an example is given of a nonpreemptive stochastic model that is, to a lesser extent, equivalent to its deterministic counterpart. Its relationship with its deterministic counterpart is not as strong as in the earlier cases, since some conditions on the distribution functions of the processing times are required.

Consider n jobs. The processing time of job j is equal to the random variable X_j with distribution F_j, provided the job is started immediately at time zero. However, over time the machine "deteriorates", i.e., the later a job starts its processing, the longer its processing time. If job j starts with its processing at time t, its processing time is $X_j a(t)$, where $a(t)$ is an increasing concave function. Thus for any starting time t the processing time is proportional to the processing time of the job if it had started its processing at time 0. Moreover, concavity of $a(t)$ implies that the deterioration process in the early stages of the process is more severe than in the later stages of the process. The original processing times are assumed to be likelihood ratio ordered in such a way that $X_1 \leq_{lr} \cdots \leq_{lr} X_n$. The objective is to minimize the expected makespan. The following lemma is needed in the subsequent analysis.

Lemma 10.3.1. *If $g(x_1, x_2)$ is a real valued function satisfying*

$$g(x_1, x_2) \geq g(x_2, x_1)$$

for all $x_1 \leq x_2$, then

$$g(X_1, X_2) \geq_{st} g(X_2, X_1)$$

whenever

$$X_1 \leq_{lr} X_2.$$

Proof. Let $U = \max(X_1, X_2)$ and $V = \min(X_1, X_2)$. Condition on $U = u$ and $V = v$ with $u \geq v$. The conditional distribution of $g(X_1, X_2)$ is concentrated on the two points $g(u, v)$ and $g(v, u)$. The probability assigned to the smaller value $g(u, v)$ is then

$$P(X_1 = \max(X_1, X_2) \mid U = u, V = v)$$

$$= \frac{f_1(u)f_2(v)}{f_1(u)f_2(v) + f_1(v)f_2(u)}.$$

In the same way $g(X_2, X_1)$ is also concentrated on two points $g(u, v)$ and $g(v, u)$. The probability assigned to the smaller value $g(u, v)$ in this case is

$$P(X_2 = \max(X_1, X_2) \mid U = u, V = v)$$

$$= \frac{f_2(u)f_1(v)}{f_2(u)f_1(v) + f_2(v)f_1(u)}.$$

As $u \geq v$ and $X_2 \geq_{lr} X_1$,

$$f_1(u)f_2(v) \leq f_2(u)f_1(v).$$

Therefore, conditional on $U = u$ and $V = v$

$$g(X_2, X_1) \leq_{st} g(X_1, X_2).$$

Unconditioning completes the proof of the Lemma. □

At first sight this lemma may seem to provide a very fundamental and useful result. Any pairwise interchange in a deterministic setting can be translated into a pairwise interchange in a stochastic setting with the random variables likelihood ratio ordered. However, the usefulness of this lemma appears to be limited to single machine problems and proportionate flow shops.

The following two lemmas contain some elementary properties of the function $a(t)$.

Lemma 10.3.2. *If $0 < x_1 < x_2$, then for all $t \geq 0$*

$$x_1 a(t) + x_2 a\Big(t + x_1 a(t)\Big) \geq x_2 a(t) + x_1 a\Big(t + x_2 a(t)\Big).$$

Proof. The proof is easy and therefore omitted. □

From Lemmas 10.3.1 and 10.3.2 it immediately follows that if there are only two jobs, scheduling the job with the larger expected processing time first minimizes the expected makespan.

Lemma 10.3.3. *The function $h_{x_1}(t) = t + x_1 a(t)$ is increasing in t, for all $x_1 > 0$.*

Proof. The proof is easy and therefore omitted. □

Theorem 10.3.4. *The Longest Expected Processing Time first (LEPT) rule minimizes the expected makespan in the class of nonpreemptive static list policies as well as in the class of nonpreemptive dynamic policies.*

Proof. Consider first the class of nonpreemptive static list policies. The proof is by induction. It has already been shown to hold for two jobs. Assume the theorem holds for $n-1$ jobs. Consider any nonpreemptive static list policy and let job k be the job that is scheduled last. From Lemma 10.3.3 it follows that among all schedules that process job k last, the one resulting in the minimum makespan is the one that stochastically minimizes the completion time of the first $n-1$ jobs. Hence, by the induction hypothesis, of all schedules that schedule job k last, the one with the stochastically smallest makespan is the one which schedules the first $n-1$ jobs according to LEPT. If k is not the smallest job, then the best schedule is the one that selects the smallest job immediately before this last job k. Let t' denote the time that this smallest job starts its processing and suppose that there are only two jobs remaining to be processed. The problem at this point is a problem with two jobs and an a function that is given by $a_{t'}(t) = a(t'+t)$. Because this function is still concave it follows, from the result for two jobs, that interchanging these last two jobs reduces the total makespan stochastically. But among all schedules which schedule the smallest job last, the one stated in the theorem, by the induction hypothesis, minimizes the makespan stochastically. This completes the proof for nonpreemptive static list policies.

It remains to be shown that the LEPT rule also stochastically minimizes the makespan in the class of nonpreemptive dynamic policies. Suppose the decision is allowed to depend on what has previously occurred, at most l times during the process (of course, such times occur only when the machine is freed). When $l = 1$ it follows from the optimality proof for static list policies that it is optimal not to deviate from the LEPT schedule. If this remains true when $l-1$ such opportunities are allowed, it follows from the same argument that it remains true when l such opportunities are allowed (because of the induction hypothesis such an opportunity would be utilized only once). As the result is true for all l, the proof for nonpreemptive dynamic policies is complete. □

Example 10.3.5 (Linear Deterioration Function)

Consider two jobs with exponential processing times. The rates are λ_1 and λ_2. The deterioration function $a(t) = 1 + t$, $t \geq 0$, is linear. If the jobs are scheduled according to sequence $1, 2$, then the expected makespan can be computed as follows. If job 1 is completed at time t, the expected time job 2 will occupy the machine is $a(t)/\lambda_2$. The probability job 1 is completed during the interval $[t,\ \ t+dt]$ is $\lambda_1 e^{-\lambda_1 t} dt$. So

$$E(C_{\max}) = \int_0^\infty \Big(t + a(t)\frac{1}{\lambda_2}\Big)\lambda_1 e^{-\lambda_1 t} dt = \frac{1}{\lambda_1} + \frac{1}{\lambda_2} + \frac{1}{\lambda_1 \lambda_2}.$$

From this expression it is clear that the expected makespan in this case does *not* depend on the sequence. ||

Example 10.3.6 (Increasing Concave Deterioration Function and LEPT)

Consider two jobs with discrete processing time distributions.

$$P(X_1 = 1) = \frac{1}{8}, \quad P(X_1 = 2) = \frac{1}{4}, \quad P(X_1 = 3) = \frac{5}{8};$$

$$P(X_2 = 1) = \frac{1}{4}, \quad P(X_2 = 2) = \frac{1}{4}, \quad P(X_1 = 3) = \frac{1}{2}.$$

It is clear that $X_1 \geq_{lr} X_2$. $E(X_1) = 2.5$ while $E(X_2) = 2.25$. The deterioration function $a(t) = 1 + t$ for $0 \leq t \leq 2$, and $a(t) = 3$ for $t \geq 2$. Clearly, $a(t)$ is increasing concave. Consider the LEPT sequence, i.e., sequence $1, 2$. The expected makespan can be computed by conditioning on the processing time of the first job.

$$\begin{aligned} E(C_{\max}(LEPT)) &= \frac{1}{8}(1 + 2E(X_2)) + \frac{1}{4}(2 + 3E(X_2)) + \frac{5}{8}(3 + 3E(X_2)) \\ &= \frac{287}{32}, \end{aligned}$$

while

$$\begin{aligned} E(C_{\max}(SEPT)) &= \frac{1}{4}(1 + 2E(X_1)) + \frac{1}{4}(2 + 3E(X_1)) + \frac{1}{2}(3 + 3E(X_1)) \\ &= \frac{292}{32}. \end{aligned}$$

Clearly, LEPT is better than SEPT. ||

Intuitively, it does make sense that if the deterioration function is increasing concave the longest job should go first. It can also be shown that if the deterioration function is increasing convex the Shortest Expected Processing Time first (SEPT) rule is optimal and if the deterioration function is linear then any sequence is optimal.

However, if the function $a(t)$ is decreasing, i.e., a form of learning takes place that makes it possible to process the jobs faster, then it does not appear that similar results can be obtained.

10.4 Exponential Distributions

This section focuses on models with exponentially distributed processing times. Consider the stochastic version of $1 \mid d_j = d \mid \sum w_j U_j$ with job j having an exponentially distributed processing time with rate λ_j and a deterministic due date d. Recall that the deterministic counterpart is equivalent to the *knapsack* problem. The objective is the expected weighted number of tardy jobs.

Theorem 10.4.1. *The WSEPT rule minimizes the expected weighted number of tardy jobs in the classes of nonpreemptive static list policies, nonpreemptive dynamic policies and preemptive dynamic policies.*

Proof. First the optimality of the WSEPT rule in the class of nonpreemptive static list policies is shown. Assume the machine is free at some time t and two jobs, with weights w_1 and w_2 and processing times X_1 and X_2, remain to be processed. Consider first the sequence $1, 2$. The probability that both jobs are late is equal to the probability that X_1 is larger than $d - t$, which is equal to $\exp(-\lambda(d - t))$. The penalty for being late is then equal to $w_1 + w_2$. The probability that only the second job is late corresponds to the event where the processing time of the first job is $x_1 < d-t$ and the sum of the processing times $x_1 + x_2 > d - t$. Evaluation of the probability of this event, by conditioning on X_1 (that is $X_1 = x$), yields

$$P(X_1 < d - t,\ X_1 + X_2 > d - t) = \int_0^{d-t} e^{-\lambda_2(d-t-x)}\lambda_1 e^{-\lambda_1 x} dx.$$

If $E(\sum wU(1, 2))$ denotes the expected value of the penalty due to jobs 1 and 2, with job 1 processed first, then

$$E\Big(\sum wU(1,2)\Big) = (w_1 + w_2)e^{-\lambda_1(d-t)} + w_2 \int_0^{d-t} e^{-\lambda_2(d-t-x)}\lambda_1 e^{-\lambda_1 x} dx.$$

The value of the objective function under sequence $2, 1$ can be obtained by interchanging the subscripts in the expression above. Straightforward computation yields

$$E\Big(\sum wU(1,2)\Big) - E\Big(\sum wU(2,1)\Big) =$$

$$(\lambda_2 w_2 - \lambda_1 w_1)\frac{e^{-\lambda_1(d-t)} - e^{-\lambda_2(d-t)}}{\lambda_2 - \lambda_1}.$$

It immediately follows that the difference in the expected values is positive if and only if $\lambda_2 w_2 > \lambda_1 w_1$. Since this result holds for all values of d and t, any permutation schedule that does not sequence the jobs in decreasing order of $\lambda_j w_j$ can be improved by swapping two adjacent jobs, where the first has a lower λw value than the second. This completes the proof of optimality for the class of nonpreemptive static list policies.

Induction can be used to show optimality in the class of nonpreemptive dynamic policies. It is immediate that this is true for 2 jobs (it follows from the same pairwise interchange argument for optimality in the class of nonpreemptive static list policies). Assume that it is true for $n - 1$ jobs. In the case of n jobs this implies that the scheduler after the completion of the first job will, because of the induction hypothesis, revert to the WSEPT rule among the remaining $n - 1$ jobs. It remains to be shown that the scheduler has to select the job with the highest $\lambda_j w_j$ as the first one to be processed. Suppose the decision-maker

selects a job that does not have the highest $\lambda_j w_j$. Then, the job with the highest value of $\lambda_j w_j$ is processed second. Changing the sequence of the first two jobs decreases the expected value of the objective function according to the pairwise interchange argument used for the nonpreemptive static list policies.

To show that WSEPT is optimal in the class of preemptive dynamic policies, suppose a preemption is contemplated at some point in time. The remaining processing time of the job is then exponentially distributed with the same rate as it had at the start of its processing (because of the memoryless property of the exponential). Since the decision to put this job on the machine did not depend on the value of t at that moment or on the value of d, the same decision remains optimal at the moment a preemption is contemplated. A nonpreemptive policy is therefore optimal in the class of preemptive dynamic policies. □

This result is in a marked contrast to the result in Chapter 3 that states that its deterministic counterpart, i.e., the knapsack problem, is NP-hard.

Consider now the discrete time version of Theorem 10.4.1. That is, the processing time of job j is geometrically distributed with parameter q_j and job j has weight w_j. All jobs have the same due date d. If a job is completed exactly at its due date it is considered on time. The objective is again $E(\sum w_j U_j)$.

Theorem 10.4.2. *The WSEPT rule minimizes the expected weighted number of tardy jobs in the classes of nonpreemptive static list policies, nonpreemptive dynamic policies and preemptive dynamic policies.*

Proof. Consider two jobs, say jobs 1 and 2, and sequence $1, 2$. The probability that both jobs are late is q_1^{d+1} and the penalty is then $w_1 + w_2$. The probability that the first job is on time and the second job is late is

$$\sum_{t=0}^{d} (1 - q_1) q_1^t q_2^{d+1-t}.$$

The penalty is then w_2. So the total penalty under sequence $1, 2$ is

$$(w_1 + w_2) q_1^{d+1} + w_2 (1 - q_1) q_2^{d+1} \Big(1 - \Big(\frac{q_1}{q_2}\Big)^{d+1}\Big) \Big/ \Big(1 - \frac{q_1}{q_2}\Big)$$

The total expected penalty under sequence $2, 1$ can be obtained by interchanging the subscripts 1 and 2. Sequence $1, 2$ is better than sequence $2, 1$ if

$$w_1 q_1^{d+1} q_2 - w_2 q_1^{d+2} - w_2 q_1 q_2^{d+2} + w_2 q_1^{d+2} q_2 \\ \leq w_1 q_2^{d+2} - w_2 q_1 q_2^{d+1} + w_1 q_1^{d+2} q_2 - w_1 q_1 q_2^{d+2}.$$

After some manipulations it turns out that sequence $1, 2$ is better than $2, 1$ if

$$w_1 (1 - q_1)/q_1 \geq w_2 (1 - q_2)/q_2,$$

which is equivalent to

$$\frac{w_1}{E(X_1)} \geq \frac{w_2}{E(X_2)}.$$

That WSEPT is also optimal in the class of nonpreemptive dynamic policies and in the class of preemptive dynamic policies can be shown through the same arguments as those used in the proof of Theorem 10.4.1. □

The WSEPT rule does not necessarily yield an optimal schedule when processing time distributions are not all exponential (or all geometric).

Example 10.4.3 (Optimal Policy when Random Variables are ICR)

Consider the case where each one of the processing times is distributed according to an Erlang(k, λ) distribution. The rate of an exponential phase of job j is λ_j. This implies $E(X_j) = k/\lambda_j$. The WSEPT rule in general will not yield optimal schedules. Job j having a deterministic processing time p_j is a special case (the number of phases of the Erlang for each one of the n jobs approaches ∞, while the mean of each phase approaches zero). It is clear how in this manner a counterexample can be constructed for ICR processing times. ||

Example 10.4.4 (Optimal Policy when Random Variables are DCR)

Consider the case where each one of the processing times is distributed according to a mixture of exponentials. Assume that the processing time of job j is 0 with probability p_j and exponentially distributed with rate λ_j with probability $1 - p_j$. Clearly,

$$E(X_j) = (1 - p_j)\frac{1}{\lambda_j}.$$

The optimal preemptive policy can be determined easily. Try each job out at time zero for an infinitesimal period of time. The jobs with zero processing times are then immediately completed. Immediately after time zero it is known which jobs have nonzero processing times. The remaining processing times of these jobs are then exponentially distributed with probability one. The optimal preemptive policy from that point in time on is then the nonpreemptive policy described in Theorem 10.4.1. ||

Theorems 10.4.1 and 10.4.2 can be generalized to include breakdown and repair. Suppose the machine goes through "uptimes", when it is functioning and "downtimes" when it is being repaired. This breakdown and repair may form an arbitrary stochastic process. Theorem 10.4.1 also holds under these more general conditions since no part of the proof depends on the remaining time till the due date.

Theorem 10.4.1 can also be generalized to include different release dates with arbitrary distributions. Assume a finite number of releases after time 0, say n^*. It is clear from the results presented above that at the time of the last release

the WSEPT policy is optimal. This may actually imply that the last release causes a preemption (if, at that point in time, the job released is the job with the highest $\lambda_j w_j$ ratio in the system). Consider now the time epoch of the second last release. After this release a preemptive version of the WSEPT rule is optimal. To see this, disregard for a moment the very last release. All the jobs in the system at the time of the second to last release (*not* including the last release) have to be sequenced according to WSEPT; the last release may in a sense be considered a random "downtime". From the previous results it follows that all the jobs in the system at the time of the second last release should be scheduled according to preemptive WSEPT, independent of the time period during which the last release is processed. Proceeding inductively towards time zero it can be shown that a preemptive version of WSEPT is optimal with arbitrarily distributed releases in the classes of preemptive static list policies and preemptive dynamic policies.

The WSEPT rule proves optimal for other objectives as well. Consider the stochastic counterpart of $1 \mid d_j = d \mid \sum w_j T_j$ with job j again exponentially distributed with rate λ_j. All n jobs are released at time 0. The objective is to minimize the sum of the expected weighted tardinesses.

Theorem 10.4.5. *The WSEPT rule minimizes the expected sum of the weighted tardinesses in the classes of nonpreemptive static list policies, nonpreemptive dynamic policies and preemptive dynamic policies.*

Proof. The objective $w_j T_j$ can be approximated by the sum of an infinite sequence of $w_j U_j$ unit penalty functions, i.e.,

$$w_j T_j \approx \sum_{l=0}^{\infty} \frac{w_j}{K} U_{jl}.$$

The first unit penalty U_{j0} corresponds to a due date d, the second unit penalty U_{j1} corresponds to a due date $d + 1/K$, the third corresponds to a due date $d + 2/K$ and so on (see Figure 10.2). From Theorem 10.4.1 it follows that λw rule minimizes each one of these unit penalty functions. If the rule minimizes each one of these unit penalty functions, it also minimizes their sum. □

This theorem can be generalized along the lines of Theorem 10.4.1 to include arbitrary breakdown and repair processes and arbitrary release processes, provided all jobs have due date d (including those released after d).

Actually, a generalization in a slightly different direction is also possible. Consider the stochastic counterpart of the problem $1 \mid\mid \sum w_j h(C_j)$. In this model the jobs have no specific due dates, but are all subject to the *same* cost function h. The objective is to minimize $E(\sum w_j h(C_j))$. Clearly, $\sum w_j h(C_j)$ is a simple generalization of $\sum w_j T_j$ when all jobs have the same due date d. The function h can again be approximated by a sum of an infinite sequence of unit penalties, the only difference being that the due dates of the unit penalties are not necessarily equidistant as in the proof of Theorem 10.4.5.

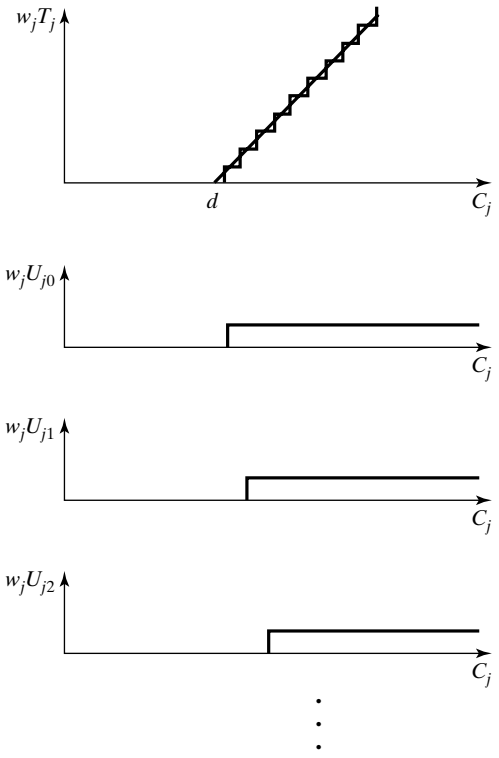


Fig. 10.2 Superposition of unit penalty functions

Consider now a stochastic counterpart of the problem $1 \mid\mid \sum w_j h_j(C_j)$, with each job having a *different* cost function. Again, all jobs are released at time 0. The objective is to minimize the total expected cost. The following ordering among cost functions is of interest: a cost function h_j is said to be *steeper* than a cost function h_k if

$$\frac{dh_j(t)}{dt} \geq \frac{dh_k(t)}{dt}$$

for every t, provided the derivatives exist. This ordering is denoted by $h_j \geq_s h_k$. If the functions are not differentiable for every t, the steepness ordering requires

$$h_j(t+\delta) - h_j(t) \geq h_k(t+\delta) - h_k(t),$$

for every t and δ. Note that a cost function being steeper than another does not necessarily imply that it is higher (see Figure 10.3).

Theorem 10.4.6. *If $\lambda_j w_j \geq \lambda_k w_k \iff h_j \geq_s h_k$, then the WSEPT rule minimizes the total expected cost in the classes of nonpreemptive static list policies, nonpreemptive dynamic policies and preemptive dynamic policies.*

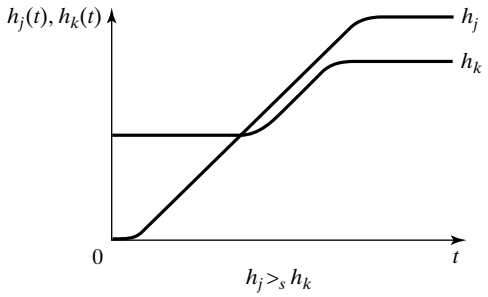


Fig. 10.3 One cost function being steeper than another

Proof. The proof follows from the fact that any increasing cost function can be approximated by an appropriate summation of a (possibly infinite) number of unit penalties at different due dates. If two cost functions, that may be at different levels, go up in the same way over an interval $[t_1, t_2]$, then a series of identical unit penalties go into effect within that interval for both jobs. It follows from Theorem 10.4.1 that the jobs have to be sequenced in decreasing order of λw in order to minimize the total expected penalties due to these unit penalties. If one cost function is steeper than another in a particular interval, then the steeper cost function has one or more unit penalties going into effect within this interval, which the other cost function has not. To minimize the total expected cost due to these unit penalties, the jobs have to be sequenced again in decreasing order of λw. □

Example 10.4.7 (Application of WSEPT when Due Dates are Agreeable)

Consider n jobs with exponentially distributed processing times with rates $\lambda_1, \ldots, \lambda_n$. The jobs have deterministic due dates $d_1 \leq d_2 \leq \cdots \leq d_n$. First, consider $E(\sum w_j T_j)$ as the objective to be minimized. If $\lambda_1 w_1 \geq \lambda_2 w_2 \geq \cdots \geq \lambda_n w_n$, then sequence $1, 2, \ldots, n$ minimizes the objective, since $T_1 \geq_s T_2 \geq_s \cdots \geq_s T_n$.

Second, consider $E(\sum w_j U_j)$ with the same due dates $d_1 \leq d_2 \leq \cdots \leq d_n$, as the objective to be minimized. It can be verified easily that the string of inequalities $U_1 \geq_s U_2 \geq_s \cdots \geq_s U_n$ does *not* hold. So sequence $1, 2, \ldots, n$ does not necessarily minimize the objective (see Exercise 10.10). ||

The result of Theorem 10.4.6 can be generalized easily to include an arbitrary machine breakdown process. It also can be extended, in the case of preemptive static list policies or in the case of preemptive dynamic policies, to include jobs with different release dates, as long as the cost functions of the new arrivals satisfy the stated "agreeability" conditions.

The results in this subsection indicate that scheduling problems with exponentially distributed processing times allow for more elegant structural results

than their deterministic counterparts. The deterministic counterparts of most of the models discussed in this section are NP-hard. It is intuitively acceptable that a deterministic problem may be NP-hard while its counterpart with exponentially distributed processing times allows for a very simple policy to be optimal. The reason is the following: all data being deterministic (i.e., perfect data) makes it very hard for the scheduler to optimize. In order to take advantage of all the information available the scheduler has to spend an inordinately amount of time doing the optimization. On the other hand when the processing times are stochastic, the data are fuzzier. The scheduler, with less data at hand, will spend less time performing the optimization. The fuzzier the data, the more likely a simple priority rule minimizes the objective in expectation. Expectation is akin to optimizing for the average case.

10.5 Discussion

This chapter has provided only a small sample of the results that have appeared in the literature with regard to single machine stochastic scheduling with all jobs released at time zero. Clearly, there are many more results in the literature.

For example, some research has focused on single machine stochastic scheduling with batch processing, i.e., stochastic counterparts of problems described in Chapter 4. In these stochastic models the batch size is assumed to be fixed (b) and either the expected makespan or the total expected completion time has to be minimized. It has been shown that if the n jobs are ordered according to symmetric variability ordering (i.e., the n jobs have the same mean but different variances), then the *Smallest Variance First* rule minimizes the expected makespan as well as the total expected completion time under fairly general conditions.

Exercises (Computational)

10.1. Consider a single machine and three jobs with i.i.d. processing times with distribution F and mean 1.

(a) Show that when F is deterministic $E(\sum C_j) = 6$ under a nonpreemptive schedule and $E(\sum C_j) = 9$ under the processor sharing schedule.

(b) Show that when F is exponential $E(\sum C_j) = 6$ under a nonpreemptive schedule and $E(\sum C_j) = 6$ under the processor sharing schedule. (Recall that under a processor sharing schedule all jobs available share the processor equally, i.e., if there are n jobs available, then each job receives $1/n$ of the processing capability of the machine (see Section 5.2)).

10.2. Consider the same scenario as in the previous exercise. Assume F is an EME distribution (as defined in Section 9.2) with the parameter p very small.

(a) Show that $E(\sum C_j) = 6$ under the nonpreemptive schedule.
(b) Show that $E(\sum C_j) = 3$ under the processor sharing schedule.

10.3. Consider a single machine and three jobs. The distribution of job j, $j = 1, 2, 3$, is discrete uniform over the set $\{10 - j, 10 - j + 1, \ldots, 10 + j\}$. Find the schedule(s) that minimize $E(\sum C_j)$ and compute the value of the objective function under the optimal schedule.

10.4. Consider the same setting as in the previous exercise. Find now the schedule that minimizes $E(\sum h_j(C_j))$, where the function $h(C_j)$ is defined as follows.

$$h(C_j) = \begin{cases} 0 & \text{if } C_j \leq 20 \\ C_j - 20 & \text{if } C_j > 20 \end{cases}$$

Is the *Largest Variance first (LV)* rule or the *Smallest Variance first (SV)* rule optimal?

10.5. Consider the same setting as in the previous exercise. Find now the schedule that minimizes $E(\sum h_j(C_j))$, where the function $h(C_j)$ is defined as follows.

$$h(C_j) = \begin{cases} C_j & \text{if } C_j \leq 20 \\ 20 & \text{if } C_j > 20 \end{cases}$$

Is the *Largest Variance first (LV)* rule or the *Smallest Variance first (SV)* rule optimal?

10.6. Consider two jobs with discrete processing time distributions:

$$P(X_1 = 1) = P(X_1 = 2) = P(X_1 = 4) = \frac{1}{3}$$

and

$$P(X_2 = 3) = P(X_2 = 5) = \frac{1}{2}.$$

The two jobs have deterministic due dates. The due date of the first job is $D_1 = 2$ and the due date of the second job is $D_2 = 4$. Compute $E(\max(L_1, L_2))$ and $\max(E(L_1), E(L_2))$ under EDD.

10.7. Consider the framework of Section 10.2. There are 3 jobs, all having a discrete uniform distribution. The processing time of job j is uniformly distributed over the set $\{5 - j, 5 - j + 1, \ldots, 5 + j - 1, 5 + j\}$. The discount factor β is equal to 0.5. The weight of job 1 is 30, the weight of job 2 is 10 and the weight of job 3 is 30. Find the optimal preemptive policy. Determine whether it is necessary to preempt any job at any point in time.

10.8. Redo the instance in Exercise 10.7 with the discount factor $\beta = 1$. Determine all optimal policies. Give an explanation for the results obtained and compare the results with the results obtained in Exercise 10.7.

10.9. Consider Example 10.3.5 with the linear deterioration function $a(t) = 1 + t$. Instead of the two jobs with exponentially distributed processing times, consider two jobs with geometrically distributed processing times with parameters q_1 and q_2. Compute the expected makespan under the two sequences.

10.10. Construct a counterexample for the stochastic problem with exponential processing times and deterministic due dates showing that if $\lambda_j w_j \geq \lambda_k w_k \Leftrightarrow d_j \leq d_k$ the λw rule does not necessarily minimize $E(\sum w_j U_j)$.

Exercises (Theory)

10.11. Consider the model in Theorem 10.1.1 with breakdowns. The up-times are exponentially distributed with rate ν and the down-times are i.i.d. (arbitrarily distributed) with mean $1/\mu$. Show that the expected time job j spends on the machine is equal to

$$E(Y_j) = \Big(1 + \frac{\nu}{\mu}\Big) E(X_j),$$

where $E(X_j)$ is the expected processing time of job j. Give an explanation why this problem is therefore equivalent to the problem without breakdowns (*Hint:* only the time-axis changes because of the breakdowns).

10.12. Consider the same model as in Exercise 10.11 but assume now that the processing time of job j is exponentially distributed with rate λ_j. Assume that the repair time is exponentially distributed with rate μ.

(a) Show that the number of times the machine breaks down during the processing of job j is geometrically distributed with rate

$$q = \frac{\nu}{\lambda_j + \nu}.$$

(b) Show that the total amount of time spent on the repair of the machine during the processing of job j is exponentially distributed with rate $\lambda_j \mu/\nu$, provided there is at least one breakdown.

(c) Show that the total time job j remains on the machine is a mixture of an exponential with rate λ_j and a convolution of two exponentials with rates λ_j and $\lambda_j \mu/\nu$. Find the mixing probabilities.

10.13. Consider the model in Exercise 10.11. Assume that the jobs are subject to precedence constraints that take the form of chains. Show that Algorithm 3.1.4 minimizes the total expected weighted completion time.

10.14. Consider the discrete time stochastic model described in Section 10.2. The continuous time version is a stochastic counterpart of the problem 1 |

$prmp \mid \sum w_j(1-\exp(-rC_j))$. Show that the Gittins index for this problem is

$$G_j(x_j) = \max_{\tau>0} \frac{w_j \int_{x_j}^{\tau} f_j(s)e^{-rs}ds}{\int_{x_j}^{\tau}(1-F_j(s))e^{-rs}ds}.$$

10.15. Consider the stochastic counterpart of $1 \mid d_j = d \mid \sum w_j U_j$ with the processing time of job j arbitrarily distributed according to F_j. All jobs have a common random due date that is exponentially distributed with rate r. Show that this problem is equivalent to the stochastic counterpart of the problem $1 \mid\mid \sum w_j(1-\exp(-rC_j))$ (that is, a problem without a due date but with a discounted cost function) with all jobs having arbitrary distributions. (*Hint:* If, in the stochastic counterpart of $1 \mid d_j = d \mid \sum w_j U_j$, job j is completed at time C_j the probability that it is late is equal to the probability that the random due date occurs before C_j. The probability that this occurs is $1-e^{-rC_j}$, which is equal to $E(U_j)$).

10.16. Show that if in the model of Section 10.3 the deterioration function is linear, i.e., $a(t) = c_1 + c_2 t$ with both c_1 and c_2 constant, the distribution of the makespan is sequence independent.

10.17. Show, through a counterexample, that LEPT does not necessarily minimize the makespan in the model of Section 10.3 when the distributions are merely ordered in expectation and not in the likelihood ratio sense. Find a counterexample with distributions that are stochastically ordered but not ordered in the likelihood ratio sense.

10.18. Consider the two processing time distributions of the jobs in Example 10.3.6. Assume the deterioration function $a(t) = 1$ for $0 \leq t \leq 1$ and $a(t) = t$ for $t \geq 1$ (i.e., the deterioration function is increasing convex). Show that SEPT minimizes the makespan.

10.19. Consider the discrete time counterparts of Theorems 10.4.3 and 10.4.4 with geometric processing time distributions. State the results and prove the optimality of the WSEPT rule.

10.20. Generalize the result presented in Theorem 10.4.6 to the case where the machine is subject to an arbitrary breakdown process.

10.21. Generalize Theorem 10.4.6 to include jobs which are released at different points in time.

10.22. Consider the following discrete time stochastic counterpart of the deterministic model $1 \mid d_j = d, prmp \mid \sum w_j U_j$. The n jobs have a common random due date D. When a job is completed before the due date, a discounted reward is obtained. When the due date occurs before its completion, no reward is obtained and it does not pay to continue processing the job. Formulate the optimal policy in the class of preemptive dynamic policies.

10.23. Show that if all weights are equal, i.e., $w_j = 1$ for all j, and $X_1 \leq_{st} X_2 \leq_{st} \cdots \leq_{st} X_n$, then the WDSEPT rule is equivalent to the SEPT rule for any r, $0 \leq r \leq 1$.

Comments and References

A number of researchers have considered nonpreemptive single machine scheduling problems with arbitrary processing time distributions, see Rothkopf (1966a, 1966b), Crabill and Maxwell (1969), Hodgson (1977) and Forst (1984). For results with regard to the WSEPT rule, or equivalently the $c\mu$ rule, see Cox and Smith (1961), Harrison (1975a, 1975b), Buyukkoc, Varaiya and Walrand (1985), and Nain, Tsoucas and Walrand (1989). For models that also include stochastic breakdowns, see Glazebrook (1984, 1987), Pinedo and Rammouz (1988), Birge, Frenk, Mittenthal and Rinnooy Kan (1990) and Frenk (1991).

The Gittins index is due to Gittins and explained in his famous paper Gittins (1979). Many researchers have subsequently studied the use of Gittins indices in single machine stochastic scheduling problems and other applications; see, for example, Whittle (1980, 1981), Glazebrook (1981a, 1981b, 1982), Chen and Katehakis (1986) and Katehakis and Veinott (1987). The proof of optimality of Gittins indices presented here is due to Weber (1992).

The section on processing time distributions that are likelihood ratio ordered and subject to deterioration is entirely based on the paper by Brown and Solomon (1973). For more results on single machine scheduling subject to deterioration, see Browne and Yechiali (1990).

For an extensive treatment of single machine scheduling with exponential processing time distributions, see Derman, Lieberman and Ross (1978), Pinedo (1983) and Pinedo and Rammouz (1988). For due date related objectives with processing time distributions that are not exponential, see Sarin, Steiner and Erel (1990).

For single machine stochastic scheduling with batch processing, see Koole and Righter (2001) and Pinedo (2007).

Chapter 11

Single Machine Models with Release Dates (Stochastic)

In many stochastic environments job releases occur at random points in time. This chapter focuses on single machine stochastic models with the jobs having besides random processing times also random release dates. The objective is the total expected weighted completion time. Preemptive as well as nonpreemptive models are considered.

An environment with random release dates is somewhat similar to the models considered in queueing theory. In a priority queue a server (or a machine) has to process customers (or jobs) from different classes with each class having it own priority level (or weight).

There are various similarities between stochastic scheduling with random release dates and priority queues. One similarity is that different jobs may have different processing times from different distributions. Another similarity is that different jobs may have different weights. However, there are also various differences. One important difference is that in scheduling the goal is typically to minimize an objective that involves n jobs, whereas in queueing one usually assumes an infinite stream of customers and the focus is on asymptotic results. In scheduling the goal is to find a policy that minimizes the total expected waiting cost of the n jobs, or, equivalently, the average expected waiting cost

M.L. Pinedo, *Scheduling*, DOI: 10.1007/978-0-387-78935-4_11,

of the n jobs, whereas in queueing the goal is to quantify the expected waiting time of a typical customer or customer class in steady state and then determine the policy that minimizes the average expected waiting cost per customer or customer class. It pays to draw parallels between stochastic scheduling and priority queues since certain approaches and methodologies are applicable to both areas of research.

The models considered in this chapter are the stochastic counterparts of $1 \mid r_j \mid \sum w_j C_j$ and $1 \mid r_j, prmp \mid \sum w_j C_j$. The objective considered is actually not $E(\sum w_j C_j)$, but rather

$$E(\sum_{j=1}^{n} w_j (C_j - R_j)).$$

However, the term $E(\sum w_j R_j)$ is, of course, a constant that does not depend on the policy. An equivalent objective is

$$E\Big(\frac{\sum_{j=1}^{n} w_j (C_j - R_j)}{n}\Big).$$

If there is an infinite number of customers, then the objective

$$\lim_{n\to\infty} E\Big(\frac{\sum_{j=1}^{n} w_j (C_j - R_j)}{n}\Big)$$

is of interest. This last objective is the one typically considered in queueing theory.

The WSEPT rule is optimal in several settings. This chapter focuses on the various conditions under which WSEPT minimizes the objectives under consideration.

11.1 Arbitrary Release Dates and Arbitrary Processing Times without Preemptions

The model considered in this section is in one sense more general and in another sense more restricted than the model described in Section 9.1. The generalization lies in the fact that now the jobs have different release dates. The restriction lies in the fact that in Section 9.1 the n jobs have processing times that come from n different distributions, whereas in this section there are only two job classes with two different distributions. The processing times of the two job classes are arbitrarily distributed according to F_1 and F_2 with means $1/\lambda_1$ and $1/\lambda_2$. The weights of the two job classes are w_1 and w_2, respectively. The release dates of the n jobs have an arbitrary joint distribution. Assume that unforced idleness is not allowed; that is, the machine is not allowed to remain idle if there

are jobs waiting for processing. Preemptions are not allowed. This model is a stochastic counterpart of $1 \mid r_j \mid \sum w_j C_j$, or, equivalently, $1 \mid r_j \mid \sum w_j(C_j - r_j)$.

Theorem 11.1.1. *Under the optimal nonpreemptive dynamic policy the decision-maker follows, whenever the machine is freed, the WSEPT rule.*

Proof. The proof is by contradiction and based on a simple adjacent pairwise interchange. Suppose that at a time when the machine is freed jobs from both priority classes are waiting for processing. Suppose the decision-maker starts a job of the lower priority class (even though a job of the higher priority class is available for processing); he schedules a job of the higher priority class immediately after the completion of the job of the lower priority class. Now perform an adjacent pairwise interchange between these two jobs. Note that a pairwise interchange between these two adjacent jobs does not affect the completion times of any one of the jobs processed after this pair of jobs. However, the pairwise interchange does reduce the sum of the weighted expected completion times of the two jobs involved in the interchange. So the original ordering could not have been optimal. It follows that the decision-maker always must use the WSEPT rule. □

The result of Theorem 11.1.1 applies to settings with a finite number of jobs as well as to settings with an infinite arrival stream. The result cannot be generalized to more than two priority classes; with three priority classes a counterexample can be found easily.

Example 11.1.2 (Counterexample to Optimality of WSEPT with three Priority Classes)

The following counterexample has three jobs and is entirely deterministic.

jobs	1	2	3
p_j	1	4	1
r_j	0	0	1
w_j	1	5	100

At time zero the job with the highest w_j/p_j ratio is job 2. However, under the optimal schedule job 1 has to be processed at time 0. After job 1 has been completed at time 1, job 3 starts its processing. Under this schedule the total weighted completion time is

$$1 + 1 \times 100 + 6 \times 5 = 131.$$

If job 2 would have started its processing at time zero, then the total weighted completion time would be

$$4 \times 5 + 4 \times 100 + 6 \times 1 = 426. \qquad ||$$

The proof of Theorem 11.1.1, for two priority classes, does not go through when unforced idleness is allowed. If unforced idleness is allowed, then it may be optimal to keep the machine idle while a job is waiting, in anticipation of an imminent release of a high priority job.

Example 11.1.3 (Counterexample to Optimality of WSEPT when Unforced Idleness is Allowed)

The following counterexample with two jobs is also deterministic.

jobs	1	2
p_j	4	1
r_j	0	1
w_j	1	100

At time 0 there is a job available for processing. However, it is optimal to keep the machine idle till time 1, process job 2 for one time unit and then process job 1. Under this optimal schedule the total weighted completion time is

$$1 \times 100 + 6 \times 1 = 106.$$

If job 1 would have been put on the machine at time 0, then the total weighted completion time is

$$4 \times 1 + 5 \times 100 = 504. \qquad ||$$

11.2 Priority Queues, Work Conservation and Poisson Releases

Assume that at the release of job j, at time R_j, the processing time X_j is drawn from distribution F_j. This implies that at any time t the total amount of processing required by the jobs waiting for processing (or, in queueing terminology, the customers waiting in queue), has already been determined. Let $x^r(t)$ denote the remaining processing time of the job that is being processed on the machine at time t. Let $V(t)$ denote the sum of the processing times of the jobs waiting for processing at time t plus $x^r(t)$. In the queueing literature this $V(t)$ is typically referred to as the amount of work that is present in the system at time t.

At each release date the $V(t)$ jumps (increases), and the size of the jump is the processing time of the job just released. Between jumps, $V(t)$ decreases continuously with slope -1, as long as the machine is busy processing a job. A realization of $V(t)$ is depicted in Figure 11.1. As long as unforced idleness of the machine is not allowed, the function $V(t)$ does not depend on the priorities

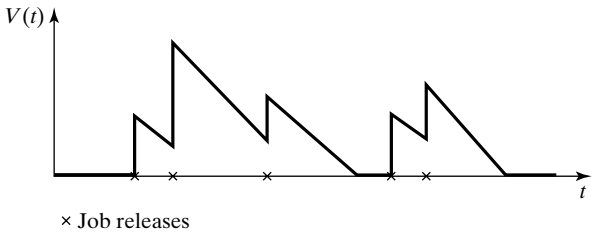


Fig. 11.1 Amount of work in system as function of time

of the different job classes nor on the sequence in which the jobs are processed on the machine.

Closed form expressions for the performance measures of interest, e.g., the expected time a typical job spends in the system under a given priority rule, can only be obtained under certain assumptions regarding the release times of the jobs. The release processes considered are similar to those typically used in queueing theory.

Suppose there is a single class of jobs and the jobs have processing times that are i.i.d. and distributed according to F. There is an infinite stream of jobs coming in. The jobs are released according to a Poisson process with rate ν, implying that the probability of the number of jobs released by time t, $N(t)$, equals ℓ is

$$P(N(t) = \ell) = \frac{e^{-\nu t}(\nu t)^{\ell}}{\ell!}.$$

The release times of the jobs are, of course, strongly dependent upon one another. The release of any given job occurs a random time after the release of the previous job. Successive interrelease times are independent and exponentially distributed with the same mean.

Poisson release processes have a very useful and important property, that in queueing theory often is referred to as *Poisson Arrivals See Time Averages (PASTA)*. Consider a single class of jobs that are released according to a Poisson process. The PASTA property implies that an arbitrary job, at its release, encounters an expected number of jobs waiting for processing that is equal to the average number of jobs waiting for processing at any other time that is selected at random. It also implies that the expected number of jobs an arbitrary release finds being processed on the machine is equal to the average number of jobs being processed at any other time selected at random (note that the number of jobs being processed is a $0-1$ random variable; so the expected number being processed is equal to the probability that the machine is processing a job). It implies also that an arbitrary job finds, at its release, an expected amount of work waiting in queue that is equal to the expected amount of work waiting at any other time that is selected at random. The PASTA property is an important characteristic of the Poisson process.

Example 11.2.1 (The PASTA Property)

Consider a Poisson release process with the expected time between consecutive releases equal to ten minutes, i.e., $1/\nu = 10$ minutes. Assume the processing times are deterministic and equal to four minutes. The expected number of jobs in process, i.e., the time average, is 0.4. The PASTA property implies that this number is equal to the average number of jobs that a new release finds in service. So a Poisson release finds the machine busy with probability 0.4.

Consider now a release process that is not Poisson. A new job is released exactly every ten minutes, i.e., with deterministic interrelease times. The processing time of each job is again exactly four minutes (deterministic). The time average of the number of jobs in the queue is 0 and the time average of the number of jobs in service is 0.4, since 40% of the time a job is being processed and 60% of the time the machine is idle. Since the release process as well as the service times are deterministic, the number of jobs each new release finds in queue is 0. So this average number in queue seen by a new release happens to be equal to the time average (since both averages are zero). However, the average number of jobs that a new release finds in service is also 0 whereas the time average of the number of jobs in service is 0.4; so the average number in service seen by a release is not equal to the time average. ||

The operating characteristics of a machine subject to Poisson releases can be analyzed within the following framework. Suppose that in each time unit every job in the system (either in queue or in service) has to pay \$1.00 for every unit of processing time it still needs. So if a job needs processing for x time units, then for each unit of time it is waiting in queue it has to pay x dollars. However, while in service, the job's payout rate steadily declines as its remaining processing time decreases.

The average rate at which the system is earning its money is the rate at which the jobs are released, ν, times the expected amount paid by a job. However, the rate at which the system is earning money is also equal to the expected amount of work in the system at a random point in time. Let the random variable V denote the amount of work in the system in steady state. So,

$$E(V) = \lim_{t \to \infty} E(V(t))$$

and

$$E(V) = \nu E(\text{amount paid by a job}).$$

Let W^q and X denote the amount of time the job spends waiting in queue and the amount of time being processed. So the total time between release and completion is

$$W^s = W^q + X.$$

Then, since the job pays at a constant rate of X per unit time while it waits in queue and at a rate $X - x$ after having spent an amount of time x in service,

$$E(\text{amount paid by a job}) = E\Big(XW^q + \int_0^X (X - x)dx\Big).$$

So

$$E(V) = \nu E(XW^q) + \frac{\nu E(X^2)}{2}.$$

In addition, if a job's wait in queue is independent of its processing time (this is the case when the priority rule is *not* a function of the processing time), then

$$E(V) = \nu E(X)E(W^q) + \frac{\nu E(X^2)}{2}.$$

In a queue that operates according to First Come First Served, a job's wait in queue is equal to the work in the system at its release. So $E(W^q)$ is equal to the average work seen by a new release. If the release process is Poisson, then (because of PASTA)

$$E(W^q) = E(V).$$

This identity, in conjunction with the identity

$$E(V) = \nu E(X)E(W^q) + \frac{\nu E(X^2)}{2}$$

yields

$$E(W^q) = \frac{\nu E(X^2)}{2(1 - \nu E(X))},$$

where $E(X)$ and $E(X^2)$ are the first two moments of the processing time distributions. This last expression is well-known in queueing theory and often referred to as the Pollaczek-Khintchine formula; it is the expected time in an M/G/1 queue.

Poisson releases of jobs make it also possible to derive a closed form expression for the expected length of a busy period $E(B)$ and the expected length of an idle period $E(I)$ of the machine. The utilization rate of the machine can be expressed as

$$\frac{E(B)}{E(I) + E(B)} = \frac{\nu}{\lambda}.$$

So

$$E(B) = \frac{\nu E(I)}{\lambda - \nu}.$$

From the fact that the idle times are exponentially distributed, it follows that $E(I) = 1/\nu$ and

$$E(B) = \frac{1}{\lambda - \nu}.$$

The approach and analysis described above can be applied also to a setting with a number of different job classes. Let $x_k^r(t)$ denote the remaining processing time of the job that is being processed on the machine at time t, if the job is of class k, $k = 1, \ldots, n$. Let $V_k(t)$ denote the sum of the processing times of all jobs of class k, $k = 1, \ldots, n$, that are waiting for processing at time t plus $x_k^r(t)$. So

$$V(t) = \sum_{k=1}^{n} V_k(t).$$

If the classes of jobs are ordered according to a given priority list, then an interchange of priorities between two classes that are adjacent on the list, in the preemptive case, does not affect the $V_k(t)$ of any one of the classes with a higher priority nor does it affect the $V_k(t)$ of any one of the classes with a lower priority. Combining adjacent classes of priorities into a single class, has no effect on either the $V_k(t)$ of a class with a lower priority or the $V_k(t)$ of a class with a higher priority under any type of policy.

Closed form expressions for performance measures such as the expected time a typical job of a given class spends in system under a given priority rule, can again only be obtained under certain assumptions regarding the release times of the jobs.

Suppose there are c different classes of jobs and jobs from class k, $k = 1, \ldots, c$, have processing times that are i.i.d. and distributed according to F_k. There are c infinite streams of job releases; jobs from class k are released according to a Poisson process with rate ν_k, implying that the probability of the number of class k jobs released by time t, $N_k(t)$, is ℓ is

$$P(N_k(t) = \ell) = \frac{e^{-\nu_k t}(\nu_k t)^\ell}{\ell!}.$$

Release times of jobs from different classes are independent from one another, but release times of jobs from the same class are, of course, strongly dependent upon one another. The release of a given job from any given class occurs a random time (exponentially distributed) after the release of the previous job of that class.

11.3 Arbitrary Releases and Exponential Processing Times with Preemptions

Consider a machine that is subject to random releases of n jobs. The processing time of job j is X_j, exponentially distributed with rate λ_j. The weight of job j is w_j and its release date is R_j. The n release dates may have an arbitrary joint distribution; they may either be independent or have some form of dependence. If at the release of a new job the machine is processing a job of lower priority, then the job just released is allowed to preempt the job being processed. The

goal is to find the optimal preemptive dynamic policy that minimizes the total expected weighted completion time.

The model considered is a stochastic counterpart of the deterministic model $1 \mid r_j, prmp \mid \sum w_j C_j$ (which is known to be strongly NP-Hard). The model is also closely related to the so-called G/M/1 priority queue; i.e., a single server with exponential processing times subject to an arbitrary arrival process that is not necessarily renewal.

Theorem 11.3.1. *The optimal preemptive dynamic policy is the preemptive WSEPT rule. That is, at any point in time, among the jobs available for processing, the one with the highest $\lambda_j w_j$ must be processed.*

Proof. Consider a fixed (but arbitrary) time t. At time t it is known which jobs already have been released. Let q_j denote the probability that under a certain policy job j, which was released before time t, has not completed its processing yet. Let $E(V(t))$ denote the total expected time that the machine needs at time t to finish all the jobs that have been released prior to t. Note that, assuming that preemptions generate no additional work and that the machine is always kept busy when there are jobs waiting for processing, $E(V(t))$ is independent of the policy used.

Clearly, the amount of work still to be done increases at a release date by the amount of processing the newly released job requires, while at other instances the amount of work decreases linearly as long as there is still work to be done.

Because of the memoryless property of the exponential distribution, the expected remaining processing time at time t of the uncompleted job j is $1/\lambda_j$, independent of the amount of processing that job j has received prior to t. Therefore,

$$E(V(t)) = \sum_{j=1}^{n} \frac{q_j}{\lambda_j}.$$

Consider an arbitrary preemptive static list policy. Reverse the priorities between jobs k and l with adjacent priorities in this static list policy. It is easy to see why this reversal does not affect the waiting times of other jobs. Interchanging the priorities of jobs k and l does not affect the waiting times of higher priority jobs as these jobs have preemptive rights over both k and l. In the same way, jobs that have a lower priority than k and l do not depend upon how jobs k and l are processed.

The expected rate at which the objective function is increasing at time t is

$$w_k q_k + w_l q_l + \sum_{j \neq k,l} w_j q_j.$$

This rate has to be minimized. Because of work conservation,

$$\frac{q_k}{\lambda_k} + \frac{q_l}{\lambda_l} = E(V(t)) - \sum_{j \neq k,l} \frac{q_j}{\lambda_j}.$$

Rewriting this equation yields

$$q_l = \lambda_l \Big(E(V(t)) - \sum_{j \neq k,l} \frac{q_j}{\lambda_j} - \frac{q_k}{\lambda_k} \Big).$$

The expected rate at which the total cost function increases becomes now, after substitution,

$$q_k \Big(w_k - w_l \frac{\lambda_l}{\lambda_k} \Big) + \lambda_l w_l \Big(E(V(t)) - \sum_{j \neq k,l} \frac{q_j}{\lambda_j} \Big) + \sum_{j \neq k,l} w_j q_j .$$

Because reversing the priorities of jobs k and l only affects q_k and q_l, the last two terms of this expected rate do not change. The coefficient of q_k (the first term) is positive only when

$$\lambda_l w_l \leq \lambda_k w_k .$$

So when this inequality holds, the q_k should be kept small. This can only be achieved by giving job k a higher priority.

So any preemptive static list policy that is not in decreasing order of $\lambda_j w_j$ can be improved by a number of adjacent pairwise interchanges. This proves that the preemptive WSEPT policy of the theorem is optimal in the class of preemptive static list policies. The policy is optimal in this class for all possible realizations of release dates.

That this policy is also optimal in the class of preemptive dynamic policies can be shown as follows. Consider first the class of dynamic policies that only allow preemption at the release of a new job. That the policy is optimal in this class of policies can be shown by induction on the number of jobs still to be released. When zero jobs remain to be released the policy is clearly optimal. Suppose that only one job remains to be released and before the release of this last job (but after the release of the second last job) a job is processed that does not have the highest ratio of $\lambda_j w_j$ among the jobs available for processing. By performing pairwise interchanges among the jobs during this time interval, the expected rate of increase of the objective function can be reduced for all time epochs after this interval. Proceeding inductively it can be shown that when n jobs remain to be released the policy of the theorem is optimal within this particular class of preemptive dynamic policies.

Consider now the class of dynamic policies with preemptions also allowed at time epochs $\delta,\ 2\delta,\ 3\delta,\ \ldots$ That the policy of the theorem is also optimal in this class of preemptive dynamic policies can be shown by induction as well. Consider the last time interval when an action is taken that does not conform to the stated policy. Then a pairwise switch of jobs during this interval decreases the cost rate for any time epoch after this interval. An inductive argument shows that the policy of the theorem has to be adopted throughout. A continuity argument completes the proof of the theorem. □

The theorem above applies also to a queueing system subject to an arbitrary input process and exponential service times, i.e., the G/M/1 queue. Closed form expressions for the expected waiting times in a G/M/1 queue can only be obtained for certain specific interrelease time distributions. The following example illustrates how the expected waiting time can be computed when the machine is subject to a stationary input process.

Example 11.3.2 (Arbitrary Releases and Exponential Processing Times with Preemptions)

Consider a machine with two job classes. Both classes have the same exponential processing time distribution with a mean of 2 minutes, i.e., $\lambda = 0.5$ jobs per minute. One class arrives at regular (fixed) intervals of 8 minutes each. The other class also arrives at regular (fixed) intervals of 8 minutes each. The two classes are synchronized in such a way that every 4 minutes a new job is released. It is easy to determine the expected waiting time of a Class 1 job and a Class 2 job in a D/M/1 queue. The expected waiting times of the two job classes separately and the two job classes together can either be determined analytically or can be found in tables. From the tables available in the literature it can be established that $E(W_1^q) = 0.04$ minutes, $E(W_2^q) = 0.98$ minutes, and $E(W^q) = 0.51$ minutes.

Consider, for comparison purposes, the following scenario with the same two job classes. The two job classes are now released according to two independent Poisson processes with mean interrelease times $1/\nu = 8$ minutes. It is easy to determine the expected waiting time of a Class 1 job and a Class 2 job in such an M/M/1 queue. Class 1 jobs have preemptive priority over Class 2 jobs. So for Class 1 jobs Class 2 jobs do not even exist. Analyzing the waiting times of the Class 1 jobs as an M/M/1 queue yields

$$E(W_1^q) = \frac{\nu}{\lambda(\lambda - \nu)} = \frac{0.125}{0.5(0.5 - 0.125)} = 0.6667.$$

The total expected waiting time of all jobs can also be analyzed as an M/M/1 queue. The total expected waiting time is

$$E(W^q) = \frac{0.25}{0.5(0.5 - 0.25)} = 2.$$

From the fact that

$$E(W^q) = 0.5E(W_1^q) + 0.5E(W_2^q),$$

it follows that $E(W_2^q) = 3.33$ min.

Note that the time a Class 2 job spends in queue may not be just a single uninterrupted period. The processing of a Class 2 job may be interrupted repeatedly by arrivals of Class 1 jobs.

Comparing exponential interrelease times to deterministic interrelease times shows that the expected waiting time of each job class is significantly shorter when interrelease times are deterministic. ||

In the example above the two job classes have the same processing time distribution. The optimality of the WSEPT rule holds also when job classes have different processing time distributions. But, in general it is not easy to find closed form expressions for the total expected waiting time of each class when the classes have different distributions. However, it is still fairly easy when jobs are released according to Poisson processes.

Example 11.3.3 (Poisson Releases and Exponential Processing Times with Preemptions)

Consider two job classes. Jobs of Class 1 are released according to a Poisson process with rate $\nu_1 = 1/8$, i.e., the mean interrelease time is 8 minutes. The distribution of the processing times of Class 1 jobs is exponential with mean and rate 1, i.e., $\lambda_1 = 1$. The weight of Class 1 jobs is $w_1 = 1$. Class 2 jobs are also released according to a Poisson process with rate $\nu_2 = 1/8$. Their processing times are exponentially distributed with rate $\lambda_2 = 1/3$. Their weight is $w_2 = 3$. Because the $\lambda_j w_j$ values of the two classes are the same both priority orderings have to be optimal. The fact that the release processes are Poisson makes it possible to obtain closed form expressions for all performance measures.

Consider first the case where the Class 1 jobs have a higher priority than the Class 2 jobs. To compute the expected waiting time of the Class 1 jobs is relatively easy. Since these jobs have a preemptive priority over the Class 2 jobs, the Class 2 jobs do not have any impact on the waiting times of the Class 1 jobs. So these waiting times can be computed by just considering an M/M/1 queue with $\nu_1 = 1/8$ and $\lambda_1 = 1$. From some elementary results in queueing theory it follows that

$$E(W_1^q) = \frac{\nu_1}{\lambda_1(\lambda_1 - \nu_1)} = \frac{0.125}{1(1 - 0.125)} = 1/7 = 0.1427.$$

and

$$E(W_1^s) = E(W_1^q) + \frac{1}{\lambda_1} = \frac{8}{7} = 1.1427.$$

The expected waiting time of a Class 2 job is harder to compute. From queueing theory it follows that an arbitrary arrival finds, upon its arrival at the queue, an amount of work already waiting for the machine that is equal to the expected waiting time in an M/G/1 queue with Poisson arrivals at a rate $\nu = \nu_1 + \nu_2$, and a service time distribution that is a mixture of two exponentials with rates 1 and 1/3 and with mixing probabilities of 0.5 and 0.5. An arbitrary release of a Class 2 job may find upon arrival a queue that contains jobs from both classes. Such an arbitrary arrival will have to wait at least for all these jobs to be completed before it can even be considered

for processing. This wait will be equal to the expected wait in an M/G/1 queue and is known to be

$$\frac{\nu E(X^2)}{2(1-\nu E(X))} = \frac{0.25\Big(0.5(2/\lambda_1^2) + 0.5(2/\lambda_2^2)\Big)}{2\Big(1 - 0.25\Big(0.5(1/\lambda_1) + 0.5(1/\lambda_2)\Big)\Big)} = 2.5.$$

So the expected total time a Class 2 job remains in system must be at least 2.5 plus its own processing time which is $1/\lambda_2 = 3$ minutes. However, it is possible that during these 5.5 minutes additional Class 1 jobs are released. These Class 1 jobs have a preemptive priority over the Class 2 job under consideration. So these Class 1 jobs that arrive during these 5.5 minutes have to be taken into account as well. The expected number of Class 1 jobs released during this period is $\nu_1 \times 5.5 = 11/16$. How much additional wait do these releases cause? It is actually more than their own processing times. Because, while these Class 1 jobs are being processed additional Class 1 jobs may be released. So instead of looking at just 11/16 jobs released, one has to look at the busy periods (consisting of only Class 1 jobs) that each one of these releases generate. The busy period that one such job may generate can be analyzed by computing the busy period in an M/M/1 queue with arrival rate $\nu_1 = 1/8$ and service rate $\lambda_1 = 1$. From elementary queueing theory it follows that the expected length of such a busy period generated by a single job is

$$\frac{1}{(\lambda_1 - \nu_1)} = \frac{8}{7}$$

So the total expected time that an arbitrary Class 2 job spends in the system before it completes its processing is

$$E(W_2^s) = \frac{11}{2} + \frac{11}{16} \times \frac{8}{7} = \frac{11}{14} = \frac{44}{7} = 6.28.$$

Therefore,

$$E(W_2^q) = E(W_2^s) - \frac{1}{\lambda_2} = 3.28.$$

The total objective is

$$\sum_{k=1}^{n} w_k \frac{\nu_k}{\nu} E(W_k^q) = 1 \times \frac{1}{2} \times \frac{1}{7} + 3 \times \frac{1}{2} \times \frac{23}{7} = 5.$$

Consider now the same environment with the priority rule reversed: Class 2 jobs have a preemptive priority over Class 1 jobs. Now the performance measure with regard to Class 2 jobs is easy. They can be viewed as a simple M/M/1 queue with $\nu = 1/8$ and $\lambda = 1/3$.

$$E(W_2^q) = \frac{\nu}{\lambda(\lambda - \nu)} = \frac{0.125}{0.33(0.33 - 0.125)} = \frac{9}{5} = 1.8$$

The computation of the expected waiting time of a Class 1 job, which has a lower priority, is now harder. The procedure is the same as the one followed for computing the expected waiting time of a Class 2 job when it has the lower priority. Consider again an arbitrary release of a Class 1 job. Upon its release it finds an amount of work waiting in queue that is identical to what an arbitrary release of Class 1 finds. This amount was computed before and is equal to 2.5. So the expected total time that a Class 1 job has to remain in the system is at least $2.5 + 1/\lambda_1 = 3.5$. However, during this time there may be additional releases of Class 2 jobs that all have a preemptive priority over the Class 1 job under consideration. The expected number of releases of Class 2 jobs over this period is $3.5 \times \nu_2 = 7/16$. However, each one of these Class 2 jobs generates a busy period of Class 2 jobs which all have priority over the Class 1 job under consideration. The expected length of a Class 2 job busy period is

$$\frac{1}{(\lambda_2 - \nu_2)} = \frac{24}{5}.$$

So the expected time an arbitrary Class 1 job spends in system before it is completed is

$$E(W_1^s) = \frac{7}{2} + \frac{7}{16} \times \frac{24}{5} = \frac{56}{10} = 5.6.$$

Therefore,

$$E(W_1^q) = E(W_1^s) - \frac{1}{\lambda_1} = 4.6.$$

The total objective is

$$\sum_{k=1}^{n} w_k \frac{\nu_k}{\nu} E(W_k^q) = 1 \times \frac{1}{2} \times \frac{46}{10} + 3 \times \frac{1}{2} \times \frac{9}{5} = 5.$$

As expected, the values of the objective function under the two orderings are the same. ||

11.4 Poisson Releases and Arbitrary Processing Times without Preemptions

In contrast to the assumptions in the previous section, preemptions are not allowed in this section. The results in this section apply to a stochastic counterpart of the deterministic model $1 \mid r_j \mid \sum w_j C_j$.

The release times of the jobs in the previous sections were assumed to be completely arbitrary. The release time of any one job could be completely independent of the release time of any other job. The release time of job j could be, for example, an arbitrarily distributed random variable R_j and this time

could be measured from time 0. The release processes considered in the previous sections could also allow for any form of stochastic dependency between the release times of the various jobs. The results in these sections hold when there is a finite set of n jobs as well as when there is an infinite arrival stream. They are therefore valid in a steady state as well as in a transient state. In contrast to the results in the previous sections, the results in this section apply only to steady state; the proof of optimality of the WSEPT rule presented in this section is only valid for the steady state case.

Consider two priority classes with release rates ν_1 and ν_2. The two release processes are independent Poisson. The processing time distributions are G_1 and G_2. Class 1 jobs have nonpreemptive priority over Class 2 jobs. Let $E(W_1^q)$ and $E(W_2^q)$ denote the average wait in queue of a Class 1 job and a Class 2 job, respectively.

Note that the total work in system at any time does not depend on the priority rule as long as the machine is always busy when there are jobs waiting for processing. So the work in the system is the same as it would be if the machine was processing the jobs according to the First Come First Served rule. Under the FCFS rule, the system is equivalent to a single class system subject to a Poisson release process with rate $\nu = \nu_1 + \nu_2$ and a processing time distribution

$$G(x) = \frac{\nu_1}{\nu} G_1(x) + \frac{\nu_2}{\nu} G_2(x),$$

which follows from the fact that a combination of two independent Poisson processes is also Poisson with a rate that is the sum of the rates of the two underlying processes. The processing time distribution G can be obtained by conditioning on which priority class the job is from. It follows that the average amount of work in the system is

$$\begin{aligned} E(V) &= \frac{\nu E(X^2)}{2(1-\nu E(X))}, \\ &= \frac{\nu\left(\frac{\nu_1}{\nu}E(X_1^2) + \frac{\nu_2}{\nu}E(X_2^2)\right)}{2\left(1-\nu\left(\frac{\nu_1}{\nu}E(X_1) + \frac{\nu_2}{\nu}E(X_2)\right)\right)} \\ &= \frac{\nu_1 E(X_1^2) + \nu_2 E(X_2^2)}{2(1-\nu_1 E(X_1) - \nu_2 E(X_2))} \end{aligned}$$

In order to simplify the notation, let

$$\rho_k = \nu_k E(X_k).$$

So

$$E(V) = \frac{\nu_1 E(X_1^2) + \nu_2 E(X_2^2)}{2(1-\rho_1-\rho_2)}.$$

Note that X and W^q, i.e., the processing time and the time waiting for processing, of any given job are not independent of one another. Any information with regard to the realization of the processing time of a job may give an indication of the class to which the job belongs and therefore also an indication of its expected waiting time.

Let V_1 and V_2 denote the amount of work of Classes 1 and 2 in the system.

$$\begin{aligned} E(V_i) &= \nu_i E(X_i) E(W_i^q) + \frac{\nu_i E(X_i^2)}{2} \\ &= \rho_i E(W_i^q) + \frac{\nu_i E(X_i^2)}{2}, \qquad i = 1, 2. \end{aligned}$$

Define

$$\begin{aligned} E(V_i^q) &= \rho_i E(W_i^q), \\ E(V_i^s) &= \nu_i E(X_i^2)/2. \end{aligned}$$

So one could interpret V_i^q as the average amount of work of Class i waiting in queue and V_i^s as the average amount of Class i work in service. To compute the average amount of waiting time of a Class 1 job ($E(W_1^q)$), consider an arbitrary job release of Class 1. Its wait in queue is the amount of Class 1 work in the system (in queue and in service) at its release plus the amount of Class 2 work in service at its release. Taking expectations and using the fact that Poisson Arrivals See Time Averages yields

$$\begin{aligned} E(W_1^q) &= E(V_1) + E(V_2^s) \\ &= \rho_1 E(W_1^q) + \nu_1 E(X_1^2)/2 + \nu_2 E(X_2^2)/2 \end{aligned}$$

or

$$E(W_1^q) = \frac{\nu_1 E(X_1^2) + \nu_2 E(X_2^2)}{2(1-\rho_1)}.$$

To obtain $E(W_2^q)$, first note that from the fact that $V = V_1 + V_2$ it follows that

$$\begin{aligned} \frac{\nu_1 E(X_1^2) + \nu_2 E(X_2^2)}{2(1-\rho_1-\rho_2)} &= \rho_1 E(W_1^q) + \rho_2 E(W_2^q) + \nu_1 E(X_1^2)/2 + \nu_2 E(X_2^2)/2 \\ &= E(W_1^q) + \rho_2 E(W_2^q). \end{aligned}$$

From the last three equations it follows that

$$\rho_2 E(W_2^q) = \frac{\nu_1 E(X_1^2) + \nu_2 E(X_2^2)}{2} \Big(\frac{1}{1-\rho_1-\rho_2} - \frac{1}{1-\rho_1} \Big)$$

or

$$E(W_2^q) = \frac{\nu_1 E(X_1^2) + \nu_2 E(X_2^2)}{2(1-\rho_1-\rho_2)(1-\rho_1)}$$
$$= \frac{\nu E(X^2)}{2(1-\rho_1-\rho_2)(1-\rho_1)}.$$

If there are n different classes of jobs, a solution can be obtained in a similar way for V_j, $j = 1, \ldots, n$. First, note that the total amount of work in the system due to jobs of Classes $1, \ldots, k$ is independent of the internal priority rule concerning classes $1, \ldots, k$ and depends only on the fact that each one of them has priority over all the jobs from Classes $k+1, \ldots, n$. So, $V_1 + \cdots + V_k$ is the same as it would be in case Classes $1, \ldots, k$ are coalesced in a single macro-class I and Classes $k+1, \ldots, n$ are coalesced in a single macro-class II.

Using this concept of macro-classes in a repetitive fashion it can be shown that

$$E(W_k^q) = \frac{\nu E(X^2)}{2(1-\rho_1-\cdots-\rho_k)(1-\rho_1-\cdots-\rho_{k-1})}.$$

Assume now that a Class k job has weight w_k. This implies that each unit of time a Class k job remains in the system costs w_k dollars. In all the scheduling models considered previously the objective has been the minimization of the total expected weighted completion time. In the model considered here there is an infinite arrival stream. The objective cannot be the total expected weighted completion time, but must be something different. The appropriate objective now is to minimize the total expected cost *per unit time*. That is, the average expected amount of money that has to be paid out per unit time by the jobs that are in the system. It can be shown that this objective is equivalent to the objective of minimizing the average cost per job. More formally, this objective is

$$\sum_{k=1}^{n} w_k \frac{\nu_k}{\nu} E(W_k^q).$$

Minimizing this objective is equivalent to minimizing

$$\sum_{k=1}^{n} w_k \nu_k E(W_k^q).$$

The following theorem specifies the static priority list that minimizes this objective.

Theorem 11.4.1. *Under the optimal nonpreemptive dynamic policy the decision-maker selects, whenever the machine is freed, from among the waiting jobs one with the highest value of $\lambda_j w_j$. This implies that the jobs are scheduled according to the WSEPT rule.*

Proof. The proof is again based on an adjacent pairwise interchange argument. Assume $n \geq 3$ and that Classes 2 and 3 are not prioritized according to the

$\lambda_j w_j$ rule, i.e., Class 2 has priority over Class 3, and

$$\lambda_2 w_2 < \lambda_3 w_3.$$

Let $\Sigma(2,3)$ denote the value of

$$\sum_{k=1}^{n} w_k \nu_k E(W_k^q).$$

under this priority assignment and let $\Sigma(3,2)$ denote the value of the objective after interchanging the priorities of Classes 2 and 3. Now,

$$\Sigma(2,3) - \Sigma(3,2) = \frac{(2 - 2\rho_1 - \rho_2 - \rho_3)\rho_2\rho_3\nu E(X^2)(\lambda_3 w_3 - \lambda_2 w_2)}{2(1-\rho_1)(1-\rho_1-\rho_2)(1-\rho_1-\rho_3)(1-\rho_1-\rho_2-\rho_3)} \geq 0$$

The above argument applies to priority Classes k and $k+1$ with $k > 1$. To see this, combine all classes with priority over Class k as a single macro-class having priority 1 (with the appropriate arrival rate and service time distribution). Treat Classes k and $k+1$ as Classes 2 and 3, respectively. A similar argument handles the case for Classes 1 and 2.

Thus any assignment of priorities that differs from the λw rule can be improved by pairwise interchanges and so the WSEPT rule is therefore optimal. □

The following example illustrates the behavior of the WSEPT rule and compares the nonpreemptive setting with the preemptive setting described in the previous section.

Example 11.4.2 (Poisson Releases and Arbitrary Processing Times without Preemptions)

Consider a single machine with two job classes. The jobs are released according to a Poisson process and a release is a Class 1 job with probability 0.5 and a Class 2 job with probability 0.5. Class 1 jobs have nonpreemptive priority over Class 2 jobs. Consider first the case where the processing time distributions of both job classes are exponential with a mean of 2 minutes. The interrelease times of Class 1 jobs as well as of Class 2 jobs are exponentially distributed with a mean of 8 minutes, i.e., $\nu_1 = \nu_2 = 1/8$ and $\nu = 1/4$. Working out the expressions in this section yields the following results for the expected times that the jobs have to wait before their processing can start:

$$E(W_1^q) = \frac{.125 \times 8 + .125 \times 8}{2(1 - .125 \times 2)} = 1.333$$

and

$$E(W_2^q) = \frac{E(W_1^q)}{1 - \rho_1 - \rho_2} = 2.667.$$

Notice that

$$E(W^q) = 0.5E(W_1^q) + 0.5E(W_2^q) = 2,$$

which is equal to $E(W^q)$ when preemptions are allowed. That the expected waiting times of all the jobs are equal in the preemptive case and in the non-preemptive case is to be expected (see Example 11.3.2 and Exercise 11.13).

Consider now the case where both job classes have deterministic processing times with a mean of 2 minutes. Working out the expressions yields the following results for the expected times that the jobs have to wait for processing:

$$E(W_1^q) = \frac{.125 \times 4 + .125 \times 4}{2(1 - .125 \times 2)} = 0.667$$

and

$$E(W_2^q) = \frac{E(W_1^q)}{1 - \rho_1 - \rho_2} = 1.333.$$

Notice that

$$E(W^q) = 0.5E(W_1^q) + 0.5E(W_2^q) = 1,$$

which is only half of the total expected waiting time obtained for the case with exponentially distributed processing times. So a smaller variance in the processing times causes a significant reduction in the expected waiting times. ||

Clearly, the example above can be generalized to arbitrary processing time distributions.

Example 11.4.3 (Poisson Releases and Exponential Processing Times without Preemptions)

Consider again a single machine and two job classes. The jobs are released according to a Poisson process with rate $\nu = 1/4$. Each release is a Class 1 job with probability 0.5 and a Class 2 job with probability 0.5. So the releases of Class 1 jobs are Poisson with rate $\nu_1 = 1/8$ and the releases of Class 2 jobs are Poisson with rate $\nu_2 = 1/8$. The processing times of the Class 1 jobs are exponentially distributed with mean 1 and the processing times of the Class 2 jobs are exponentially distributed with mean 3. The weight of a Class 1 job is 1 and the weight of a Class 2 job is 3.

If Class 1 jobs have a higher priority than Class 2 jobs, then

$$E(W_1^q) = \frac{.125 \times 2 + .125 \times 18}{2(1 - .125 \times 1)} = 1.429$$

and

$$E(W_2^q) = \frac{E(W_1^q)}{1 - \rho_1 - \rho_2} = 2.857.$$

The value of the objective to be minimized is

$$\sum_{k=1}^{n} w_k \frac{\nu_k}{\nu} E(W_k^q) = 1 \times 0.5 \times 1.429 + 3 \times 0.5 \times 2.857 = 5.$$

Consider now the case where the Class 2 jobs have a higher priority than the Class 1 jobs:

$$E(W_2^q) = \frac{.125 \times 2 + .125 \times 18}{2(1 - .125 \times 3)} = 2$$

and

$$E(W_1^q) = \frac{E(W_2^q)}{1 - \rho_1 - \rho_2} = 4.$$

The value of the objective to be minimized is

$$\sum_{k=1}^{n} w_k \frac{\nu_k}{\nu} E(W_k^q) = 1 \times 0.5 \times 4 + 3 \times 0.5 \times 2 = 5.$$

Note that the value of the objective to be minimized is the same under the two orderings. This was to be expected since the $\lambda_j w_j$ values of the two job classes are the same. Both priority orderings have to be optimal. ||

11.5 Discussion

Consider the case with many job classes, Poisson releases, and exponential processing time distributions (i.e., a model that satisfies the conditions in Section 11.3 as well as those in Section 11.4). The results in Section 11.3 imply that the preemptive WSEPT rule minimizes the total expected weighted completion time in the class of preemptive dynamic policies, while the results in Section 11.4 imply that the nonpreemptive WSEPT rule minimizes the total expected weighted completion time in the class of nonpreemptive dynamic policies. Clearly, the realizations of the process under the two different rules are different.

In order to obtain some more insight into the results presented in Section 11.4, consider the following limiting case. Suppose that there are many, say 10,000, different job classes. Each class has an extremely low Poisson release rate. The total job release rate will keep the machine occupied, say, 40% of the time. The machine will alternate between busy periods and idle periods, and during the busy periods it may process on the average, say, 10 jobs. These 10 jobs are most likely jobs from 10 different classes. The process during such a busy period may be viewed as a nonpreemptive stochastic scheduling problem (with a random, but finite number of jobs). The results in Section 11.4 imply

that the nonpreemptive WSEPT rule minimizes the total expected weighted completion time.

The case not considered in this chapter is a generalization of the case described in Section 10.2, i.e., the jobs have arbitrary processing time distributions and different release dates with preemptions allowed. When all the jobs are released at the same time, then the Gittins index policy is optimal. It turns out that when the jobs are released according to a Poisson process, an index policy is again optimal. However, the index is then not as easy to characterize as the Gittins index described in Section 10.2. A very special case for which the optimal preemptive policy can be characterized easily is considered in Exercise 11.7.

Some of the results in this chapter can be extended to machines in parallel. For example, the results in Section 11.4 can be generalized to machines in parallel under the condition that the processing time distributions of all classes are the same.

This chapter mainly focuses on the conditions under which the WSEPT rule is optimal under the assumption that the jobs are released at different points in time. It does not appear that similar results can be obtained for many other priority rules as well.

However, it can be shown that under certain conditions the preemptive EDD rule minimizes $L_{\max}$. Assume that the jobs are released at random times, and that the processing times are random variables from arbitrary distributions. So the model is a stochastic counterpart of $1 \mid r_j, prmp \mid L_{\max}$. If the due dates are deterministic, then it can be shown fairly easily that the preemptive EDD rule minimizes $L_{\max}$ (see Exercise 11.20). If the time between the release date and the due date of each job is exponentially distributed with mean $1/\mu$, then the policy that minimizes $E(L_{\max})$ can also be determined.

Exercises (Computational)

11.1. Consider three jobs. The three jobs have exponential processing times with rates $\lambda_1 = \lambda_2 = 1$ and $\lambda_3 = 2$ and the weights are $w_1 = 1$ and $w_2 = w_3 = 2$. Jobs 1 and 2 are released at time zero and job 3 is released after an exponential time with mean 1. Preemptions are allowed.

(a) Show that the preemptive WSEPT rule minimizes the total expected weighted completion time.

(b) Compute the total expected weighted completion time under this rule.

11.2. Consider the same three jobs as in Exercise 11.1. However, preemptions are now not allowed.

(a) Show that the nonpreemptive WSEPT rule minimizes the total expected weighted completion time.

(b) Compute the total expected weighted completion time under this rule.

(c) Compare the outcome of part (b) with the outcome of part (b) in the previous exercise and explain the difference.

11.3. Consider a single machine that is subject to Poisson releases with rate $\nu = 0.5$. All the jobs are of the same class and the processing time distribution is a mixture of two exponentials. With probability $1 - p$ the processing time is 0 and with probability p the processing time is exponentially distributed with rate p. So the mean of the mixture is 1. The jobs are served according to the First In First Out (FIFO) rule and preemptions are not allowed.

(a) Apply the Pollaczek-Khintchine formula and find an expression for $E(W_q)$.
(b) What happens with $E(W_q)$ when $p \to 0$?

11.4. Consider again a single machine that is subject to Poisson releases with rate $\nu = 0.5$. All the jobs are of the same class and the processing time distribution is a mixture of two exponentials. With probability $1 - p$ the processing time is 0 and with probability p it is exponential with rate p. However, now preemptions are allowed.

(a) Formulate the policy that minimizes the long term average waiting (or flow) time.
(b) Find an expression for $E(W_q)$ as a function of p under this optimal policy.
(c) Compare the expression under (b) with the expression found for $E(W_q)$ in Exercise 11.3. How does the value of p affect the difference?

11.5. Consider a single machine subject to Poisson releases with rate $\nu = 0.5$. All the jobs are of the same class and the processing time distribution is an Erlang(k, λ) distribution with mean 1, i.e., $k/\lambda = 1$. Preemptions are not allowed.

(a) Find an expression for $E(W_q)$ as a function of k.
(b) How does $E(W_q)$ depend on k?

11.6. Consider the following setting that is somewhat similar to Example 11.4.3. There are two job classes. The two classes are released according to Poisson processes with rates $\nu_1 = \nu_2 = 0.25$. Preemptions are not allowed. The processing time distribution of each one of the two job classes is a mixture of two exponentials with one of the two exponentials having mean 0. The processing time of a Class 1 job is 0 with probability $1 - p_1$ and exponentially distributed with rate p_1 with probability p_1. The processing time of a Class 2 job is 0 with probability $1 - p_2$ and exponentially distributed with rate p_2 with probability p_2. So the means of the processing times of both job classes are 1. Class 1 jobs have nonpreemptive priority over Class 2 jobs. Compute the expected waiting time of a Class 1 job and of a Class 2 job.

11.7. Consider the same setting as in the previous exercise. However, now Class 1 jobs have preemptive priority over Class 2 jobs. Compute the expected waiting time of a Class 1 job and of a Class 2 job and compare your results with the results obtained for the previous exercise.

11.8. Consider two job classes. The classes are released according to Poisson processes with rates $\nu_1 = \nu_2 = 0.25$. The processing time distributions of both job classes are mixtures of two exponentials with one of them having mean 0. The processing time of a Class 1 job is 0 with probability 0.5 and exponentially distributed with mean 2 with probability 0.5. The processing time of a Class 2 job is 0 with probability 0.75 and exponentially distributed with mean 4 with probability 0.25. Assume that the weight of a Class 1 job is $w_1 = 2$ and the weight of a Class 2 job is $w_2 = 3$. Preemptions are allowed.

(a) Describe the optimal policy when preemptions are not allowed.
(b) Describe the optimal policy when preemptions are allowed.

11.9. Consider a single machine. Jobs are released in batches of two according to a Poisson process. The arrival rate of the batches is $\nu = 0.25$. Each batch contains one job of Class 1 and one job of Class 2. The processing times of all jobs (from both classes) are exponential with mean 1. Class 1 jobs have preemptive priority over Class 2 jobs. Compute the expected waiting time of Class 1 jobs and the expected waiting time of Class 2 jobs.

11.10. Consider a single machine and two job classes. The processing times of Class 1 (2) jobs are exponentially distributed with rate $\lambda_1 = 6$ ($\lambda_2 = 12$). The weight of a Class 1 (2) job is $w_1 = 4$ ($w_2 = 3$). Preemptions are not allowed. When the machine is processing a job of either one of the two classes it is subject to breakdowns. The up times of a machine when it is processing a Class 1 (2) job are exponentially distributed with rate $\nu_1 = 6$ ($\nu_2 = 5$). The repair times after a breakdown when processing a Class 1 (2) job are arbitrarily distributed with mean $1/\mu_1 = 0.5$ ($1/\mu_2 = 1$). Which class of jobs should have a higher priority? (*Hint:* See Exercise 10.11.)

Exercises (Theory)

11.11. Consider three jobs. Two are available at time zero and the third one is released after an exponential amount of time. The three processing times are all deterministic. Preemptions are not allowed. Is it always optimal to start at time zero the job with the highest w_j/p_j ratio?

11.12. Consider three jobs. Two are available at time zero and the third one is released after a deterministic (fixed) amount of time. The three processing times are all exponential. Preemptions are not allowed. Is it always optimal to start at time zero the job with the highest $\lambda_j w_j$ ratio?

11.13. Compare the numerical results of Examples 11.3.2 and 11.4.2. Explain why the value of the total objective in the preemptive case is equal to the value of the total objective in the nonpreemptive case.

11.14. Consider the setting of Exercise 11.6. Assume $p_1 > p_2$. Which class should have nonpreemptive priority if the objective is to minimize the total expected waiting time over both classes?

11.15. In queueing theory there is a well-known result known as Little's Law which establishes a relationship between the long term average of the number of jobs waiting in queue and the expected waiting time of a job in steady state. Little's Law states that

$$E(N^q) = \nu E(W^q).$$

(a) Give an intuitive argument why this relationship holds.

(b) Present a proof for this relationship.

11.16. Consider an arbitrary release process (not necessarily Poisson). Job j has with probability p_j a processing time that is 0 and with probability $1 - p_j$ a processing time that is exponentially distributed with mean $1/\lambda_j$. Job j has a weight w_j. Preemptions are allowed. Describe the optimal preemptive dynamic policy that minimizes the total weighted expected completion time.

11.17. Consider a machine subject to Poisson releases of a single class. The processing time distribution is Erlang$(2, \lambda)$.

(a) Describe the optimal preemptive policy.

(b) When do preemptions occur?

11.18. Consider a machine that is subject to breakdowns. Breakdowns can occur only when the machine is processing a job. The up times of the machine, in between breakdowns, are i.i.d. exponential. The down times are i.i.d. (arbitrarily distributed) with mean $1/\mu$. There are two job classes. Both job classes are released according to Poisson processes. The processing times of the jobs of Class 1 (2) are arbitrarily distributed according to G_1 (G_2). The weight of Class 1 (2) is w_1 (w_2). Show that the nonpreemptive WSEPT rule is optimal in the class of nonpreemptive policies.

11.19. Consider a machine that is subject to breakdowns. The machine can only break down when it is processing a job. The up times of the machine, when it is processing jobs, are i.i.d. exponential and the down times are i.i.d. exponential as well. There are two job classes that are released according to independent Poisson processes. Class 1 jobs have preemptive priority over Class 2 jobs. The processing times of Class 1 (2) jobs are exponentially distributed with rate λ_1 (λ_2).

(a) Can Theorem 11.3.1 be generalized to this setting?

(b) Can Theorem 11.3.1 be generalized to include the case where the machine can break down at any time, i.e., also when it is idle.

11.20. Consider the following stochastic counterpart of $1 \mid r_j, prmp \mid L_{\max}$. The release dates and due dates are deterministic. The processing time of job j has an arbitrary distribution F_j. Show that EDD minimizes $E(L_{\max})$.

11.21. Consider the following stochastic counterpart of $1 \mid r_j, prmp \mid L_{\max}$. The release dates have an arbitrary distribution and the due date $D_j = R_j + X_j$, where X_j is exponentially distributed with rate 1. The processing time of job j has an arbitrary distribution F_j. Formulate the policy that minimizes $E(L_{\max})$.

Comments and References

Most of the results in this chapter have appeared in the queueing literature. Many queueing books consider priority queues, see Kleinrock (1976), Heyman and Sobel (1982), Wolff (1989), and Ross (2006). The application of work conservation in queueing theory was first considered by Wolff (1970); his paper contains the results described in Sections 11.1 and 11.2.

The optimality of WSEPT when processing times are exponentially distributed, release times arbitrarily distributed, and preemptions allowed is due to Pinedo (1983). The optimality of WSEPT when jobs are released according to Poisson processes, processing times are arbitrarily distributed, and preemptions not allowed has been established by Cobham (1954).

Chapter 12

Parallel Machine Models (Stochastic)

This chapter deals with parallel machine models that are stochastic counterparts of models discussed in Chapter 5. The body of knowledge in the stochastic case is somewhat less extensive than in the deterministic case.

The results focus mainly on the expected makespan, the total expected completion time and the expected number of tardy jobs. In what follows the number of machines is usually limited to two. Some of the proofs can be extended to more than two machines, but such extensions usually require more elaborate notation; since these extensions would not provide any additional insight, they are not presented here. The proofs for some of the structural properties of the stochastic models tend to be more involved than the proofs for the corresponding properties of their deterministic counterparts.

The first section deals with nonpreemptive models; the results in this section are obtained through interchange arguments. The second section focuses on preemptive models; the results in this section are obtained via dynamic programming approaches. The third section deals with due date related models. The fourth section shows how bounds obtained for online scheduling models can lead to bounds for parallel machine stochastic scheduling models.

12.1 The Makespan without Preemptions

This section considers optimal policies in the classes of nonpreemptive static list policies and nonpreemptive dynamic policies. Since preemptions are not al-

M.L. Pinedo, *Scheduling*, DOI: 10.1007/978-0-387-78935-4_12,

lowed, the main technique for determining optimal policies is based on pairwise interchange arguments.

First, the exponential distribution is considered in detail as its special properties facilitate the analysis considerably.

Consider two machines in parallel and n jobs. The processing time of job j is equal to the random variable X_j, that is exponentially distributed with rate λ_j. The objective is to minimize $E(C_{\max})$. Note that this problem is a stochastic counterpart of $P2 \mid\mid C_{\max}$, which is known to be NP-hard. However, in Section 10.4 it already became clear that scheduling processes with exponentially distributed processing times often have structural properties that their deterministic counterparts do not have. It turns out that this is also the case with machines in parallel.

A nonpreemptive static list policy is adopted. The jobs are put into a list and at time zero the two jobs at the top of the list begin their processing on the two machines. When a machine becomes free the next job on the list is put on the machine. It is not specified in advance on which machine each job will be processed, nor is it known a priori which job will be the last one to be completed.

Let Z_1 denote the time when the second to last job is completed, i.e., the first time a machine becomes free with the job list being empty. At this time the other machine is still processing its last job. Let Z_2 denote the time that the last job is completed on the other machine (i.e., Z_2 equals the makespan $C_{\max}$). Let the difference D be equal to $Z_2 - Z_1$. It is clear that the random variable D depends on the schedule. It is easy to see that minimizing $E(D)$ is equivalent to minimizing $E(C_{\max})$. This follows from the fact that

$$Z_1 + Z_2 = 2C_{\max} - D = \sum_{j=1}^{n} X_j,$$

which is a constant independent of the schedule.

In what follows, a slightly more general two-machine problem is considered for reasons that will become clear later. It is assumed that one of the machines is not available at time zero and becomes available only after a random time X_0, distributed exponentially with rate λ_0. The random variable X_0 may be thought of as the processing time of an additional job that takes precedence and *must* go first. Let $D(X_0, X_1, X_2, \ldots, X_n)$ denote the random variable D, under the assumption that, at time zero, a job with remaining processing time X_0 is being processed on one machine and a job with processing time X_1 is being started on the other. When one of the two machines is freed a job with processing time X_2 is started, and so on (see Figure 12.1). The next lemma examines the effect on D of changing a schedule by swapping the two consecutive jobs 1 and 2.

Lemma 12.1.1. *For any* λ_0 *and for* $\lambda_1 = \min(\lambda_1, \lambda_2, \ldots, \lambda_n)$

$$E(D(X_0, X_1, X_2, \ldots, X_n)) \leq E(D(X_0, X_2, X_1, \ldots, X_n)).$$

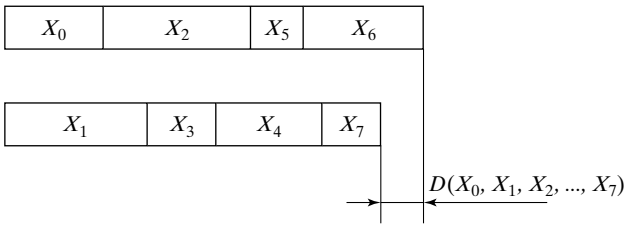


Fig. 12.1 The random variable D

Proof. Let q_j (r_j), $j = 0, 1, \ldots, n$, denote the probability that job j is the last job to be completed under schedule $1, 2, \ldots, n$ $(2, 1, \ldots, n)$. The distribution of D may be regarded as a mixture of $n+1$ distributions (all being exponential) with either mixing probabilities $q_0, \ldots, q_n$ or mixing probabilities $r_0, \ldots, r_n$. If the last job to be completed is exponential with rate λ_j, then (conditioned on that fact) D is exponentially distributed with rate λ_j. (Recall that if job j is still being processed on a machine, while the other machine is idle and no other job remains to be processed, then the remaining processing time of job j is still exponentially distributed with rate λ_j). So

$$E(D(X_0, X_1, X_2, \ldots, X_n)) = \sum_{j=0}^{n} q_j \frac{1}{\lambda_j},$$

and

$$E(D(X_0, X_2, X_1, \ldots, X_n)) = \sum_{j=0}^{n} r_j \frac{1}{\lambda_j}.$$

In order to prove the lemma it suffices to show that

$$\begin{aligned} q_0 &= r_0 \\ q_1 &\leq r_1 \\ q_j &\geq r_j \end{aligned}$$

for $j = 2, \ldots, n$. For job 0 to be the last one completed, it has to be larger than the sum of the other n processing times. Clearly, if this is the case, an interchange between jobs 1 and 2 does neither affect the probability of job 0 being completed last nor the value of D. In order to establish the necessary relationships between the mixing probabilities q_j and r_j consider first the case $n = 2$. It can be shown easily that

$$q_0 = \Big(\frac{\lambda_1}{\lambda_1 + \lambda_0}\Big)\Big(\frac{\lambda_2}{\lambda_2 + \lambda_0}\Big)$$

$$q_1 = \Big(\frac{\lambda_0}{\lambda_0+\lambda_1}\Big)\Big(\frac{\lambda_2}{\lambda_1+\lambda_2}\Big)$$
$$q_2 = 1 - q_0 - q_1$$

and

$$r_0 = \Big(\frac{\lambda_1}{\lambda_1+\lambda_0}\Big)\Big(\frac{\lambda_2}{\lambda_2+\lambda_0}\Big)$$
$$r_1 = 1 - r_0 - r_2$$
$$r_2 = \Big(\frac{\lambda_0}{\lambda_0+\lambda_2}\Big)\Big(\frac{\lambda_1}{\lambda_1+\lambda_2}\Big).$$

So

$$r_1 - q_1 = \frac{2\lambda_1\lambda_2}{\lambda_2^2-\lambda_1^2}\Big(\frac{\lambda_0}{\lambda_0+\lambda_1} - \frac{\lambda_0}{\lambda_0+\lambda_2}\Big) \geq 0.$$

This proves the lemma for $n = 2$.

Assume the lemma is true for $n-1$ jobs and let q'_j and r'_j, $j = 1, \ldots, n-1$, denote the corresponding probabilities under schedules $0, 1, 2, 3, \ldots, n-1$ and $0, 2, 1, 3, \ldots, n-1$. Assume as the induction hypothesis that

$$q'_0 = r'_0$$
$$q'_1 \leq r'_1$$
$$q'_j \geq r'_j,$$

for $j = 2, \ldots, n-1$. Let q_j and r_j denote now the corresponding probabilities with one additional job. Then

$$q_0 = r_0 = P(X_1 + X_2 + \cdots + X_n < X_0) = \prod_{j=1}^{n} \frac{\lambda_j}{\lambda_j+\lambda_0}$$

and

$$q_j = q'_j \frac{\lambda_n}{\lambda_j+\lambda_n}$$
$$r_j = r'_j \frac{\lambda_n}{\lambda_j+\lambda_n}$$

for $j = 1, \ldots, n-1$. So, from the induction hypothesis, it follows that

$$q_1 \leq r_1$$
$$q_j \geq r_j,$$

for $j = 2, \ldots, n-1$. Also, because $\lambda_1 \leq \lambda_j$ for all $j = 2, \ldots, n-1$, it follows that

$$\frac{\lambda_n}{\lambda_1+\lambda_n} \geq \frac{\lambda_n}{\lambda_j+\lambda_n}$$

for $j = 2, \ldots, n-1$. So

$$\sum_{j=1}^{n-1} q'_j \frac{\lambda_n}{\lambda_j + \lambda_n} \leq \sum_{j=1}^{n-1} r'_j \frac{\lambda_n}{\lambda_j + \lambda_n}.$$

Therefore

$$q_n \geq r_n$$

which completes the proof of the lemma. □

This lemma constitutes a crucial element in the proof of the following theorem.

Theorem 12.1.2.. *If there are two machines in parallel and the processing times are exponentially distributed, then the LEPT rule minimizes the expected makespan in the class of nonpreemptive static list policies.*

Proof. By contradiction. Suppose that a different rule is optimal. Suppose that according to this presumably optimal rule, the job with the longest expected processing time is not scheduled for processing either as the first or the second job. (Note that the first and second job are interchangeable as they both start at time zero). Then an improvement can be obtained by performing a pairwise interchange between this longest job and the job immediately preceding this job in the schedule; according to Lemma 12.1.1 this reduces the expected difference between the completion times of the last two jobs. Through a series of interchanges it can be shown that the longest job has to be one of the first two jobs in the schedule. In the same way it can be shown that the second longest job has to be among the first two jobs as well. The third longest job can be moved into the third position to improve the objective, and so on. With each interchange the expected difference, and thus the expected makespan, are reduced. □

The approach used in proving the theorem is basically an adjacent pairwise interchange argument. However, this pairwise interchange argument is not identical to the pairwise interchange arguments used in single machine scheduling. In pairwise interchange arguments applied to single machine problems, usually *no* restrictions are put on the relation between the interchanged jobs and those that come after them. In Lemma 12.1.1 jobs not involved in the interchange have to satisfy a special condition, viz., one of the two jobs being interchanged must have a larger expected processing time than all the jobs following it. Requiring such a condition has certain implications. In general, when no special conditions are imposed, an adjacent pairwise interchange argument actually yields two results: it shows that one schedule minimizes the objective while the *reverse* schedule maximizes that same objective. With a special condition like the one in Lemma 12.1.1 the argument works only in one direction. It actually can be shown that the SEPT rule does *not* always maximize $E(D)$ among nonpreemptive static list policies.

The result presented in Theorem 12.1.2 differs from the results obtained for its deterministic counterpart considerably. One difference is the following: minimizing the makespan in a deterministic setting requires only an optimal *partition* of the n jobs over the two machines. After the allocation has been determined, the set of jobs allocated to a specific machine may be sequenced in any order. With exponential processing times, a sequence is determined in which the jobs are to be *released* in order to minimize the expected makespan. No deviation is allowed from this sequence and it is not specified at time zero how the jobs will be partitioned between the machines. This depends on the evolution of the process.

The following example shows that distributions other than exponential may result in optimal schedules that differ considerably from LEPT.

Example 12.1.3 (Counterexample to Optimality of LEPT with Arbitrary Processing Times)

Consider two machines and n jobs. The processing time of job j is a mixture of two exponentials. With probability p_j it is exponential with rate ∞, i.e., it has a zero processing time, and with probability $1 - p_j$ it is exponential with rate λ_j. So

$$E(X_j) = \frac{1 - p_j}{\lambda_j}.$$

Assume $\lambda_1 \leq \cdots \leq \lambda_n$. Since there are no assumptions on $p_1, \ldots, p_n$, sequence $1, \ldots, n$ may not be LEPT. That the sequence $1, \ldots, n$ still minimizes the expected makespan can be argued as follows. Note that if a job has zero processing time, it basically does not exist and does not have any effect on the process. Suppose it is known in advance which jobs have zero processing times and which jobs are exponentially distributed with processing times larger than zero. From Theorem 12.1.2 it follows that sequencing the nonzero jobs in increasing order of their rates, that is, according to LEPT, minimizes the expected makespan. This implies that the sequence $1, \ldots, n$ is optimal for *any* subset of jobs having zero processing times. So, sequence $1, \ldots, n$, which is not necessarily LEPT, is always optimal. The $p_1, \ldots, p_n$ can be such that actually the SEPT rule minimizes the expected makespan. ||

In contrast to the results of Section 10.4, which do not appear to hold for distributions other than the exponential, the LEPT rule does minimize the expected makespan for other distributions as well.

Consider the case where the processing time of job j is distributed according to a mixture of two exponentials, i.e., with probability p_{1j} according to an exponential with rate λ_1 and with probability p_{2j} $(= 1 - p_{1j})$ according to an exponential with rate λ_2. Assume $\lambda_1 < \lambda_2$. So

$$P(X_j > t) = p_{1j} e^{-\lambda_1 t} + p_{2j} e^{-\lambda_2 t}.$$

This distribution can be described as follows: when job j is put on the machine a (biased) coin is tossed. Dependent upon the outcome of the toss the processing time of job j is either exponential with rate λ_1 or exponential with rate λ_2. After the rate has been determined this way the distribution of the remaining processing time of job j does not change while the job is being processed. So each processing time is distributed according to one of the two exponentials with rates λ_1 and λ_2.

The subsequent lemma again examines the effect on D of an interchange between two consecutive jobs 1 and 2 on two machines in parallel. Assume again that X_0 denotes the processing time of a job 0 with an exponential distribution with rate λ_0. This rate λ_0 may be different from either λ_1 or λ_2.

Lemma 12.1.4. *For arbitrary λ_0, if $p_{11} \geq p_{12}$, i.e., $E(X_1) \geq E(X_2)$, then*

$$E(D(X_0, X_1, X_2, \ldots, X_n)) \leq E(D(X_0, X_2, X_1, \ldots, X_n)).$$

Proof. Note that $D(X_0, X_1, X_2 \ldots, X_n)$ is exponential with one of three rates, namely λ_0, λ_1 or λ_2. Denote the probabilities of these three events under schedule $0, 1, 2, \ldots, n$ by q_0, q_1 and q_2, where $q_0 + q_1 + q_2 = 1$. Denote by r_0, r_1 and r_2 the probabilities of the same events under schedule $0, 2, 1, 3, \ldots, n$. It is clear that q_0 is equal to r_0 by a similar argument as in Lemma 12.1.1. In order to prove the lemma it suffices to show that $q_1 < r_1$.

For $n = 2$ there are four cases to be considered in the computation of q_1. With probability $p_{21}p_{22}$ both jobs 1 and 2 are exponential with rate λ_2. In this case the probability that $D(X_0, X_1, X_2)$ is exponentially distributed with rate λ_1 is zero. With probability $p_{11}p_{22}$ job 1 is exponentially distributed with rate λ_1, while job 2 is exponentially distributed with rate λ_2. In order for $D(X_0, X_1, X_2)$ to have rate λ_1, job 1 has to outlast job 0 and, after job 2 is started on the machine on which job 0 is completed, job 1 has to outlast job 2 as well. This happens with probability

$$\Big(\frac{\lambda_0}{\lambda_0 + \lambda_1}\Big)\Big(\frac{\lambda_2}{\lambda_2 + \lambda_1}\Big).$$

The other two cases can also be computed easily. Summarizing,

$$q_1 = p_{11}p_{12}\Big(\frac{\lambda_0}{\lambda_0 + \lambda_1} + \Big(\frac{\lambda_1}{\lambda_0 + \lambda_1}\Big)\Big(\frac{\lambda_0}{\lambda_0 + \lambda_1}\Big)\Big) + p_{11}p_{22}\Big(\frac{\lambda_0}{\lambda_0 + \lambda_1}\Big)\Big(\frac{\lambda_2}{\lambda_1 + \lambda_2}\Big)$$

$$+ p_{21}p_{12}\Big(\Big(\frac{\lambda_0}{\lambda_0 + \lambda_2}\Big)\Big(\frac{\lambda_2}{\lambda_1 + \lambda_2}\Big) + \Big(\frac{\lambda_2}{\lambda_0 + \lambda_2}\Big)\Big(\frac{\lambda_0}{\lambda_0 + \lambda_1}\Big)\Big) + p_{21}p_{22}\Big(0\Big).$$

So, the first term on the R.H.S. of this expression corresponds to the event where both jobs 1 and 2 are exponentially distributed with rates λ_1, which happens with probability $p_{11}p_{12}$; the second term corresponds to the event where job 1 is distributed according to an exponential with rate λ_1 and job 2 according to an exponential with rate λ_2 and the third term corresponds to the event

where job 1 is distributed according to an exponential with rate λ_2 and job 2 according to an exponential with rate λ_1. The fourth term corresponds to the event where both jobs 1 and 2 are exponentially distributed with rate λ_2. A similar expression can be obtained for r_1. Subtracting yields

$$r_1 - q_1 = (p_1 - p_2)\frac{2\lambda_1\lambda_2}{\lambda_2^2 - \lambda_1^2}\Big(\frac{\lambda_0}{\lambda_0 + \lambda_1} - \frac{\lambda_0}{\lambda_0 + \lambda_2}\Big) \geq 0,$$

which shows that $r_1 \geq q_1$ for $n = 2$.

Let q_0', q_1' and q_2' and r_0', r_1' and r_2' denote the corresponding probabilities when there are only $n-1$ jobs under schedule $1, 2, 3, \ldots, n-1$ and schedule $2, 1, 3, \ldots, n-1$, respectively.

Now let $n > 2$, and assume inductively that $r_1' \geq q_1'$. Then

$$q_1 = q_1' p_{1n} + q_1' p_{2n}\frac{\lambda_2}{\lambda_1 + \lambda_2} + q_2' p_{1n}\frac{\lambda_2}{\lambda_1 + \lambda_2} + q_0' p_{1n}\frac{\lambda_0}{\lambda_0 + \lambda_1}$$

where the last term on the R.H.S. is independent of the schedule $1, 2, \ldots, n-1$. A similar expression holds for r_1, and

$$r_1 - q_1 = (r_1' - q_1')\Big(p_{1n}\frac{\lambda_1}{\lambda_1 + \lambda_2} + p_{2n}\frac{\lambda_2}{\lambda_1 + \lambda_2}\Big) \geq 0.$$

This completes the proof of the lemma. □

Note that there are no conditions on λ_0; the rate λ_0 may or may not be equal to either λ_1 or λ_2. With this lemma the following theorem can be shown easily.

Theorem 12.1.5. *The LEPT rule minimizes the expected makespan in the class of nonpreemptive static list policies when there are two machines in parallel and when the processing times are distributed according to a mixture of two exponentials with rates λ_1 and λ_2.*

Proof. Any permutation schedule can be transformed into the LEPT schedule through a series of adjacent pairwise interchanges between a longer job and a shorter job immediately preceding it. With each interchange $E(D)$ decreases because of Lemma 12.1.4. □

Showing that LEPT minimizes the expected makespan can be done in this case without any conditions on the jobs that are not part of the interchange. This is in contrast to Theorem 12.1.2, where the jobs following the jobs in the interchange had to be smaller than the largest job involved in the pairwise interchange. The additional condition requiring the other expected processing times to be smaller than the expected processing time of the larger of the two jobs in the interchange, is *not* required in this case.

Just as Example 12.1.3 extends the result of Theorem 12.1.2 to a mixture of an exponential and zero, Theorem 12.1.5 can be extended to include mixtures

of three exponentials, with rates λ_1, λ_2 and ∞. The next example shows that the LEPT rule, again, does not necessarily minimize the expected makespan.

Example 12.1.6 (Counterexample to Optimality of LEPT with Arbitrary Processing Times)

Let p_{1j} denote the probability job j is exponentially distributed with rate λ_1 and p_{2j} the probability it is distributed with rate λ_2. Assume $\lambda_1 < \lambda_2$. The probability that the processing time of job j is zero is $p_{0j} = 1 - p_{1j} - p_{2j}$. Through similar arguments as those used in Lemma 12.1.4 and Theorem 12.1.5 it can be shown that in order to minimize the expected makespan the jobs in the optimal nonpreemptive static list policy have to be ordered in decreasing order of p_{1j}/p_{2j}. The jobs with the zero processing times again do not play a role in the schedule. As in Example 12.1.3, this implies that the optimal sequence is not necessarily LEPT. ||

The following example is a continuation of the previous example and is an illustration of the Largest Variance first (LV) rule.

Example 12.1.7 (Application of Largest Variance first (LV) Rule)

Consider the special case of the previous example with

$$\frac{1}{\lambda_1} = 2$$

and

$$\frac{1}{\lambda_2} = 1.$$

Let

$$\begin{aligned} p_{0j} &= a_j \\ p_{1j} &= a_j \\ p_{2j} &= 1 - 2a_j \end{aligned}$$

So

$$E(X_j) = \frac{p_{1j}}{\lambda_1} + \frac{p_{2j}}{\lambda_2} = 1,$$

for all j and

$$Var(X_j) = 1 + 4a_j.$$

From the previous example it follows that sequencing the jobs in decreasing order of p_{1j}/p_{2j} minimizes the expected makespan. This rule is equivalent to scheduling the jobs in decreasing order of $a_j/(1 - 2a_j)$. Since $0 \leq a_j \leq 1/2$, scheduling the jobs in decreasing order of $a_j/(1 - 2a_j)$ is equivalent to scheduling in decreasing order of a_j, which in turn is equivalent to the *Largest Variance first* rule. ||

The methodology used in proving that LEPT is optimal for the expected makespan on two machines does not easily extend to problems with more than two machines or problems with other processing time distributions. Consider the following generalization of this approach for m machines. Let Z_1 denote the time that the first machine becomes idle with no jobs waiting for processing, Z_2 the time the second machine becomes idle, and so on, and let Z_m denote the time the last machine becomes idle. Clearly Z_m equals the makespan. Let

$$D_i = Z_{i+1} - Z_i \qquad i = 1, \ldots, m-1.$$

From the fact that the sum of the processing times is

$$\sum_{j=1}^{n} X_j = \sum_{i=1}^{m} Z_i = mC_{\max} - D_1 - 2D_2 - \cdots - (m-1)D_{m-1},$$

which is independent of the schedule, it follows that minimizing the makespan is equivalent to minimizing

$$\sum_{i=1}^{m-1} iD_i.$$

A limited number of processing time distributions can be handled this way. For example, the setting of Theorem 12.1.5 can be extended relatively easily following this approach. However, the scenario of Theorem 12.1.2 cannot be extended that easily.

So far only the class of nonpreemptive static list policies has been considered in this section. It turns out, that most optimal policies in the class of nonpreemptive static list policies are also optimal in the classes of nonpreemptive dynamic policies and preemptive dynamic policies. The proof that a nonpreemptive static list policy is optimal in these other two classes of policies as well is based on induction arguments that are very similar to those used in the second and third part of the proof of Theorem 10.4.1. In Section 12.2 an entirely different approach is presented which first proves optimality in the class of preemptive dynamic policies. As the optimal policy is a nonpreemptive static list policy, the policy is then also optimal in the classes of nonpreemptive dynamic policies and nonpreemptive static list policies.

Only the expected makespan has been considered in this section. In a nonpreemptive setting the total expected completion time $E(\sum C_j)$ is more difficult to deal with than the expected makespan. Indeed, an approach similar to the one used to show that LEPT minimizes the expected makespan for exponential processing times, has not been found to show that SEPT minimizes the total expected completion time. However, if the processing times are distributed as in Theorem 12.1.5, it can be shown that SEPT minimizes the total expected completion time and if the processing times are distributed as in Example 12.1.7 it can be shown that LV minimizes the total expected completion time.

12.2 The Makespan and Total Completion Time with Preemptions

Pairwise interchange arguments are basically used for determining optimal policies in the class of nonpreemptive static list policies. After determining an optimal nonpreemptive static list policy, it can often be argued that this policy is also optimal in the class of nonpreemptive dynamic policies and possibly in the class of preemptive dynamic policies.

In this section an alternative proof for Theorem 12.1.2 is presented. The approach is entirely different. A dynamic programming type proof is constructed within the class of *preemptive* dynamic policies. After obtaining the result that the nonpreemptive LEPT policy minimizes the expected makespan in the class of preemptive dynamic policies, it is concluded that it is also optimal in the class of nonpreemptive dynamic policies as well as in the class of nonpreemptive static list policies.

This approach can be used for proving that LEPT minimizes the expected makespan for m machines in parallel. It will be illustrated for two machines in parallel since the notation is then much simpler.

Suppose $\lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_n$. Let $V(J)$ denote the expected value of the minimum remaining time needed (that is, under the optimal policy) to finish all jobs given that all the jobs in the set $J = j_1, \ldots, j_l$ already have been completed. If $J = \emptyset$, then $V(J)$ is simply denoted by V. Let $V^*(J)$ denote the same time quantity under the LEPT policy. Similarly, V^* denotes the expected value of the remaining completion time under LEPT when no job has yet been completed.

Theorem 12.2.1. *The nonpreemptive LEPT policy minimizes the expected makespan in the class of preemptive dynamic policies.*

Proof. The proof is by induction on the number of jobs. Suppose that the result is true when there are less than n jobs. It has to be shown that it is also true when there are n jobs. That is, a policy that at time 0 (when there are n jobs waiting for processing) does not act according to LEPT but at the first job completion (when there are $n-1$ jobs remaining to be processed) switches over to LEPT results in a longer expected makespan than a policy that acts according to LEPT starting at time zero. In a sense the structure of this proof is somewhat similar to the proof of Theorem 5.2.7.

Conditioning on the first job completion yields

$$V = \min_{j,k} \Big(\frac{1}{\lambda_j + \lambda_k} + \frac{\lambda_j}{\lambda_j + \lambda_k} V^*(\{j\}) + \frac{\lambda_k}{\lambda_j + \lambda_k} V^*(\{k\}) \Big).$$

The expected time until the first job completion is the first term on the RHS. The second (third) term is equal to the probability of job j (k) being the first job to be completed, multiplied by the expected remaining time needed to complete

the $n-1$ remaining jobs under LEPT. This last equation is equivalent to

$$0 = \min_{j,k} \Big(1 + \lambda_j \Big(V^*(\{j\}) - V^*\Big) + \lambda_k \Big(V^*(\{k\}) - V^*\Big) + (\lambda_j + \lambda_k)\Big(V^* - V\Big)\Big).$$

Since λ_1 and λ_2 are the two smallest λ_j values and supposedly $V^* \geq V$ the fourth term on the R.H.S. is minimized by $\{j,k\} = \{1,2\}$. Hence to show that LEPT is optimal it suffices to show that $\{j,k\} = \{1,2\}$ also minimizes the sum of the second and third term. In order to simplify the presentation let

$$A_j = \lambda_j \Big(V^*(\{j\}) - V^*\Big)$$

and let

$$D_{jk} = A_j - A_k.$$

In order to show that

$$\lambda_j \Big(V^*(\{j\}) - V^*\Big) + \lambda_k \Big(V^*(\{k\}) - V^*\Big) = A_j + A_k$$

is minimized by $\{j,k\} = \{1,2\}$, it suffices to show that $\lambda_j < \lambda_k$ implies $A_j \leq A_k$ or, equivalently, $D_{jk} \leq 0$. To prove that $D_{jk} \leq 0$ is done in what follows by induction.

Throughout the remaining part of the proof V^*, A_j and D_{jk} are considered functions of $\lambda_1, \ldots, \lambda_n$. Define $A_j(J)$ and $D_{jk}(J)$, assuming jobs j and k are not in J, in the same way as A_j and D_{jk}, e.g.,

$$A_j(J) = \lambda_j \Big(V^*(J \cup \{j\}) - V^*(J)\Big).$$

Before proceeding with the induction a number of identities have to be established. If j and k are the two smallest jobs not in set J, then LEPT processes jobs j and k first. Conditioning on the first job completion results in the identity

$$V^*(J) = \frac{1}{\lambda_j + \lambda_k} + \frac{\lambda_j}{\lambda_j + \lambda_k} V^*(J \cup \{j\}) + \frac{\lambda_k}{\lambda_j + \lambda_k} V^*(J \cup \{k\})$$

or

$$(\lambda_j + \lambda_k) V^*(J) = 1 + \lambda_j V^*(J \cup \{j\}) + \lambda_k V^*(J \cup \{k\}).$$

Similarly,

$$\begin{aligned}(\lambda_1 + \lambda_2 + \lambda_3) A_1 &= \lambda_1(\lambda_1 + \lambda_2 + \lambda_3) V^*(\{1\}) - \lambda_1(\lambda_1 + \lambda_2 + \lambda_3) V^* \\ &= \lambda_1 \Big(1 + \lambda_1 V^*(\{1\}) + \lambda_2 V^*(\{1,2\}) + \lambda_3 V^*(\{1,3\})\Big) \\ &\quad - \lambda_1 \Big(1 + \lambda_1 V^*(\{1\}) + \lambda_2 V^*(\{2\}) + \lambda_3 V^*\Big)\end{aligned}$$

$$= \lambda_1 \Big(\lambda_3 V^*(\{1,3\}) - \lambda_3 V^*(\{1\}) \Big) + \lambda_2 \Big(\lambda_1 V^*(\{1,2\}) - \lambda_1 V^*(\{2\}) \Big) + \lambda_3 A_1$$

or

$$(\lambda_1 + \lambda_2) A_1 = \lambda_1 A_3(\{1\}) + \lambda_2 A_1(\{2\}).$$

The following identities can be established in the same way:

$$(\lambda_1 + \lambda_2) A_2 = \lambda_1 A_2(\{1\}) + \lambda_2 A_3(\{2\})$$

and

$$(\lambda_1 + \lambda_2) A_j = \lambda_1 A_j(\{1\}) + \lambda_2 A_j(\{2\}),$$

for $j = 3, \ldots, n$. Thus, with $D_{12} = A_1 - A_2$, it follows that

$$D_{12} = \frac{\lambda_1}{\lambda_1 + \lambda_2} D_{32}(\{1\}) + \frac{\lambda_2}{\lambda_1 + \lambda_2} D_{13}(\{2\}),$$

and

$$D_{2j} = \frac{\lambda_1}{\lambda_1 + \lambda_2} D_{2j}(\{1\}) + \frac{\lambda_2}{\lambda_1 + \lambda_2} D_{3j}(\{2\}),$$

for $j = 3, \ldots, n$.

Assume now as induction hypothesis that if $\lambda_j < \lambda_k$, and $\lambda_1 \leq \cdots \leq \lambda_n$, then

$$D_{jk} \leq 0$$

and

$$\frac{dD_{12}}{d\lambda_1} \geq 0.$$

In the remaining part of the proof, these two inequalities are shown by induction on n. When $n = 2$

$$D_{jk} = \frac{\lambda_j - \lambda_k}{\lambda_j + \lambda_k}$$

and the two inequalities can be established easily.

Assume that the two inequalities of the induction hypothesis hold when there are less than n jobs remaining to be processed. The induction hypothesis now implies that $D_{13}(\{2\})$ as well as $D_{23}(\{1\})$ are nonpositive when there are n jobs remaining to be completed. It also provides

$$\frac{dD_{13}(\{2\})}{d\lambda_1} \geq 0.$$

This last inequality has the following implication: if λ_1 increases then $D_{13}(\{2\})$ increases. The moment λ_1 reaches the value of λ_2 jobs 1 and 2 become interchangeable. Therefore

$$D_{13}(\{2\}) \leq D_{23}(\{1\}) = -D_{32}(\{1\}) \leq 0.$$

From the fact that $\lambda_1 < \lambda_2$ it follows that D_{12} is nonpositive. The induction hypothesis also implies that $D_{2j}(\{1\})$ and $D_{3j}(\{2\})$ are nonpositive. So D_{2j} is nonpositive. This completes the induction argument for the first inequality of the induction hypothesis. The induction argument for the second inequality can be established by differentiating

$$\frac{\lambda_1}{\lambda_1 + \lambda_2} D_{32}(\{1\}) + \frac{\lambda_2}{\lambda_1 + \lambda_2} D_{13}(\{2\})$$

with respect to λ_1 and then using induction to show that every term is positive. □

This proof shows that LEPT is optimal in the class of preemptive dynamic policies. As the optimal policy is a nonpreemptive static list policy it also has to be optimal in the class of nonpreemptive static list policies and in the class of preemptive dynamic policies. In contrast to the first proof of the same result, this approach also works for an arbitrary number of machines in parallel. The notation, however, becomes significantly more involved.

The interchange approach described in Section 12.1 is not entirely useless. To show that the nonpreemptive LEPT policy is optimal when the processing time distributions are ICR, one *must* adopt a pairwise interchange type of argument. The reason is obvious. In a preemptive framework the remaining processing time of an ICR job that has received a certain amount of processing may become less (in expectation) than the expected processing time of a job that is waiting for processing. This then would lead to a preemption. The approach used in Lemma 12.1.1 and Theorem 12.1.2 can be applied easily to a number of different classes of distributions for which the approach used in Theorem 12.2.1 does not appear to yield the optimal nonpreemptive schedule.

Consider minimizing the total expected completion time of the n jobs. The processing time of job j is exponentially distributed with rate λ_j. In Chapter 5 it was shown that the SPT rule is optimal for the deterministic counterpart of this problem. This gives an indication that in a stochastic setting the *Shortest Expected Processing Time first (SEPT)* rule may minimize the total expected completion time under suitable conditions. Consider again two machines in parallel with n jobs. The processing time of job j is exponentially distributed with rate λ_j. Assume now $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n$. An approach similar to the one followed in Theorem 12.2.1 for the expected makespan can be followed for the total expected completion time. Let $W(J)$ denote the expected value of the minimum total remaining completion time needed to finish the jobs still in the system under the optimal policy, given that the jobs in set J already have been completed. Again, if $J = \emptyset$, then $W(J)$ is simply denoted by W. Let $W^*(J)$ denote the same quantity under the SEPT policy.

Theorem 12.2.2. *The nonpreemptive SEPT policy minimizes the total expected completion time in the class of preemptive dynamic policies.*

Proof. The approach is similar to the one used in Theorem 12.2.1. The proof is again by induction on the number of jobs. Suppose that the result is true when there are less than n jobs. It has to be shown that it is also true when there are n jobs. That is, a policy that at time 0 (when there are n jobs waiting for processing) does not act according to SEPT but at the first job completion (when there are $n-1$ jobs remaining to be processed) switches over to SEPT results in a larger total expected completion time than when SEPT is adopted immediately from time zero on.

Conditioning on the event that occurs at the first job completion yields

$$W = \min_{j,k} \Big(\frac{n}{\lambda_j + \lambda_k} + \frac{\lambda_j}{\lambda_j + \lambda_k} W^*(\{j\}) + \frac{\lambda_k}{\lambda_j + \lambda_k} W^*(\{k\}) \Big).$$

The increase in the total expected completion time until the first job completion is the first term on the RHS. (Recall that if during a time interval $[t_1, t_2]$ there are k jobs in the system that have not yet been completed, then the total completion time increases by an amount $k(t_2 - t_1)$.) The second (third) term is equal to the probability of job j (k) being the first job to be completed, multiplied by the total expected remaining completion time under SEPT. This last equation is equivalent to

$$0 = \min_{j,k} \Big(n + \lambda_j (W^*(\{j\}) - W^*) + \lambda_k (W^*(\{k\}) - W^*) + (\lambda_j + \lambda_k)(W^* - W) \Big).$$

It has to be shown now that if $\lambda_k > \lambda_j$, $\lambda_1 \geq \cdots \geq \lambda_n$, then

$$-1 \leq D_{jk} \leq 0$$

and

$$\frac{dD_{12}}{d\lambda_1} \leq 0.$$

This can be shown, as in Theorem 12.2.1, by induction. If $n = 2$, then $D_{12} = 0$ and the result holds. Doing similar manipulations as in Theorem 12.2.1 yields the following equations:

$$D_{12} = \frac{\lambda_1}{\lambda_1 + \lambda_2} \Big(D_{32}(\{1\}) - 1 \Big) + \frac{\lambda_2}{\lambda_1 + \lambda_2} \Big(D_{13}(\{2\}) + 1 \Big),$$

and

$$D_{2j} = \frac{\lambda_1}{\lambda_1 + \lambda_2} D_{2j}(\{1\}) + \frac{\lambda_2}{\lambda_1 + \lambda_2} (D_{3j}(\{2\}) - 1),$$

for $j = 3, \ldots, n$. Using these two equations, it is easy to complete the proof. □

Although the proof of the theorem is presented only for the case of two machines in parallel, the approach does work for an arbitrary number of machines in parallel, but again the notation gets more involved.

More can be said about the total expected completion time on m machines. Consider n processing times $X_1, \ldots, X_n$ from arbitrary distributions $F_1, \ldots, F_n$ such that $X_1 \leq_{st} X_2 \leq_{st} \cdots \leq_{st} X_n$. It has been shown in the literature that SEPT minimizes the total completion time in expectation and even stochastically. Recall that LEPT does *not* minimize the expected makespan when the $X_1, \ldots, X_n$ are arbitrarily distributed and stochastically ordered.

Consider now the problem of two machines in parallel with i.i.d. job processing times distributed exponentially with mean 1, with precedence constraints that have the form of an intree, and the expected makespan to be minimized in the class of preemptive dynamic policies (that is, a stochastic counterpart of $P2 \mid p_j = 1, intree \mid C_{\max}$). For the deterministic version of this problem the *Critical Path (CP)* rule (sometimes also referred to as the *Highest Level first (HL)* rule) is optimal. The CP rule is in the deterministic case optimal for an arbitrary number of machines in parallel, not just for two machines.

For the stochastic version the following notation is needed. The root of the intree is level 0. A job is at level k if there is a chain of $k-1$ jobs between it and the root of the intree. A precedence graph G_1 with n jobs is said to be *flatter* than a precedence graph G_2 with n jobs if the number of jobs at or below any given level k in G_1 is larger than the number of jobs at or below level k in graph G_2. This is denoted by $G_1 \succ_{fl} G_2$. In the following lemma two scenarios, both with two machines and n jobs but with different intrees, are compared. Let $E(C^{(i)}_{\max}(CP))$ denote the expected makespan under the CP rule when the precedence constraints graph takes the form of intree G_i, $i = 1, 2$.

In the subsequent lemma and theorem preemptions are allowed. However, it will become clear afterwards that for intree precedence constraints the CP rule does not require any preemptions. Also, recall that whenever a job is completed on one machine, the remaining processing time of the job being processed on the other machine is still exponentially distributed with mean one.

Lemma 12.2.3. *If $G_1 \succ_{fl} G_2$ then*

$$E(C^{(1)}_{\max}(CP)) \leq E(C^{(2)}_{\max}(CP))$$

Proof. The proof is by induction. If $n \leq 2$, it is clear that the lemma holds, since graph G_1 must consist of two jobs with no precedence relationship and graph G_2 must consist of a chain of two jobs.

Assume that the lemma holds for graphs of $n-1$ jobs. Assume that both graphs of n jobs allow two jobs to be started simultaneously at time zero on the two machines. If this is not the case, then graph G_2 has to be a chain of n jobs and in this special case the lemma does hold.

For both graphs the two jobs at the highest levels are selected. Suppose that in both graphs, the job at the very highest level is put on machine 1 and the other job is put on machine 2. Clearly, the job from G_1 on machine 1 is at a lower level than the job from G_2 on machine 1. The job from G_1 on machine 2 can be either at a higher or at a lower level than the job from G_2 on machine 2.

Suppose machine 1 is the first one to complete its job. The remaining time that the job on machine 2 needs to complete its processing is exponential with mean 1. So at the completion of the job on machine 1, both remaining graphs have $n-1$ jobs exponentially distributed with mean 1. It is clear that the remaining graphs are ordered in the same way as they were ordered originally, i.e., $G_1 \succ_{fl} G_2$ and the expected remaining processing time is less with G_1 than with G_2 because of the induction hypothesis for $n-1$ jobs.

Suppose machine 2 is the first one to complete its job. In order to show that the flatness property of the two remaining graphs is maintained consider two subcases. In the first subcase the job from G_2 on machine 2 is either at the same or at a lower level than the job from G_1 on machine 2. It is clear that when machine 2 is the first one to complete its processing the remaining part of G_1 is still flatter than the remaining part of G_2. The second subcase is a little bit more involved. Assume the job of G_2 on machine 2 is at level l and the job of G_1 on machine 2 is at level k, $k < l$. The number of jobs below or at level k (l) in G_1 is larger than the number of jobs below or at level k (l) in G_2. It is easy to see that the number of jobs below or at level k in G_1 plus $l-k$ is larger than or equal to the number of jobs below or at level l in G_2. So, at the completion of the job on machine 2, the number of jobs below or at level k in G_1 plus $l-k$ is still larger than or equal to the number of jobs below or at level l in G_2 ($k < l$). So the remaining graph of G_1 is still flatter than the remaining graph of G_2 Thus, because of the induction hypothesis, the expected time remaining till the completion of the last job is less under G_1 than under G_2. □

Theorem 12.2.4. *The nonpreemptive CP rule minimizes the expected makespan in the class of nonpreemptive dynamic policies and in the class of preemptive dynamic policies.*

Proof. The theorem is first shown to hold in the class of preemptive dynamic policies. The proof is by contradiction as well as by induction. Suppose the CP rule is optimal with $n-1$ jobs, but another policy, say policy π, is optimal with n jobs. This policy, at the first job completion, must switch over to the CP rule, as the CP rule is optimal with $n-1$ jobs. In the comparison between π and the CP rule, assume that under both policies the job at the higher level is put on machine 1 and the job at the lower level is put on machine 2. It is easy to check that, if machine 1 is the first one to complete its job, the remaining graph under the CP rule is flatter than the remaining graph under policy π. Because of Lemma 12.2.3 the expected makespan is therefore smaller under the CP rule than under policy π. It is also easy to check that if machine 2 is the first one to complete its job, the remaining graph under the CP rule is flatter than the remaining graph under policy π. Again, the expected makespan is therefore smaller under CP than under π. This proves the optimality of CP in the class of preemptive dynamic policies. It is clear that the CP rule even in the class of preemptive dynamic policies never causes any preemptions. The CP rule is therefore optimal in the class of nonpreemptive dynamic policies as well. □

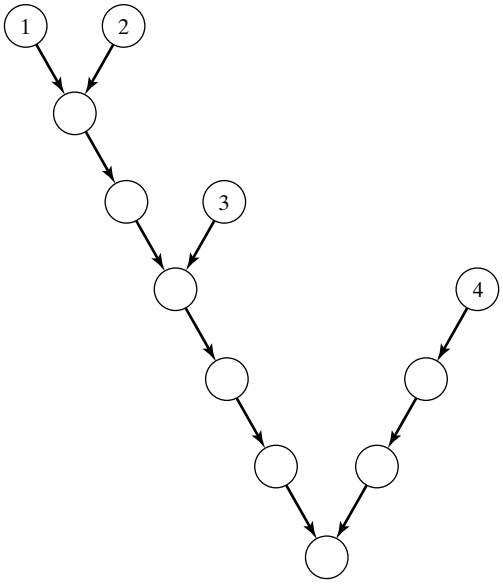


Fig. 12.2 The CP rule is not optimal for three machines (Example 12.2.5)

As mentioned before, the results presented in Theorems 12.1.2 and 12.2.1, even though they were only shown for $m = 2$, hold for arbitrary m. The CP rule in Theorem 12.2.4 is, however, not necessarily optimal for m larger than two.

Example 12.2.5 (Counterexample to Optimality of CP Rule with Three Machines)

Consider three machines and 12 jobs. The jobs are all i.i.d. exponential with mean 1 and subject to the precedence constraints described in Figure 12.2. Scheduling according to the CP rule would put jobs 1, 2 and 3 at time zero on the three machines. However, straightforward algebra shows that starting with jobs 1, 2 and 4 results in a smaller expected makespan (see Exercise 12.7). ||

In the deterministic setting discussed in Chapter 5 it was shown that the CP rule is optimal for $Pm \mid p_j = 1, intree \mid C_{\max}$ and $Pm \mid p_j = 1, outtree \mid C_{\max}$, for any m. One may expect the CP rule to be optimal when all processing times are exponential with mean 1 and precedence constraints take the form of an outtree. However, a counterexample can be found easily when there are two machines in parallel.

Consider once more a problem with two machines in parallel and jobs having i.i.d. exponentially distributed processing times subject to precedence con-

straints that take the form of an intree. But now the total expected completion time is the objective to be minimized.

Theorem 12.2.6. *The nonpreemptive CP rule minimizes the total expected completion time in the class of nonpreemptive dynamic policies and in the class of preemptive dynamic policies.*

Proof. The proof is similar to the proof of Theorem 12.2.4. A preliminary result similar to Lemma 12.2.3 is again needed. □

12.3 Due Date Related Objectives

Problems with due dates are significantly harder in a stochastic setting than in a deterministic setting. One of the reasons is that in a stochastic setting one cannot work backwards from the due dates as can be done in a deterministic setting. The actual realizations of the processing times and the due dates are now not known a priori and working backwards is therefore not possible.

However, some results can still be obtained. Consider m parallel machines and n jobs with all processing times being deterministic and equal to 1. The weight of job j is equal to w_j and the distribution of the due date of job j is F_j (arbitrary). The problem of determining the schedule that minimizes $E(\sum w_j U_j)$ in the class of nonpreemptive static list policies turns out to be equivalent to a deterministic assignment problem.

Theorem 12.3.1. *The nonpreemptive static list policy that minimizes $E(\sum w_j U_j)$ can be obtained by solving a deterministic assignment problem with the following cost matrix: if job j is assigned to position i in the permutation schedule, where $km + 1 \leq i \leq (k+1)m$, then the cost is $w_j F_j(k+1)$ for $k = 0, 1, 2, \ldots$ The assignment that minimizes the expected total cost corresponds to the optimal nonpreemptive static list policy.*

Proof. The first batch of m jobs in the list complete their processing at time 1, the second batch of m jobs at time 2, and so on. The probability that a job from the first batch of m jobs is overdue is $F_j(1)$, so the expected cost is $w_j F_j(1)$. The expected cost for a job from the second batch of m jobs is $w_j F_j(2)$, and so on. □

Consider now the case where the processing times of the n jobs are i.i.d. exponential with mean 1. Suppose the due date of job j is exponential with rate μ_j, but the due dates are not necessarily independent. Again, the objective is to minimize $E(\sum w_j U_j)$ with m identical machines in parallel.

Theorem 12.3.2. *The nonpreemptive static list policy that minimizes $E(\sum w_j U_j)$ can be obtained by solving a deterministic assignment problem with the following cost matrix: If job j is assigned to position i, $i = 1, \ldots, m$, on the*

list, then the expected cost is $w_j\mu_j/(1+\mu_j)$. *If job* j *is assigned to position* i, $i = m+1,\ldots,n$, *then the expected cost is*

$$w_j\Big(1-\Big(\frac{m}{m+\mu_j}\Big)^{i-m}\frac{1}{1+\mu_j}\Big).$$

The assignment that minimizes the expected total cost corresponds to the optimal nonpreemptive static list policy.

Proof. Observe that job j in slot $i = 1,\ldots,m$ starts at time zero. The probability that this job is not completed before its due date is $\mu_j/(1+\mu_j)$. So its expected cost is $w_j\mu_j/(1+\mu_j)$. Job j in slot i, $i = m+1,\ldots,n$ has to wait for $i-m$ job completions before it starts its processing. Given that all machines are busy, the time between successive completions is exponentially distributed with rate m. Thus the probability that a job in position $i > m$ starts its processing before its due date is $(m/(m+\mu_j))^{i-m}$. Consequently, the probability that it finishes before its due date is

$$\Big(\frac{m}{m+\mu_j}\Big)^{i-m}\frac{1}{1+\mu_j}.$$

So the probability that it is not completed by its due date is

$$1-\Big(\frac{m}{m+\mu_j}\Big)^{i-m}\frac{1}{1+\mu_j}$$

and thus has expected cost

$$w_j\Big(1-\Big(\frac{m}{m+\mu_j}\Big)^{i-m}\frac{1}{1+\mu_j}\Big).$$

□

12.4 Bounds Obtained through Online Scheduling

The online scheduling paradigm was first introduced in Chapter 5. In online scheduling it is assumed that the decision-maker has an extremely limited amount of information at his disposal. In the most common online scheduling paradigm, the decision-maker knows at any time t only the number of machines available (m), the number of jobs released so far, the number of jobs already completed, and the amounts of processing the remaining jobs already have received. The decision-maker has no information with regard to the future of the process. He does not know anything about the remaining processing times of the jobs that are not yet completed. He does not know how many jobs are still going to be released and what their release dates will be.

The amount of information a decision-maker has in an online scheduling environment is actually less than the amount of information a decision-maker has

in a stochastic scheduling environment. In a stochastic scheduling environment, a decision-maker has at least some information with regard to the distributions of the processing times and the distributions of the release dates. Also, knowing the original processing time distributions enables the decision-maker to determine the distributions of the remaining processing times of jobs that already have received a certain amount of processing.

This section describes how bounds obtained for online scheduling can be used to obtain bounds in stochastic scheduling. Consider the stochastic counterpart of $Pm \mid\mid C_{\max}$. Assume there are n jobs with random processing times $X_1, \ldots, X_n$ with arbitrary distributions $F_1, \ldots, F_n$. Suppose the decision-maker decides to adopt an arbitrary nonpreemptive dynamic policy π which does not allow for any unforced idleness. Let $E(C_{\max}(\pi))$ denote the expected makespan under policy π and let $E(C_{\max}(OPT))$ denote the expected makespan if the decision-maker adopts the optimal nonpreemptive dynamic policy. By extending the results obtained for online scheduling in Chapter 5, the following result can be shown for a stochastic scheduling environment with m identical machines in parallel.

Theorem 12.4.1. *For any nonpreemptive dynamic policy π that does not allow unforced idleness*

$$\frac{E(C_{\max}(\pi))}{E(C_{\max}(OPT))} \leq 2 - \frac{1}{m}.$$

Proof. Consider any realization $x_1, \ldots, x_n$ of the random variables $X_1, \ldots, X_n$. From Theorem 5.6.1 it follows that for this realization of processing times

$$\frac{C_{\max}(A)}{C_{\max}(OPT)} \leq 2 - \frac{1}{m}$$

for any deterministic rule A that does not allow unforced idleness.

Note that the optimal nonpreemptive dynamic policy in the stochastic setting would not necessarily generate for the processing times $x_1, \ldots, x_n$ the same schedule as the optimal deterministic scheduling rule (which knows the exact processing times a priori). The schedule generated by the optimal nonpreemptive dynamic policy for $x_1, \ldots, x_n$ may actually result in a makespan that is strictly *larger* than the makespan that would be generated by the optimal deterministic scheduling rule which knows the processing times $x_1, \ldots, x_n$ a priori.

Unconditioning with regard to the processing times results in the following: The numerator is an integral over all realizations of the processing times with the makespans generated by applying policy π and the denominator is an integral over all realizations with makespans generated by the optimal nonpreemptive dynamic policy. For each realization the ratio satisfies the inequality, so the inequality is satisfied for the ratio of the integrals as well. □

Consider now the stochastic counterpart of $Pm \mid prmp \mid \sum C_j$ with the n jobs having processing times that are arbitrarily distributed according to

$F_1, \ldots, F_n$. The result shown in Section 5.6 with regard to the *Round Robin* rule can be used for generating a bound on the total expected completion time under an arbitrary preemptive dynamic policy π.

The Round Robin rule is in a stochastic setting a well defined preemptive dynamic policy. Let $E(\sum_{j=1}^{n} C_j(RR))$ denote the total expected completion time under the Round Robin rule, let $E(\sum_{j=1}^{n} C_j(OPT))$ denote the total expected completion time under the optimal preemptive dynamic policy, and let $E(\sum_{j=1}^{n} C_j(\pi))$ denote the total expected completion time under an arbitrary preemptive dynamic policy π.

Theorem 12.4.2. *For any preemptive dynamic policy* π

$$\frac{E(\sum_{j=1}^{n} C_j(RR))}{E(\sum_{j=1}^{n} C_j(\pi))} \leq 2 - \frac{2m}{n+m}.$$

Proof. From Theorem 5.6.2 it follows that for any realization $x_1, \ldots, x_n$ of the n processing times

$$\frac{\sum_{j=1}^{n} C_j(RR)}{\sum_{j=1}^{n} C_j(OPT)} \leq 2 - \frac{2m}{n+m},$$

where $\sum_{j=1}^{n} C_j(OPT)$ is the total completion under the optimal preemptive deterministic scheduling rule for the processing times $x_1, \ldots, x_n$. Again, if the preemptive dynamic policy that is optimal in the stochastic setting would be applied to the specific instance $x_1, \ldots, x_n$, the total completion time most likely would exceed the total completion time of the optimal preemptive deterministic scheduling rule applied to the fixed values $x_1, \ldots, x_n$.

Unconditioning over the processing times yields

$$\frac{E(\sum_{j=1}^{n} C_j(RR))}{E(\sum_{j=1}^{n} C_j(OPT))} \leq 2 - \frac{2m}{n+m}.$$

Since

$$E(\sum_{j=1}^{n} C_j(OPT)) \leq E(\sum_{j=1}^{n} C_j(\pi))$$

the theorem follows. □

Note that the two bounds presented in Theorems 12.4.1 and 12.4.2 are of a different nature. Theorem 12.4.1 provides an upper bound for the worst case behaviour of an arbitrary nonpreemptive dynamic policy with respect to the expected makespan objective. Theorem 12.4.2, on the other hand, compares the performance of the Round Robin policy with that of an arbitrary preemptive dynamic policy. It provides an upper bound on the performance of the Round Robin policy relative to any policy. Or, equivalently, it provides a lower bound for the performance of an arbitrary policy's performance relative to the performance of the Round Robin policy.

Note that with respect to the total expected completion there is no upper bound on the performance ratio of an arbitrary policy relative to the optimal policy. To see why there is no upper bound, consider two machines in parallel and n jobs. Two jobs have a deterministic processing time of p (p being very large) and $n-2$ jobs have processing times that are very close to 0. The optimal rule is clearly the SEPT rule. Consider the performance of the LEPT rule. The total expected completion time under the LEPT rule divided by the total expected completion time under the SEPT rule can be made arbitrarily high.

12.5 Discussion

This chapter presents only a small sample of the results that have appeared in the literature concerning stochastic scheduling on parallel machines.

A significant amount of research has focused on the uniform machine case, i.e., m machines in parallel with different speeds. Preemptive as well as nonpreemptive models have been considered. The preemptive models tend to be somewhat easier. It has been shown that under certain conditions the preemptive dynamic policy that always assigns the Longest Expected Remaining Processing Time to the Fastest Machine minimizes the expected makespan while the preemptive dynamic policy that always assigns the Shortest Expected Remaining Processing Time to the Fastest Machine minimizes the total expected completion time.

The nonpreemptive models are considerably harder. A distinction can be made between models that allow unforced idleness and models that do not allow unforced idleness. In a model that allows unforced idleness the decision-maker may decide to keep a slow machine idle and let a job wait for a faster machine to become available. Under certain conditions it has been shown that threshold policies are optimal. That is, the decision to let a job wait for a faster machine to become available depends on the expected processing time of the job, the differences in the speeds of the machines, the total expected processing time still to be done by the other machine(s), and so on.

Exercises (Computational)

12.1. Consider two machines in parallel and four jobs with exponentially distributed processing times with means 1/5, 1/4, 1/3 and 1/2.

(a) Compute the probability of each job being the last one to finish under the LEPT rule.

(b) Do the same for the SEPT rule.

12.2. Consider the same scenario as in the previous exercise and compute the expected makespan under LEPT and under SEPT.

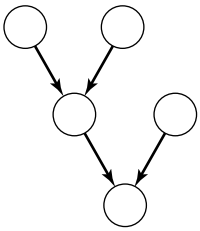


Fig. 12.3 Intree (Exercise 12.4)

12.3. Consider the same scenario as in Exercise 12.1. Compute the total expected completion time under SEPT and LEPT. (*Hint:* note that the sum of the expected completion times is equal to the sum of the expected starting times plus the sum of the expected processing times. So it suffices to compute the four expected starting times (two of which being zero)).

12.4. Consider two machines in parallel and 5 jobs under the intree precedence constraints depicted in Figure 12.3. All processing times are i.i.d. exponential with mean 1. Compute the expected makespan under the CP rule and under a different schedule.

12.5. Consider two machines and n jobs. The processing times of the n jobs are i.i.d. and distributed according to a mixture of exponentials: $P(X_j = 0) = 1 - p$ and with probability p the processing time is exponentially distributed with rate p (the value p is not necessarily close to 0). Compute the expected makespan under a nonpreemptive schedule and under a preemptive schedule.

12.6. Consider the same scenario as in the previous exercise.

(a) Compute the total expected completion time for the same set of jobs under a nonpreemptive schedule and under a preemptive schedule.

(b) Compare the result to the case where all n processing times are deterministic and equal to 1.

12.7. Consider Example 12.2.5 with three machines and 12 jobs. Compute the expected makespan under the CP rule and under the rule suggested in Example 12.2.5.

12.8. Find a counterexample showing that the nonpreemptive SEPT rule does not necessarily minimize the total expected completion time in the class of nonpreemptive static list policies when the processing times are merely ordered in expectation and not ordered stochastically.

12.9. Consider the same scenario as in Exercise 12.1. The four jobs are geometrically distributed with means 1/5, 1/4, 1/3, 1/2 (i.e., q is 1/4, 1/3, 1/2, 1).

(a) Compute the probability of each job being the last one to complete its processing under LEPT (if two jobs finish at the same time, then no job finishes last).

(b) Compute the expected makespan and compare it to the expected makespan obtained in Exercise 12.2.

12.10. Consider 2 machines and 4 jobs. The 4 jobs are i.i.d. according to the EME distribution with mean 1 (recall that the probability p in the definition of the EME distribution is very close to zero, so $p^2 \ll p$). Compute the expected makespan as a function of p (disregard all terms of p^2 and smaller).

Exercises (Theory)

12.11. Consider m machines in parallel and n jobs. The processing times of the n jobs are i.i.d. exponential with mean 1. Find expressions for $E(C_{\max})$ and $E(\sum C_j)$.

12.12. Consider m machines in parallel and n jobs. The processing times of the n jobs are i.i.d. exponential with mean 1. The jobs are subject to precedence constraints that take the form of chains of different lengths. Find the policy that minimizes the expected makespan and prove that it results in an optimal schedule.

12.13. Consider n jobs and 2 machines in parallel. The processing time of job j is with probability p_j exponentially distributed with rate λ. The processing time is with probability q_j distributed according to a convolution of an exponential with rate μ and an exponential with rate λ. The processing time is zero with probability $1 - p_j - q_j$. Show that scheduling the jobs in decreasing order of q_j/p_j minimizes the expected makespan (*Hint:* Every time a machine is freed the other machine can be only in one of two states: the remaining processing time of the job on that machine is either exponentially distributed with rate λ or the remaining processing time is distributed according to a convolution of two exponentials with rates μ and λ).

12.14. Consider the processing time distribution in Example 12.1.7. Show that

$$Var(X_j) = 1 + 4a_j.$$

12.15. Consider two machines and n jobs. The processing times of the n jobs are i.i.d. exponential with mean 1. To process job j, an amount ρ_j, $0 \leq \rho_j \leq 1$, of an additional resource is needed. The total amount of that resource available at any point in time is 1. Formulate the policy that minimizes the expected makespan and show that it leads to the optimal schedule.

12.16. Consider n jobs and 2 machines in parallel. The processing time of job j is with probability p_j exponentially distributed with rate λ_1, and with

probability $1 - p_j$ exponentially distributed with rate λ_2. Job j has a weight w_j. Show that if the WSEPT rule results in the same sequence as the LEPT rule, then the WSEPT rule minimizes the total expected weighted completion time.

12.17. Consider 2 machines in parallel and 5 jobs. The processing time of job j is 1 with probability p_j and 2 with probability $1 - p_j$.

(a) Show that the random variable D (as defined in Section 12.1, can only assume values 0, 1 or 2.

(b) Show that the probability that the random variable D assumes the value 1 is equal to the probability that the sum of the n processing times is odd and therefore independent of the schedule.

(c) Show that minimizing the expected makespan is equivalent to minimizing $P(D = 2)$ and maximizing $P(D = 0)$.

(d) Find the optimal sequence.

12.18. Consider two machines in parallel and n jobs. The processing time of job j is zero with probability p_{0j}, 1 with probability p_{1j} and 2 with probability $p_{2j} = 1 - p_{1j} - p_{0j}$. Show through a counterexample that the Largest Variance first (LV) rule is not necessarily optimal.

12.19. Consider the two machine setting in Example 12.1.7. Show that the Largest Variance first rule minimizes $E(\sum C_j)$ with two machines in parallel.

12.20. Consider 2 machines and n jobs. Assume all jobs have the same fixed due date d. Show through counterexamples that there are cases where neither SEPT nor LEPT stochastically maximize the number of jobs completed in time.

12.21. Consider m machines in parallel and n jobs. The processing times of all n jobs are i.i.d. according to a mixture of exponentials. The processing time of job j is zero with probability p and exponential with mean 1 with probability $1 - p$. The due date of job j is exponential with rate μ_j. Show that determining the optimal nonpreemptive static list policy is identical to a deterministic assignment problem. Describe the cost structure of this assignment problem.

12.22. Consider the same setting as in the previous problem. However, the machines are now subject to breakdowns. The up-times are i.i.d. exponentially distributed with rate ν and the down times are also i.i.d. exponential with a different rate. Show that determining the optimal permutation schedule is again equivalent to a deterministic assignment problem. Describe the cost structure of this assignment problem.

Comments and References

Many researchers have worked on the scheduling of exponential jobs on identical machines in parallel, see Pinedo and Weiss (1979), Weiss and Pinedo

(1980), Bruno, Downey and Frederickson (1981), Gittins (1981), Van der Heyden (1981), Pinedo (1981a), Weber (1982a, 1982b), Weber, Varaya, Walrand (1986), Kämpke (1987a, 1987b, 1989) Chang, Nelson and Pinedo (1992) and Chang, Chao, Pinedo and Weber (1992). Most of these papers deal with both the makespan and the total completion time objectives.

The interchange approach described here, showing that LEPT minimizes the expected makespan when the processing times are exponentially distributed, is based on the paper by Pinedo and Weiss (1979). This paper discusses also hyperexponential distributions that are mixtures of two exponentials. An analysis of the LV rule appears in Pinedo and Weiss (1987).

The dynamic programming type approach showing that LEPT minimizes the expected makespan and SEPT the total expected completion time in a preemptive setting with two machines is based on the paper by Weber (1982b). More general results appear in Weiss and Pinedo (1980), Weber(1982a), Kämpke (1987a, 1987b, 1989), Chang, Chao, Pinedo and Weber (1992) and Righter (1988, 1992). The most general results with regard to the minimization of the total (unweighted) completion time are probably the ones obtained by Weber, Varaya and Walrand (1986). Weiss (1990) analyzes approximation techniques for parallel machines and Möhring, Schulz and Uetz (1999) study LP-based priority policies for the total weighted completion time objective. The fact that the CP rule minimizes the expected makespan in a preemptive as well as a nonpreemptive setting is shown first by Chandy and Reynolds (1975). Pinedo and Weiss (1984) obtain more general results allowing processing times at the different levels of the intree to have different means. Frostig (1988) generalizes this result even further, allowing distributions other than exponential.

There is an extensive literature on the scheduling of non-identical machines in parallel, i.e., machines with different speeds, which has not been discussed in this chapter. This research focuses on preemptive as well as on nonpreemptive models with as objectives either the makespan or the total completion time. See, for example, Agrawala, Coffman, Garey and Tripathi (1984), Coffman, Flatto, Garey and Weber (1987), Righter (1988), Righter and Xu (1991a, 1991b), Xu, Mirchandani, Kumar and Weber (1990) and Xu (1991a, 1991b).

The due date related results in the last section are based on results in a paper by Emmons and Pinedo (1990). For other due date related results, see Chang, Chao, Pinedo and Weber (1992) and Xu (1991b).

Chapter 13

Flow Shops, Job Shops and Open Shops (Stochastic)

The results for stochastic flow shops, job shops, and open shops are somewhat less extensive than those for their deterministic counterparts.

This chapter focuses first on nonpreemptive static list policies, i.e., permutation schedules, for stochastic flow shops. The optimal permutation schedules often remain optimal in the class of nonpreemptive dynamic policies as well as in the class of preemptive dynamic policies. For open shops and job shops, only the classes of nonpreemptive dynamic policies and preemptive dynamic policies are considered.

The results obtained for stochastic flow shops and job shops are somewhat similar to those obtained for deterministic flow shops and job shops. Stochastic open shops are, however, very different from their deterministic counterparts.

The first section discusses stochastic flow shops with unlimited intermediate storage and jobs not subject to blocking. The second section deals with stochastic flow shops with zero intermediate storage; the jobs are subject to blocking. The third section focuses on stochastic job shops and the last section goes over stochastic open shops.

M.L. Pinedo, *Scheduling*, DOI: 10.1007/978-0-387-78935-4_13,

13.1 Stochastic Flow Shops with Unlimited Intermediate Storage

Consider two machines in series with unlimited storage between the machines and no blocking. There are n jobs. The processing time of job j on machine 1 is X_{1j}, exponentially distributed with rate λ_j. The processing time of job j on machine 2 is X_{2j}, exponentially distributed with rate μ_j. The objective is to find the nonpreemptive static list policy or permutation schedule that minimizes the expected makespan $E(C_{\max})$.

Note that this problem is a stochastic counterpart of the deterministic problem $F2 \mid\mid C_{\max}$. The deterministic two machine problem has a very simple solution. It turns out that the stochastic version with exponential processing times has a very elegant solution as well.

Theorem 13.1.1. *Sequencing the jobs in decreasing order of $\lambda_j - \mu_j$ minimizes the expected makespan in the class of nonpreemptive static list policies, in the class of nonpreemptive dynamic policies, and in the class of preemptive dynamic policies.*

Proof. The proof of optimality in the class of nonpreemptive static list policies is in a sense similar to the proof of optimality in the deterministic case. It is by contradiction. Suppose another sequence is optimal. Under this sequence, there must be two adjacent jobs, say job j followed by job k, such that $\lambda_j - \mu_j < \lambda_k - \mu_k$. It suffices to show that a pairwise interchange of these two jobs reduces the expected makespan. Assume job l precedes job j and let C_{1l} (C_{2l}) denote the (random) completion time of job l on machine 1 (2). Let $D_l = C_{2l} - C_{1l}$.

Perform an adjacent pairwise interchange on jobs j and k. Let C_{1k} and C_{2k} denote the completion times of job k on the two machines under the original, supposedly optimal, sequence and let C'_{1j} and C'_{2j} denote the completion times of job j under the schedule obtained after the pairwise interchange. Let m denote the job following job k. Clearly, the pairwise interchange does not affect the starting time of job m on machine 1 as this starting time is equal to $C_{1k} = C'_{1j} = C_{1l} + X_{1j} + X_{1k}$. Consider the random variables

$$D_k = C_{2k} - C_{1k}$$

and

$$D'_j = C'_{2j} - C'_{1j}.$$

Clearly, $C_{1k} + D_k$ is the time at which machine 2 becomes available for job m under the original schedule, while $C_{1k} + D'_j$ is the corresponding time after the pairwise interchange. First it is shown that the random variable D'_j is stochastically smaller than the random variable D_k. If $D_l \geq X_{1j} + X_{1k}$, then clearly $D_k = D'_j$. The case $D_l \leq X_{1j} + X_{1k}$ is slightly more complicated. Now

$$P(D_k > t \mid D_l \leq X_{1j} + X_{1k}) = \frac{\mu_j}{\lambda_k + \mu_j} e^{-\mu_k t}$$

$$+\frac{\lambda_k}{\lambda_k+\mu_j}\Big(\frac{\mu_k}{\mu_k-\mu_j}e^{-\mu_j t}-\frac{\mu_j}{\mu_k-\mu_j}e^{-\mu_k t}\Big).$$

This expression can be explained as follows. If $D_l \le X_{1j}+X_{1k}$, then, whenever job j starts on machine 2, job k is either being started or still being processed on machine 1. The first term on the R.H.S. corresponds to the event where job j's processing time on machine 2 finishes before job k's processing time on machine 1, which happens with probability $\mu_j/(\mu_j+\lambda_k)$. The second term corresponds to the event where job j finishes on machine 2 after job k finishes on machine 1; in this case the distribution of D_k is a convolution of an exponential with rate μ_j and an exponential with rate μ_k.

An expression for $P(D'_j > t \mid D_l \le X_{1j}+X_{1k})$ can be obtained by interchanging the subscripts j with the subscripts k. Now

$$\begin{aligned}P(D'_j > t \mid D_l \le X_{1j}+X_{1k}) - P(D_k > t \mid D_l \le X_{1j}+X_{1k})\\ = \frac{\mu_j\mu_k}{(\lambda_j+\mu_k)(\lambda_k+\mu_j)}\frac{e^{-\mu_j t}-e^{-\mu_k t}}{\mu_k-\mu_j}(\lambda_j+\mu_k-\lambda_k-\mu_j) \le 0.\end{aligned}$$

So D'_j is stochastically smaller than D_k. It can be shown easily, through a straightforward sample path analysis (i.e., fixing the processing times of job m and of all the jobs following job m), that if the realization of D'_j is smaller than the realization of D_k, then the actual makespan after the interchange is smaller than or equal to the actual makespan under the original sequence before the interchange. So, given that D'_j is stochastically smaller than D_k, the expected makespan is reduced by the interchange. This completes the proof of optimality in the class of nonpreemptive static list (i.e., permutation) policies.

That the rule is also optimal in the class of nonpreemptive dynamic policies can be argued as follows. It is clear that the sequence on machine 2 does not matter. This is because the time machine 2 remains busy processing available jobs is simply the sum of their processing times and the order in which this happens does not affect the makespan. Consider the decisions that have to be made every time machine 1 is freed. The last decision to be made occurs at that point in time when there are only two jobs remaining to be processed on machine 1. From the pairwise interchange argument described above, it immediately follows that the job with the highest $\lambda_j-\mu_j$ value has to go first. Suppose that there are three jobs remaining to be processed on machine 1. From the previous argument it follows that the last two of these three have to be processed in decreasing order of $\lambda_j-\mu_j$. If the first one of the three is not the one with the highest $\lambda_j-\mu_j$ value, a pairwise interchange between the first and the second reduces the expected makespan. So the last three jobs have to be sequenced in decreasing order of $\lambda_j-\mu_j$. Continuing in this manner it is shown that sequencing the jobs in decreasing order of $\lambda_j-\mu_j$ is optimal in the class of nonpreemptive dynamic policies.

That the nonpreemptive rule is also optimal in the class of preemptive dynamic policies can be shown as follows. It is shown above that in the class of

nonpreemptive dynamic policies the optimal rule is to order the jobs in decreasing order of $\lambda_j - \mu_j$. Suppose during the processing of a job on machine 1 a preemption is considered. The situation at this point in time is essentially no different from the situation at the point in time the job was started (because of the memoryless property of the exponential distribution). So, every time a preemption is contemplated, the optimal decision is to keep the current job on the machine. Thus the permutation policy is also optimal in the class of preemptive dynamic policies. □

From the statement of the theorem, it appears that the number of optimal schedules in the exponential case is often smaller than the number of optimal schedules in the deterministic case.

Example 13.1.2 (Comparison between Exponential and Deterministic Processing Times)

Consider n jobs with exponentially distributed processing times. One job has a zero processing time on machine 1 and a processing time with a very large mean on machine 2. Assume that this mean is larger than the sum of the expected processing times of the remaining $n-1$ jobs on machine 1. According to Theorem 13.1.1 these remaining $n-1$ jobs still have to be ordered in decreasing order of $\lambda_j - \mu_j$ for the sequence to have a minimum expected makespan.

If all the processing times were deterministic with processing times that were equal to the means of the exponential processing times, then it would not have mattered in what order the remaining $n-1$ jobs were sequenced. ||

Although at first glance Theorem 13.1.1 does not appear to be very similar to Theorem 6.1.4, the optimal schedule with exponential processing times is somewhat similar to the optimal schedule with deterministic processing times. If job k follows job j in the optimal sequence with exponential processing times, then

$$\lambda_j - \mu_j \geq \lambda_k - \mu_k$$

or

$$\lambda_j + \mu_k \geq \lambda_k + \mu_j$$

or

$$\frac{1}{\lambda_j + \mu_k} \leq \frac{1}{\lambda_k + \mu_j},$$

which, with exponential processing times, is equivalent to

$$E(\min(X_{1j}, X_{2k})) \leq E(\min(X_{1k}, X_{2j}))$$

(see Exercise 9.13). This adjacency condition is quite similar to the condition for job k to follow job j in a deterministic setting, namely

$$\min(p_{1j}, p_{2k}) \leq \min(p_{1k}, p_{2j}).$$

There is another similarity between exponential and deterministic settings. Consider the case where the processing times of job j on both machines are i.i.d. exponentially distributed with the same rate, λ_j, for each j. According to the theorem all sequences must have the same expected makespan. This result is similar to the one for the deterministic proportionate flow shop, where all sequences also have the same makespan.

The remaining part of this section focuses on m machine *permutation* flow shops. For these flow shops only the class of nonpreemptive static list policies is of interest, since the order of the jobs, once determined, is not allowed to change.

Consider an m machine permutation flow shop where the processing times of job j on the m machines are i.i.d. according to distribution F_j with mean $1/\lambda_j$. For such a flow shop it is easy to obtain a lower bound for $E(C_{\max})$.

Lemma 13.1.3. *For any sequence*

$$E(C_{\max}) \geq \sum_{j=1}^{n} \frac{1}{\lambda_j} + (m-1) \max\left(\frac{1}{\lambda_1}, \ldots, \frac{1}{\lambda_n}\right)$$

Proof. The expected time it takes the job with the longest expected processing time to traverse the flow shop is at least $m \times \max(1/\lambda_1, \ldots, 1/\lambda_n)$. This longest job starts on machine 1 at a time that is equal to the sum of the processing times on the first machine of the jobs preceding it. After the longest job completes its processing on the last machine, this machine remains busy for a time that is at least as long as the sum of the processing times on the last machine of the jobs succeeding it. The lemma thus follows. □

One class of sequences plays an important role in stochastic permutation flow shops. A sequence $j_1, \ldots, j_n$ is called a SEPT-LEPT sequence, if there is a job j_k in the sequence such that

$$\frac{1}{\lambda_{j_1}} \leq \frac{1}{\lambda_{j_2}} \leq \cdots \leq \frac{1}{\lambda_{j_k}}$$

and

$$\frac{1}{\lambda_{j_k}} \geq \frac{1}{\lambda_{j_{k+1}}} \geq \cdots \geq \frac{1}{\lambda_{j_n}}.$$

Both the SEPT and the LEPT sequence are examples of SEPT-LEPT sequences.

Theorem 13.1.4. *If $F_1 \leq_{a.s.} F_2 \leq_{a.s.} \cdots \leq_{a.s.} F_n$, then*
(i) any SEPT-LEPT sequence minimizes the expected makespan in the class of nonpreemptive static list policies and

$$E(C_{\max}) = \sum_{j=1}^{n} \frac{1}{\lambda_j} + (m-1)\frac{1}{\lambda_n}.$$

(ii) the SEPT sequence minimizes the total expected completion time in the class of nonpreemptive static list policies and

$$E(\sum_{j=1}^{n} C_j) = m \sum_{j=1}^{n} \frac{1}{\lambda_j} + \sum_{j=1}^{n-1} \frac{j}{\lambda_{n-j}}.$$

Proof. (i) The first part of the theorem follows from the observation that the jobs in the SEPT segment of the sequence *never* have to wait for a machine while they go through the system. This includes the longest job, i.e., the last job of the SEPT segment of the sequence. The jobs in the LEPT segment of the sequence (excluding the first, i.e., the longest job) have to wait for each machine; that is, they complete their processing on one machine before the next one becomes available. The machines then never have to wait for a job. The makespan is therefore equal to the lower bound established in Lemma 13.1.3.

(ii) The second part follows from the fact that the SEPT sequence is one of the sequences that minimize the expected makespan. In order to minimize the expected completion time of the job in the kth position in the sequence, the k smallest jobs have to go first and these k jobs have to be sequenced according to a sequence that minimizes the expected completion time of this kth job. So these k smallest jobs may be sequenced according to any SEPT-LEPT sequence including SEPT. This is true for any k. It is clear that under SEPT the expected completion time of the job in each one of the n positions in the sequence is minimized. SEPT therefore minimizes the total expected completion time. The actual value of the total expected completion time is easy to compute. □

It is easy to find examples with $F_1 \leq_{a.s.} F_2 \leq_{a.s.} \cdots \leq_{a.s.} F_n$, where sequences that are not SEPT-LEPT are also optimal. (In Theorem 6.1.8 it was shown that when $F_1, F_2, \ldots, F_n$ are deterministic all sequences are optimal.) However, in contrast to deterministic proportionate flow shops, when processing times are stochastic and $F_1 \leq_{a.s.} F_2 \leq_{a.s.} \cdots \leq_{a.s.} F_n$ not *all* sequences are optimal.

Example 13.1.5 (Optimality of SEPT-LEPT and Other Sequences)
Consider a flow shop with 2 machines and 3 jobs. Job 1 has a deterministic processing time of 11 time units. Job 2 has a deterministic processing time of 10 time units. The processing time of job 3 is zero with probability 0.5 and 10 with probability 0.5. It can be verified easily that *only* SEPT-LEPT sequences minimize the expected makespan. If the processing time of job 1 is changed from 11 to 20, then all sequences have the same expected makespan. ||

The model in Theorem 13.1.4 assumes that the processing times of job j on the m machines are obtained by m *independent* draws from the same distribution F_j. If the m processing times of job j are all equal to the *same* random

variable drawn from distribution F_j, then the model resembles the deterministic proportionate flow shop even more closely. So in this case

$$X_{1j} = X_{2j} = \cdots = X_{mj} = X_j.$$

It is easy to see that in this case, just like in the case of a deterministic proportionate flow shop, the makespan is independent of the job sequence (no particular form of stochastic dominance has to be imposed on the n distributions for this to be true).

Consider the case where the processing times of job j on each one of the m machines are equal to the *same* random variable X_j from distribution F_j and assume

$$F_1 \leq_{cx} F_2 \leq_{cx} \cdots \leq_{cx} F_n.$$

The expectations of the n processing times are therefore all the same, but the variances may be different. Let the objective to be minimized be the total expected completion time $E(\sum_{j=1}^{n} C_j)$. This problem leads to the application of the so-called *Smallest Variance first (SV)* rule, which selects, whenever machine 1 is freed, among the remaining jobs the one with the smallest variance.

Theorem 13.1.6. *The SV rule minimizes the total expected completion time in the class of nonpreemptive static list policies.*

Proof. The proof is by contradiction. Suppose another sequence is optimal. In this supposedly optimal sequence there has to be a job j followed by a job k such that $X_j \geq_{cx} X_k$. Assume job h precedes job j and job l follows job k. Let C_{ih} denote the completion time of job h on machine i. Let A denote the sum of the processing times of all the jobs preceding and including job h in the sequence and let B denote the maximum processing time among the jobs preceding and including job h. Then

$$C_{1h} = A$$

and

$$C_{ih} = A + (i-1)B.$$

Performing an interchange between jobs j and k does not affect the profile job l encounters upon entering the system. So an interchange does not affect the completion times of job l or the jobs following job l in the sequence. The pairwise interchange only affects the sum of the expected completion times of jobs j and k. The completion time of job k before the interchange is equal to the completion time of job j after the interchange. In order to analyze the effect of the pairwise interchange it suffices to compare the completion time of job j before the interchange with the completion time of job k after the interchange. Let C_{ij} denote the completion time of job j before the interchange and C'_{ik} the completion time of job k after the interchange. Clearly,

$$C_{mj} = A + X_j + (m-1)\max(B, X_j)$$

and

$$C'_{mk} = A + X_k + (m-1)\max(B, X_k).$$

The expected completion time of job k after the pairwise interchange is smaller than the expected completion time of job j before the interchange if

$$\int_0^\infty \Big(A+t+(m-1)\max(B,t)\Big)dF_k(t) \leq \int_0^\infty \Big(A+t+(m-1)\max(B,t)\Big)dF_j(t)$$

The integrand is a function that is increasing convex in t. So the inequality indeed holds if $F_k \leq_{cx} F_j$. This implies that the original sequence cannot be optimal and the proof is complete. □

This result is in contrast to the result stated in Exercise 12.19, where in the case of machines in parallel the Largest Variance first rule minimizes the total expected completion time.

13.2 Stochastic Flow Shops with Blocking

Consider the stochastic counterpart of $F2 \mid \mathit{block} \mid C_{\max}$ with the processing time of job j on machine 1 (2) being a random variable X_{1j} (X_{2j}) from distribution F_{1j} (F_{2j}). There is zero intermediate storage between the two machines. The objective is to minimize the expected makespan in the class of nonpreemptive static list policies.

When a job starts its processing on machine 1, the preceding job in the sequence starts its processing on machine 2. If job j follows job k in the sequence, then the expected time that job j remains on machine 1, either being processed or being blocked, is $E(\max(X_{1j}, X_{2k}))$. If job j is the first job in the sequence then job j spends only an expected amount of time $E(X_{1j})$ on machine 1 while machine 2 remains idle. If job j is the last job in the sequence, then it spends an expected amount of time $E(X_{2j})$ on machine 2 while machine 1 remains idle. In the same way that the deterministic $F2 \mid \mathit{block} \mid C_{\max}$ problem is equivalent to a deterministic Travelling Salesman Problem, this stochastic model is equivalent to a *deterministic* Travelling Salesman Problem. However, the efficient algorithm described in Section 4.4, which is applicable to the deterministic $F2 \mid \mathit{block} \mid C_{\max}$ problem, is not applicable to the stochastic version of the model. The distance matrix of the Travelling Salesman Problem is determined as follows:

$$\begin{aligned}
d_{0k} &= E(X_{1k}) \\
d_{j0} &= E(X_{2j}) \\
d_{jk} &= E(\max(X_{2j}, X_{1k})) \\
&= E(X_{2j}) + E(X_{1k}) - E(\min(X_{2j}, X_{1k})) \\
&= \int_0^\infty \bar{F}_{2j}(t)\,dt + \int_0^\infty \bar{F}_{1k}(t)\,dt - \int_0^\infty \bar{F}_{2j}(t)\bar{F}_{1k}(t)\,dt
\end{aligned}$$

It is clear that a value for d_{jk} can be computed, but that this value is now not a simple function of two parameters like in Sections 4.4 and 6.2. However, the Travelling Salesman Problem described above can be simplified somewhat. The problem is equivalent to a Travelling Salesman Problem with a simpler distance matrix in which the total distance has to be *maximized*. The distance matrix is modified by subtracting the expected sum of the $2n$ processing times from the distances and multiplying the remaining parts by -1.

$$\begin{aligned} d_{0k} &= 0 \\ d_{j0} &= 0 \\ d_{jk} &= E(\min(X_{2j}, X_{1k})) \\ &= \int_0^\infty \bar{F}_{2j}(t)\bar{F}_{1k}(t)\,dt \end{aligned}$$

Example 13.2.1 (Flow Shop with Blocking and Exponential Processing Times)

Consider the case where F_{1j} is exponentially distributed with rate λ_j and F_{2j} is exponentially distributed with rate μ_j. The distance

$$d_{jk} = E(\min(X_{2j}, X_{1k})) = \int_0^\infty \bar{F}_{2j}(t)\bar{F}_{1k}(t)dt = \frac{1}{\lambda_k + \mu_j}.$$

Although this deterministic Travelling Salesman Problem (in which the total distance must be maximized) still has a fairly nice structure, it has been shown that it cannot be solved in polynomial time. ||

It is of interest to study special cases of the problem, with additional structure, in order to obtain more insight. Consider the case where $F_{1j} = F_{2j} = F_j$. The random variables X_{1j} and X_{2j} are independent draws from distribution F_j. This model is somewhat similar to the deterministic proportionate flow shop model since the distributions of the processing times of any given job on the two machines are identical. However, the actual realizations of the two processing times are not necessarily identical.

Theorem 13.2.2. *If $F_1 \leq_{st} F_2 \leq_{st} \cdots \leq_{st} F_n$ then the sequences $1, 3, 5, \ldots, n, \ldots, 6, 4, 2$ and $2, 4, 6, \ldots, n, \ldots, 5, 3, 1$ minimize the expected makespan in the class of nonpreemptive static list policies.*

Proof. Consider the sequence

$$j_1, \ldots, j_{k-1},\ \ j_k, j_{k+1}, \ldots, j_{l-1}, j_l,\ \ j_{l+1}, \ldots, j_n.$$

Partition the sequence into three subsequences, the first being $j_1, \ldots, j_{k-1}$, the second $j_k, \ldots, j_l$ and the third $j_{l+1}, \ldots, j_n$. Construct a new sequence by re-

versing the second subsequence. The new sequence is

$$j_1, \ldots, j_{k-1}, \;\; j_l, j_{l-1}, \ldots, j_{k+1}, j_k, \;\; j_{l+1}, \ldots, j_n.$$

If $E(C_{\max})$ denotes the expected makespan of the original sequence and $E(C'_{\max})$ the expected makespan of the new sequence, then

$$\begin{aligned} E(C_{\max}) - E(C'_{\max}) = \;& E(\max(X_{2,j_{k-1}}, X_{1,j_k})) + E(\max(X_{2,j_l}, X_{1,j_{l+1}})) \\ & - E(\max(X_{2,j_{k-1}}, X_{1,j_l})) - E(\max(X_{2,j_k}, X_{1,j_{l+1}})) \\ = \;& -E(\min(X_{2,j_{k-1}}, X_{1,j_k})) - E(\min(X_{2,j_l}, X_{1,j_{l+1}})) \\ & + E(\min(X_{2,j_{k-1}}, X_{1,j_l})) + E(\min(X_{2,j_k}, X_{1,j_{l+1}})) \end{aligned}$$

So the expected makespan of the second sequence is less than the expected makespan of the first sequence if

$$\int_0^\infty \Big(\bar{F}_{j_{k-1}}(t)\bar{F}_{j_k}(t) + \bar{F}_{j_l}(t)\bar{F}_{j_{l+1}}(t)\Big)dt < \int_0^\infty \Big(\bar{F}_{j_{k-1}}(t)\bar{F}_{j_l}(t) + \bar{F}_{j_k}(t)\bar{F}_{j_{l+1}}(t)\Big)dt$$

Note that the makespan under an arbitrary sequence does not change if two jobs are added, both with zero processing times on the two machines, one scheduled first and the other one last. The processing time distributions of these two jobs are stochastically less than the distribution of any one of the other n jobs. So in the proof an assumption can be made that there are two additional jobs with zero processing times and that one of these jobs goes first and the other one goes last. In what follows these two jobs are referred to as jobs 0 and $0'$.

Consider four processing time distributions $F_j \geq_{st} F_k \geq_{st} F_p \geq_{st} F_q$. It can be easily verified that

$$\begin{aligned} \int_0^\infty \Big(\bar{F}_j(t)\bar{F}_k(t) + \bar{F}_p(t)\bar{F}_q(t)\Big)dt &\geq \int_0^\infty \Big(\bar{F}_j(t)\bar{F}_p(t) + \bar{F}_k(t)\bar{F}_q(t)\Big)dt \\ &\geq \int_0^\infty \Big(\bar{F}_j(t)\bar{F}_q(t) + \bar{F}_k(t)\bar{F}_p(t)\Big)dt. \end{aligned}$$

The remaining part of the proof is based on a contradiction argument. An arbitrary sequence that is not according to the theorem can be improved through a series of successive subsequence reversals until a sequence of the theorem is obtained. Consider a sequence

$$0, \; j_1, \ldots, j_k, 1, \; j_{k+1}, \ldots, j_l, 2, j_{l+1}, \ldots, j_{n-2}, 0',$$

where $j_1, \ldots, j_{n-2}$ is a permutation of $3, 4, \ldots, n$. From the inequalities above it follows that a subsequence reversal results in the sequence

$$0,\ 1, j_k, \ldots, j_1,\ j_{k+1}, \ldots, j_l,\ 2, j_{l+1}, \ldots, j_{n-2},\ 0'$$

with a smaller expected makespan. The makespan can be reduced even further through a second subsequence reversal that results in

$$0, 1, j_k, \ldots, j_1, j_{k+1}, \ldots, j_l,\ j_{n-2}, \ldots, j_{l+1}, 2,\ 0'.$$

Proceeding in this manner it can be shown easily that any sequence can be improved through a series of subsequence reversals until one of the sequences in the theorem is obtained. □

Clearly, $1, 3, 5, \ldots, n, \ldots, 6, 4, 2$ is a SEPT-LEPT sequence. That such a sequence is optimal should have been expected. Short jobs should be scheduled in the beginning of the sequence just to make sure that machine 2 does not remain idle for too long, while short jobs should also be scheduled towards the end of the sequence in order to avoid machine 2 being busy for a long time after machine 1 has completed all its processing. Note that optimal sequences are slightly different when n is even or odd. If n is even the optimal sequence is $1, 3, 5, \ldots, n-1, n, n-2, \ldots, 6, 4, 2$, and if n is odd the optimal sequence is $1, 3, 5, \ldots, n-2, n, n-1, \ldots, 6, 4, 2$.

Theorem 13.2.2 thus gives an indication of the impact of the means of the processing times on the optimal sequence. Generalizing the result of Theorem 13.2.2 to more than two machines is impossible. Counterexamples can be found.

Consider now the same model with $F_{1j} = F_{2j} = F_j$, $j = 1, \ldots, n$, but with the means of the distributions $F_1, F_2, \ldots, F_n$ being identical and equal to 1. However, the variances of the distributions are now different. Assume that the distributions have symmetric probability density functions and that $F_1 \geq_{sv} F_2 \geq_{sv} \cdots \geq_{sv} F_n$. This implies that all random variables lie between 0 and 2. For the two machine model with no intermediate buffers the following theorem holds.

Theorem 13.2.3. *If $F_1 \geq_{sv} F_2 \geq_{sv} \cdots \geq_{sv} F_n$ then the sequences $1, 3, 5, \ldots, n, \ldots, 6, 4, 2$ and $2, 4, 6, \ldots, n, \ldots, 5, 3, 1$ minimize the expected makespan in the class of nonpreemptive static list policies.*

Proof. First it is shown that any sequence can be transformed into a better sequence (with a smaller expected makespan) of the form $1, j_1, \ldots, j_{n-2}, 2$, where $j_1, \ldots, j_{n-2}$ is a permutation of jobs $3, \ldots, n$. Compare sequence

$$j_1, \ldots, j_k, 1,\ \ j_{k+1}, \ldots, j_l, 2, j_{l+1}, \ldots, j_{n-2}$$

with sequence

$$1, j_k, \ldots, j_1,\ \ j_{k+1}, \ldots, j_l, 2, j_{l+1}, \ldots, j_{n-2}.$$

Subtracting the makespan of the second sequence from that of the first yields:

$$\begin{aligned}
&E(\max(X_1, X_{j_{k+1}})) - E(\max(X_{j_1}, X_{j_{k+1}})) \\
&\qquad = E(\min(X_{j_1}, X_{j_{k+1}})) - E(\min(X_1, X_{j_{k+1}})) \\
&\qquad = \int_0^2 \Big(\bar{F}_{j_1}(t)\bar{F}_{j_{k+1}}(t) - \bar{F}_1(t)\bar{F}_{j_{k+1}}(t)\Big)dt \\
&\qquad = \int_0^2 \Big(\bar{F}_{j_{k+1}}(t)\Big(\bar{F}_{j_1}(t) - \bar{F}_1(t)\Big)\Big)dt \\
&\qquad \geq 0
\end{aligned}$$

It is therefore better to schedule job 1 first. A similar argument shows that job 2 has to be scheduled last.

The next step is to show that any sequence can be transformed into a sequence of the form $1, 3, j_1, \ldots, j_{n-3}, 2$ with a smaller expected makespan. Compare sequence

$$1, \;\; j_1, \ldots, j_k, 3, \;\; j_{k+1}, \ldots, j_{n-3}, 2$$

with sequence

$$1, \;\; 3, j_k, \ldots, j_1, \;\; j_{k+1}, \ldots, j_{n-3}, 2.$$

The expected makespan of the first sequence minus the expected makespan of the second sequence is

$$\begin{aligned}
&E(\max(X_1, X_{j_1})) + E(\max(X_3, X_{j_{k+1}})) - E(\max(X_1, X_3)) - E(\max(X_{j_1}, X_{j_{k+1}})) \\
&= E(\min(X_1, X_3)) + E(\min(X_{j_1}, X_{j_{k+1}})) - E(\min(X_1, X_{j_1})) - E(\min(X_3, X_{j_{k+1}})) \\
&= \int_0^2 \Big(\bar{F}_1(t)\bar{F}_3(t) + \bar{F}_{j_1}(t)\bar{F}_{j_{k+1}}(t)\Big)dt - \int_0^2 \Big(\bar{F}_1(t)\bar{F}_{j_1}(t) + \bar{F}_3(t)\bar{F}_{j_{k+1}}(t)\Big)dt \\
&= \int_0^2 \Big(\bar{F}_{j_{k+1}}(t) - \bar{F}_1(t)\Big)\Big(\bar{F}_{j_1}(t) - \bar{F}_3(t)\Big)dt \\
&\geq 0.
\end{aligned}$$

So the optimal sequence has to be of the form $1, 3, j_1, \ldots, j_{n-3}, 2$. Proceeding in this manner the optimality of the two sequences stated in the theorem can be verified easily. □

This result basically states that the optimal sequence puts jobs with larger variances more towards the beginning and end of the sequence, and jobs with smaller variances more towards the middle of the sequence. Such a sequence could be referred to as an *LV-SV* sequence.

It is not clear whether similar results hold when there are more than two machines in series. Results for problems with more machines are extremely hard to come by, since the complexity of the problem increases considerably when going from two to three machines.

Nevertheless, some properties can be shown for m machines in series with blocking, i.e., $Fm \mid block \mid C_{\max}$. Assume that $F_{1j} = F_{2j} = \cdots = F_{mj} = F_j$ with mean $1/\lambda_j$ and that $X_{1j}, \ldots, X_{mj}$ are independent.

Theorem 13.2.4. *If $F_1 \leq_{a.s.} F_2 \leq_{a.s.} \cdots \leq_{a.s.} F_n$, then a sequence minimizes the expected makespan if and only if it is a SEPT-LEPT sequence.*

Proof. As the proof of this theorem is straightforward, only a short outline is given. The proof is similar to the proof of Theorem 6.2.4 and consists of two parts. In the first part it is shown that every SEPT-LEPT sequence attains the lower bound of Lemma 13.1.3 and in the second part it is shown that any sequence that is not SEPT-LEPT leads to a makespan that is strictly larger than the lower bound. □

13.3 Stochastic Job Shops

Consider now the two machine job shop with job j having a processing time on machine 1 that is exponentially distributed with rate λ_j and a processing time on machine 2 that is exponentially distributed with rate μ_j. Some of the jobs have to be processed first on machine 1 and then on machine 2, while the remaining jobs have to be processed first on machine 2 and then on machine 1. Let $J_{1,2}$ denote the first set of jobs and $J_{2,1}$ the second set of jobs. Minimizing the expected makespan turns out to be an easy extension of the two machine flow shop model with exponential processing times.

Theorem 13.3.1. *The following nonpreemptive policy minimizes the expected makespan in the classes of nonpreemptive dynamic policies and preemptive dynamic policies: when machine 1 is freed the decision-maker selects from $J_{1,2}$ the job with the highest $\lambda_j - \mu_j$; if all jobs from $J_{1,2}$ already have completed their processing on machine 1 the decision-maker may take any job from $J_{2,1}$ that already has completed its processing on machine 2. When machine 2 is freed the decision-maker selects from $J_{2,1}$ the job with the highest $\mu_j - \lambda_j$; if all jobs from $J_{2,1}$ already have completed their processing on machine 2 the decision-maker may take any job from $J_{1,2}$ that already has completed its processing on machine 1.*

Proof. The proof consists of two parts. First, it is shown that jobs from $J_{2,1}$ have a lower priority on machine 1 than jobs from $J_{1,2}$ and jobs from $J_{1,2}$ have a lower priority on machine 2 than jobs from $J_{2,1}$. After that, it is shown that jobs from $J_{1,2}$ are ordered on machine 1 in decreasing order of $\lambda_j - \mu_j$ and jobs from $J_{2,1}$ on machine 2 in decreasing order of $\mu_j - \lambda_j$.

In order to show the first part, condition on a realization of all $2n$ processing times. The argument is by contradiction. Suppose an optimal schedule puts at one point in time a job from $J_{2,1}$ on machine 1 rather than a job from $J_{1,2}$. Consider the last job from $J_{2,1}$ processed on machine 1 before a job from $J_{1,2}$. Perform the following change in the schedule: Take this job from $J_{2,1}$

and postpone its processing until the last job from $J_{1,2}$ has been completed. After this change all jobs from $J_{1,2}$ are completed earlier on machine 1 and are available earlier at machine 2. This implies that machine 1 will finish with all its processing at the same time as it did before the interchange. However, machine 2 may finish with all its processing earlier than before the interchange because now the jobs from $J_{1,2}$ are available earlier at machine 2. This completes the first part of the proof of the theorem.

In order to prove the second part proceed as follows. First, consider $J_{1,2}$. In order to show that the jobs from $J_{1,2}$ should be scheduled in decreasing order of $\lambda_j - \mu_j$, condition first on the processing times of all the jobs in $J_{2,1}$ on both machines. The jobs from $J_{2,1}$ have a higher priority on machine 2 and a lower priority on machine 1. Assume that two adjacent jobs from $J_{1,2}$ are not scheduled in decreasing order of $\lambda_j - \mu_j$. Performing a pairwise interchange in the same way as in Theorem 13.1.1 results in a smaller expected makespan. This shows that the jobs from $J_{1,2}$ have to be scheduled on machine 1 in decreasing order of $\lambda_j - \mu_j$. A similar argument shows that the jobs from $J_{2,1}$ have to be scheduled on machine 2 in decreasing order of $\mu_j - \lambda_j$. □

The result described in Theorem 13.3.1 is similar to the result described in Section 7.1 with regard to $J2 \mid\mid C_{\max}$. In deterministic scheduling the research on the more general $Jm \mid\mid C_{\max}$ problem has focused on heuristics and enumerative procedures. Stochastic job shops with more than two machines have not received as much attention in the literature.

13.4 Stochastic Open Shops

Consider a two machine open shop where the processing time of job j on machine 1 is the random variable X_{1j}, distributed according to F_{1j}, and on machine 2 the random variable X_{2j}, distributed according to F_{2j}. The objective is to minimize the expected makespan. As before, the exponential distribution is considered first. In this case, however, it is not known what the optimal policy is when F_{1j} is exponential with rate λ_j and F_{2j} exponential with rate μ_j. It appears that the optimal policy may not have a simple structure and may even depend on the values of the λ's and μ's. However, the special case with $\lambda_j = \mu_j$ can be analyzed. In contrast to the results obtained for the stochastic flow shops the optimal policy now cannot be regarded as a permutation sequence, but rather as a policy that prescribes a given action dependent upon the state of the system.

Theorem 13.4.1. *The following policy minimizes the expected makespan in the class of preemptive dynamic policies as well as in the class of nonpreemptive dynamic policies: whenever a machine is freed, the scheduler selects from the jobs that have not yet undergone processing on either one of the two machines, the job with the longest expected processing time. If there are no such*

jobs remaining the decision-maker may take any job that only needs processing on the machine just freed. Preemptions do not occur.

Proof. Just as in the deterministic case the two machines are continuously busy with the possible exception of at most a single idle period on at most one machine. The idle period can be either an idle period of *Type I* or an idle period of *Type II* (see Figure 8.1). In the case of no idle period at all or an idle period of Type II the makespan is equal to the maximum of the workloads on the two machines, i.e.,

$$C_{\max} = \max\Big(\sum_{j=1}^{n} X_{1j}, \sum_{j=1}^{n} X_{2j}\Big).$$

In the case of an idle period of Type I, the makespan is strictly larger than the R.H.S. of the expression above. Actually, in this case the makespan is

$$C_{\max} = \max\Big(\sum_{j=1}^{n} X_{1j}, \sum_{j=1}^{n} X_{2j}\Big) + \min(I_1, I_2),$$

where I_1 is the length of the idle period and I_2 is the makespan minus the workload on the machine that did not experience an idle period. It is clear that the first term of the R.H.S. of the expression above does not depend on the policy used. In order to prove the theorem it suffices to show that the described policy minimizes the expected value of the second term on the R.H.S., i.e., $E(\min(I_1, I_2))$. This term clearly depends on the policy used.

In order to obtain some more insight in this second term, consider the following: Suppose job j is the job causing the idle period, that is, job j is the last job to be completed. Given that job j causes an idle period of Type I, it follows from Exercise 9.13 that

$$E(\min(I_1, I_2)) = \frac{1}{2\lambda_j}$$

If q'_j denotes the probability of job j causing an idle period of Type II under policy π', then

$$E(C_{\max}(\pi')) = E\Big(\max\Big(\sum_{j=1}^{n} X_{1j}, \sum_{j=1}^{n} X_{2j}\Big)\Big) + E(H'),$$

where

$$E(H') = \sum_{j=1}^{n} q'_j \frac{1}{2\lambda_j}.$$

From the theory of dynamic programming (see Appendix B), it follows that in order to prove optimality of the policy stated in the theorem, say policy π^*,

it suffices to show that using π^* from any time t onwards results in a smaller expected makespan than acting differently at time t and using π^* from the *next* decision moment onwards. Two types of actions at time t would violate π^*. First, it is possible to start a job that is not the longest job among the jobs not processed yet on either machine; second, it is possible to start a job that already has been processed on the other machine while there are still jobs in the system that have not yet received processing on either machine.

In the remaining part of the proof, the following notation is used: set J_1 represents the set of jobs which, at time t, have not yet completed their first processing, while set J_2 represents the set of jobs which at time t have not yet started with their second processing. Clearly, set J_2 includes set J_1, $J_1 \subset J_2$.

Case 1. Let π' denote the policy that, at time t, puts job k on the machine freed, with $k \in J_1$ and k not being the largest job in J_1, and that reverts back to π^* from the next decision moment onwards. Let job 0 be the job that is being processed, at time t, on the busy machine. Let r'_j (r^*_j) denote the probability that job j is the last job to complete its *first* processing under policy π' (π^*) and therefore be a candidate to cause an idle period. Suppose this job j is processed on machine 1. For job j to cause an idle period of Type I it has to outlast all those jobs that still have to receive their second processing on machine 2, and then after job j completes its processing on machine 1 and starts its processing on machine 2, it has to outlast all those jobs that still have to receive their second processing on machine 1. So

$$q^*_j = r^*_j \prod_{l \in \{J_2 - j\}} \frac{\lambda_l}{\lambda_l + \lambda_j}$$

and

$$q'_j = r'_j \prod_{l \in \{J_2 - j\}} \frac{\lambda_l}{\lambda_l + \lambda_j}.$$

Also

$$q'_0 = q^*_0.$$

Note that the expressions for q^*_j and q'_j indicate that q^*_j and q'_j do not depend on the machine on which job l, $l \in \{J_2 - j\}$, receives its second processing: processing job l the second time on the same machine where it was processed the first time, results in values for q^*_j and q'_j that are the same as when job l is processed the second time on the machine it was not processed the first time. In order to show that $E(H^*) \leq E(H')$ it suffices to show that

$$\sum_{j \in \{J_1 - 0\}} \Big(r^*_j \Big(\prod_{l \in \{J_2 - j\}} \frac{\lambda_l}{\lambda_l + \lambda_j} \Big) \frac{1}{2\lambda_j} \Big) \leq \sum_{j \in \{J_1 - 0\}} \Big(r'_j \Big(\prod_{l \in \{J_2 - j\}} \frac{\lambda_l}{\lambda_l + \lambda_j} \Big) \frac{1}{2\lambda_j} \Big)$$

Note that if $\lambda_a \leq \lambda_b$, then

$$\Big(\prod_{l\in\{J_2-a\}}\frac{\lambda_l}{\lambda_l+\lambda_a}\Big)\frac{1}{2\lambda_a}\geq\Big(\prod_{l\in\{J_2-b\}}\frac{\lambda_l}{\lambda_l+\lambda_b}\Big)\frac{1}{2\lambda_b}$$

Suppose the sequence in which the jobs in J_1 start with their first processing under π' is $0,k,1,2,\ldots,k-1,k+1,\ldots$ where

$$\lambda_1\leq\lambda_2\leq\cdots\leq\lambda_{k-1}\leq\lambda_k\leq\lambda_{k+1}\leq\cdots$$

Performing a pairwise swap in this sequence results in $0,1,k,2,\ldots,k-1,k+1,\ldots$. Let this new sequence correspond to policy π''. Now Lemma 12.1.1 can be used whereby π' (π'') corresponds to sequence $X_0,X_2,X_1,\ldots,X_n$ ($X_0,X_1,X_2,\ldots,X_n$) and r'_j (r''_j) corresponds to the r_j (q_j) in Lemma 12.1.1. Using Lemma 12.1.1 and the inequalities above, it is established that $E(H'')\leq E(H')$. Proceeding in this manner whereby at each step a pairwise interchange is performed between job k and the job immediately following it, the sequence $1,2,\ldots,k-1,k,k+1,\ldots$ is obtained. At each step it is shown that the expected makespan decreases.

Case 2. Let π' in this case denote the policy that instructs the scheduler at time t to start job l with rate λ_l, $l\in\{J_2-J_1\}$ and to adopt policy π^* from the next decision moment onwards. That is, job l starts at time t with its second processing while there are still jobs in J_1 that have not completed their first processing yet. Let r'_j, $j\in J_1$, in this case denote the probability that job j under π' completes its first processing after all jobs in J_1 have completed their first processing and after job l has completed its second processing. Let r'_l denote the probability that job l under π' completes its second processing after all jobs in J_1 have completed their first processing. Assume that when using π^* from t onwards the scheduler may, after having started all jobs in J_1, choose job l as the first job to undergo its second processing and may do this on the machine that becomes available first (under Case 1 it became clear that the probability of job j, $j\neq l$, causing a Type I idle period does not depend on the machine on which job l is processed the second time). Let r^*_j now denote the probability that job j completes its first processing after jobs J_1-j complete their first processing and after job l completes its second processing. Let r^*_l denote the probability that job l completes its second processing after all jobs in J_1 have completed their first processing. So

$$q^*_j=r^*_j\prod_{i\in\{J_2-j-l\}}\frac{\lambda_i}{\lambda_i+\lambda_j}$$

for all j in J_1 and

$$q'_j=r'_j\prod_{i\in\{J_2-j-l\}}\frac{\lambda_i}{\lambda_i+\lambda_j}$$

for all j in J_1. Again

$$q'_0=q^*_0.$$

In order to show that $E(H^*) \leq E(H')$ it suffices to show that

$$\sum_{j\in\{J_1-0\}} \Big(q_j^* \Big(\prod_{i\in\{J_2-j-l\}} \frac{\lambda_i}{\lambda_i+\lambda_j}\Big)\frac{1}{2\lambda_j}\Big) \leq \sum_{j\in\{J_1-0\}} \Big(q_j' \Big(\prod_{i\in\{J_2-j-l\}} \frac{\lambda_i}{\lambda_i+\lambda_j}\Big)\frac{1}{2\lambda_j}\Big).$$

From Lemma 12.1.1 it follows that $r_l^* \geq r_l'$ and $r_i^* \leq r_i'$, $i \in J_1$. It then follows that $E(H^*) \leq E(H')$. This completes the proof of the theorem. □

It appears to be very hard to generalize this result to include a larger class of distributions.

Example 13.4.2 (Open Shop with Processing Times that are Mixtures of Exponentials)

Let the processing time of job j on machine i, $i = 1, 2$, be a mixture of an exponential with rate λ_j and zero with arbitrary mixing probabilities. The optimal policy is to process at time 0 all jobs for a very short period of time on both machines just to check whether their processing times on the two machines are zero or positive. After the nature of all the processing times have been determined, the problem is reduced to the scenario covered by Theorem 13.4.1. ||

Theorem 13.4.1 states that jobs that still have to undergo processing on both machines have priority over jobs that only need processing on one machine. In a sense, the policy described in Theorem 13.4.1 is similar to the LAPT rule introduced in Section 8.1 for the deterministic $O2 \,||\, C_{\max}$ problem.

From Theorem 13.4.1 it follows that the problem is tractable also if the processing time of job j on machine 1 as well as on machine 2 is exponentially distributed with rate 1. The policy that minimizes the expected makespan always gives priority to jobs that have not yet undergone processing on either machine. This particular rule does not require any preemptions. In the literature, this rule has been referred to in this scenario as the *Longest Expected Remaining Processing Time first (LERPT)* rule.

Actually, if in the two-machine case all processing times are exponential with mean 1 and if preemptions are allowed, then the total expected completion time can also be analyzed. This model is an exponential counterpart of $O2 \mid p_{ij} = 1, prmp \mid \sum C_j$. The total expected completion time clearly requires a different policy. One particular policy in the class of preemptive dynamic policies is appealing: consider the policy that prescribes the scheduler to process, whenever possible, on each one of the machines a job that already has been processed on the other machine. This policy may require the scheduler at times to interrupt the processing of a job and start with the processing of a job that just has completed its operation on the other machine. In what follows this policy is referred to as the Shortest Expected Remaining Processing Time first (SERPT) policy.

Theorem 13.4.3. *The preemptive SERPT policy minimizes the total expected completion time in a two machine open shop in the class of preemptive dynamic policies.*

Proof. Let A_{ij}, $i = 1, 2$, $j = 1, \ldots, n$, denote the time that j jobs have completed their processing requirements on machine i. An idle period on machine 2 occurs if and only if

$$A_{1,n-1} \leq A_{2,n-1} \leq A_{1,n}$$

and an idle period on machine 1 occurs if and only if

$$A_{2,n-1} \leq A_{1,n-1} \leq A_{2,n}.$$

Let $j_1, j_2, \ldots, j_n$ denote the sequence in which the jobs leave the system, i.e., job j_1 is the first one to complete both operations, job j_2 the second, and so on. Under the SERPT policy

$$C_{j_k} = \max(A_{1,k}, A_{2,k}) = \max\Big(\sum_{l=1}^{k} X_{1l}, \sum_{l=1}^{k} X_{2l}\Big), \qquad k = 1, \ldots, n-1$$

This implies that the time epoch of the kth job completion, $k = 1, \ldots, n-1$, is a random variable that is the maximum of two independent random variables, both with Erlang(k, λ) distributions. The distribution of the last job completion, the makespan, is different. It is clear that under the preemptive SERPT policy the sum of the expected completion times of the first $n-1$ jobs that leave the system are minimized. It is not immediately obvious that SERPT minimizes the sum of all n completion times. Let

$$B = \max(A_{1,n-1}, A_{2,n-1}).$$

The random variable B is independent of the policy. At time B, each machine has at most one more job to complete. A distinction can now be made between two cases.

First, consider the case where, at B, a job remains to be completed on only one of the two machines. In this case, neither the probability of this event occurring nor the waiting cost incurred by the last job that leaves the system (at $\max(A_{1,n}, A_{2,n})$) depends on the policy. Since SERPT minimizes the expected sum of completion times of the first $n-1$ jobs to leave the system, it follows that SERPT minimizes the expected sum of the completion times of all n jobs.

Second, consider the case where, at time B, a job still remains to be processed on both machines. Either (i) there is one job left that needs processing on both machines or (ii) there are two jobs left, each requiring processing on one machine (a different machine for each). Under (i) the expected sum of the completion times of the last two jobs that leave the system is $E(B) + E(B+2)$, while under (ii) it is $E(B) + 1 + E(B) + 1$. In both subcases the expected sum of the completion times of the last two jobs is the same. As SERPT minimizes

the expected sum of the completion times of the first $n-2$ jobs that leave the system, it follows that SERPT minimizes the expected sum of the completion times of all n jobs. □

Unfortunately, no results have appeared in the literature concerning stochastic open shops with more than 2 machines.

13.5 Discussion

Among the models discussed in this chapter, the stochastic flow shops tend to be the easiest. Stochastic job shops and stochastic open shops tend to be considerably harder.

The stochastic flow shops that are the most tractable are usually counterparts of deterministic permutation flow shops and deterministic proportionate flow shops. The natural relationship between stochastic flow shops and tandem queues may also yield additional structural insights into stochastic flow shops (based on known results in queuing theory). One direction that may also lead to new results in the future lies in the realm of asymptotic analyses. For example, what happens with an objective function, e.g., $\sum_{j=1}^{n} E(C_j)/n$, when the number of jobs (or the number of machines) goes to ∞?

Exercises (Computational)

13.1. Consider a two machine flow shop with unlimited intermediate storage and three jobs. Each job has an exponentially distributed processing time with mean 1 on both machines (the job sequence is therefore immaterial). The six processing times are i.i.d. Compute the expected makespan and the total expected completion time.

13.2. Consider an m machine permutation flow shop without blocking. The processing times of job j on the m machines are identical and equal to the random variable X_j from an exponential distribution with mean 1. The random variables $X_1, \ldots, X_n$ are i.i.d. Determine the expected makespan and the total expected completion time as a function of m and n.

13.3. Compare the expected makespan obtained in Exercise 13.1 with the expected makespan obtained in Exercise 13.2 for $m = 2$ and $n = 3$. Determine which one is larger and give an explanation.

13.4. Consider a two machine flow shop with zero intermediate storage and blocking and n jobs. The processing time of job j on machine i is X_{ij}, exponentially distributed with mean 1. The $2n$ processing times are i.i.d. Compute the expected makespan and the total expected completion time as a function of n.

13.5. Consider a two machine flow shop with zero intermediate storage and blocking and n jobs. The processing time of job j on each one of the two machines is equal to the random variable X_j from an exponential distribution with mean 1. The variables $X_1, \ldots, X_n$ are i.i.d. Compute again the expected makespan and the total expected completion time. Compare the results with the results obtained in Exercise 13.4.

13.6. Consider the two machine open shop and n jobs. The processing times X_{ij} are all i.i.d. exponential with mean 1. Assume that the policy in Theorem 13.3.1 is followed, i.e., jobs that still need processing on both machines have priority over jobs that only need processing on one of the machines.

(a) Show that

$$E\Big(\max\big(\sum_{j=1}^{n} X_{1j}, \sum_{j=1}^{n} X_{2j}\big)\Big) = 2n - \sum_{k=n}^{2n-1} k \binom{k-1}{n-1} \Big(\frac{1}{2}\Big)^k$$

(b) Show that the probability of the jth job that starts its first processing causing an idle period of Type I is

$$\Big(\frac{1}{2}\Big)^{n-1-(j-2-i)} \Big(\frac{1}{2}\Big)^{n-1-i} = \Big(\frac{1}{2}\Big)^{2n-j}.$$

(c) Show that the expected makespan is equal to

$$E(C_{\max}) = 2n - \sum_{k=n}^{2n-1} k \binom{k-1}{n-1} \Big(\frac{1}{2}\Big)^k + \Big(\frac{1}{2}\Big)^n.$$

13.7. Consider the same scenario as in Exercise 13.1 with two machines and three jobs. However, now all processing times are i.i.d. according to the EME distribution with mean 1. Compute the expected makespan and the total expected completion time and compare the outcome with the results of Exercise 13.1.

13.8. Consider the same scenario as in Exercise 13.2 with m machines and n jobs. However, now $X_1, \ldots, X_n$ are i.i.d. according to the EME distribution with mean 1. Compute the expected makespan and the total expected completion time as a function of m and n. Compare the results obtained with the results from Exercises 13.2 and 13.7.

13.9. Consider a two machine job shop and three jobs. Jobs 1 and 2 have to be processed first on machine 1 and then on machine 2. Job 3 has to be processed first on machine 2 and then on machine 1. Compute the expected makespan under the assumption that the optimal policy is being followed.

13.10. Consider the following proportionate two machine open shop. The processing time of job j on the two machines is equal to the *same* random variable X_j that is exponentially distributed with mean 1. Assume that the two

machine open shop is being operated under the nonpreemptive dynamic policy that always gives priority to a job that has not yet received processing on the other machine. Compute the expected makespan with two jobs and with three jobs.

Exercises (Theory)

13.11. Consider the stochastic counterpart of $F2 \mid block \mid C_{\max}$ that is equivalent to a deterministic TSP with a distance matrix that has to be *minimized*. Verify whether this distance matrix satisfies the triangle inequality (i.e., $d_{jk} + d_{kl} \geq d_{jl}$ for all j, k and l).

13.12. Consider an m machine flow shop with zero intermediate storages between machines and n jobs. The processing times of $n-1$ jobs on each one of the m machines are 1. Job n has processing time X_{in} on machine i and the random variables $X_{1n}, X_{2n}, \ldots, X_{mn}$ are i.i.d. from distribution F_n with mean 1.

(a) Show that the sequence that minimizes the expected makespan puts the stochastic job either first or last.

(b) Show that the sequence that puts the stochastic job last minimizes the total expected completion time.

13.13. Consider the two machine flow shop with zero intermediate storage between the two machines. Of $n-2$ jobs the processing times are deterministic 1 on each one of the 2 machines. Of the two remaining jobs the four processing times are i.i.d. from an arbitrary distribution F with mean 1. Show that in order to minimize the expected makespan one of the stochastic jobs has to go first and the other one last.

13.14. Consider a stochastic counterpart of $Fm \mid p_{ij} = p_j \mid C_{\max}$. The processing time of job j on each one of the m machines is X_j from distribution F with mean 1.

(a) Find an upper and a lower bound for the expected makespan when F is ICR.

(b) Find an upper and a lower bound for the expected makespan when F is DCR.

13.15. Consider a stochastic counterpart of $F2 \mid block \mid C_{\max}$. The processing time of job j on machine i is X_{ij} from distribution F with mean 1. The $2n$ processing times are independent.

(a) Find an upper and a lower bound for the expected makespan when F is ICR.

(b) Find an upper and a lower bound for the expected makespan when F is DCR.

13.16. Consider a two machine open shop with n jobs. The processing times of job j on machines 1 and 2 are equal to X_j from distribution F. The random variables $X_1, \ldots, X_n$ are i.i.d. Show that in order to minimize the expected makespan the scheduler, whenever a machine is freed, has to select a job that has not yet been processed on the other machine.

13.17. Consider an m machine permutation flow shop with finite intermediate storages and blocking. The processing times of job j on the m machines are $X_{1j}, X_{2j}, \ldots, X_{mj}$ which are i.i.d. from distribution F_j. Assume that

$$F_1 \leq_{a.s.} F_2 \leq_{a.s.} \cdots \leq_{a.s.} F_n.$$

Show that SEPT minimizes the total expected completion time.

13.18. Consider stochastic counterparts of the following five deterministic problems:

(i) $F2 \mid block \mid C_{\max}$,
(ii) $F2 \mid\mid C_{\max}$,
(iii) $O2 \mid\mid C_{\max}$,
(iv) $J2 \mid\mid C_{\max}$,
(v) $P2 \mid chains \mid C_{\max}$

Problems (i), (ii), (iii) and (iv) all have n jobs. Problem (iv) has k jobs that have to be processed first on machine 1 and then on machine 2 and $n-k$ jobs that have to be processed first on machine 2 and then on machine 1. Problem (v) has $2n$ jobs in n chains of 2 jobs each. All processing times are i.i.d. exponential with mean 1. Compare the five problems with regard to the expected makespan and the total expected completion time under the optimal policy.

13.19. Consider a two machine proportionate flow shop with n jobs. If $X_{1j} = X_{2j} = D_j$ (deterministic) the makespan is sequence independent. If X_{1j} and X_{2j} are i.i.d. exponential with rate λ_j, then the expected makespan is sequence independent as well. Consider now a proportionate flow shop with X_{1j} and X_{2j} i.i.d. Erlang$(2, \lambda_j)$ with each one of the two phases distributed according to an exponential with rate λ_j, $j = 1, \ldots, n$. Show via an example that the expected makespan *does* depend on the sequence.

13.20. Consider two machines in series with a buffer storage of size b in between the two machines. The processing times are proportionate. Show that the makespan is decreasing convex in the buffer size b.

13.21. Consider a stochastic counterpart of $Fm \mid\mid C_{\max}$ with machines that have different speeds, say $v_1, \ldots, v_m$. These speeds are fixed (deterministic). Job j requires an amount of work Y_j, distributed according to F_j, on each one of the m machines. The amount of time it takes machine i to process job j is equal to $X_{ij} = Y_j / v_i$. Show that if $v_1 \geq v_2 \geq \cdots \geq v_n$ ($v_1 \leq v_2 \leq \cdots \leq v_n$) and $F_1 \leq_{lr} F_2 \leq_{lr} \cdots \leq_{lr} F_n$, the SEPT (LEPT) sequence minimizes the

expected makespan. In other words, show that if the flow shop is decelerating (accelerating) the SEPT (LEPT) sequence is optimal.

Comments and References

The result stated in Theorem 13.1.1 has a very long history. The first publications with regard to this result are by Talwar (1967), Bagga (1970) and Cunningham and Dutta (1973). An excellent paper on this problem is by Ku and Niu (1986). The proof of Theorem 13.1.1, as presented here, is due to Weiss (1982). The SEPT-LEPT sequences for flow shops were introduced and analyzed by Pinedo (1982) and the SV sequences were studied by Pinedo and Wie (1986). For stochastic flow shops with due dates see Lee and Lin (1991). For models where the processing times of any given job on the m machines are equal to the *same* random variable, see Pinedo (1985). An analysis of the impact of randomness of processing times on the expected makespan and on the total expected completion time is presented in Pinedo and Weber (1984). Frostig and Adiri (1985) obtain results for flow shops with three machines. Boxma and Forst (1986) obtain results with regard to the expected weighted number of tardy jobs.

The material in the section on stochastic flow shops with blocking is based primarily on the paper by Pinedo (1982). For more results on stochastic flow shops with blocking, see Pinedo and Weber (1984), Foley and Suresh (1984a, 1984b, 1986), Suresh, Foley and Dickey (1985), and Kijima, Makimoto and Shirakawa (1990). Kalczynski and Kamburowski (2005) show that the stochastic counterpart of $F2 \mid block \mid C_{\max}$ with exponentially distributed processing times (which is equivalent to a deterministic Travelling Salesman Problem) cannot be solved in polynomial time. The analysis of the two machine job shop is from the paper by Pinedo (1981b).

The optimal policy for minimizing the expected makespan in two machine open shops is due to Pinedo and Ross (1982). The result with regard to the minimization of the total expected completion time in two machine open shops is from the paper by Pinedo (1984).

Part III

Scheduling in Practice

Chapter 14

General Purpose Procedures for Deterministic Scheduling

This chapter describes a number of general purpose procedures that are useful in dealing with scheduling problems in practice and that can be implemented with relative ease in industrial scheduling systems. All the techniques described are heuristics that do not guarantee an *optimal* solution; they instead aim at finding reasonably good solutions in a relatively short time. The heuristics tend to be fairly generic and can be adapted easily to a large variety of scheduling problems.

This chapter does not cover exact optimization techniques such as branch-and-bound or dynamic programming. Applications of such techniques tend to be more problem specific and are therefore discussed in detail in the coverage of specific problems in other chapters and in the appendices.

The first section gives a classification and overview of some of the more elementary priority or dispatching rules such as those described in previous chapters. The second section discusses a method of combining priority or dispatching rules. These composite dispatching rules are combinations of a number of elementary dispatching rules. The third, fourth and fifth sections deal with procedures that are based on local search. These techniques tend to be fairly generic and can be applied to a variety of scheduling problems with only minor customization. The third section discusses simulated annealing and tabu-search while the fourth section describes a more general local search procedure, namely genetic algorithms. The fifth section describes a framework that combines several heuristic approaches, including dispatching rules and local search. This

M.L. Pinedo, *Scheduling*, DOI: 10.1007/978-0-387-78935-4_14,

framework is referred to as Ant Colony Optimization (ACO). The last section discusses other ways in which to combine the different empirical techniques with one another in a single framework.

14.1 Dispatching Rules

Research in dispatching rules has been active for several decades and many different rules have been studied in the literature. These rules can be classified in various ways. For example, a distinction can be made between *static* and *dynamic* rules. Static rules are not time dependent. They are just a function of the job and/or of the machine data, for instance, WSPT. Dynamic rules are time dependent. One example of a dynamic rule is the *Minimum Slack (MS) first* rule that orders jobs according to $\max(d_j - p_j - t, 0)$, which is time dependent. This implies that at some point in time job j may have a higher priority than job k and at some later point in time jobs j and k may have the same priority.

A second way of classifying rules is according to the information they are based upon. A *local* rule uses only information pertaining to either the queue where the job is waiting or to the machine where the job is queued. Most of the rules introduced in the previous chapters can be used as local rules. A *global* rule may use information regarding other machines, such as the processing time of the job on the next machine on its route. An example of a global rule is the LAPT rule for the two machine open shop.

In the preceding chapters many different rules have come up. Of course, there are many more besides those discussed. A simple one, very often used in practice, is the *Service In Random Order (SIRO)* rule. Under this rule no attempt is made to optimize anything. Another rule often used is the *First Come First Served* rule, which is equivalent to the *Earliest Release Date first (ERD)* rule. This rule attempts to equalize the waiting times of the jobs, i.e., to minimize the variance of the waiting times. Some rules are only applicable under given conditions in certain machine environments. For example, consider a bank of parallel machines, each with its own queue. According to the *Shortest Queue (SQ) first* rule every newly released job is assigned to the machine with the shortest queue. This rule is clearly time dependent and therefore dynamic. Many global dynamic rules have been designed for job shops. According to the *Shortest Queue at the Next Operation (SQNO)* rule, every time a machine is freed the job with the shortest queue at the next machine on its route is selected for processing.

In Table 14.1 an overview of some of the better known dispatching rules is given. A number of these rules yield optimal schedules in some machine environments and are reasonable heuristics in others. All of these rules have variations that can be applied in more complicated settings.

	RULE	DATA	ENVIRONMENT	SECTION
1	SIRO	–	–	14.1
2	ERD	r_j	$1 \mid r_j \mid Var(\sum(C_j - r_j)/n)$	14.1
3	EDD	d_j	$1 \mid\mid L_{\max}$	3.2
4	MS	d_j	$1 \mid\mid L_{\max}$	14.1
5	SPT	p_j	$Pm \mid\mid \sum C_j$; $Fm \mid p_{ij} = p_j \mid \sum C_j$	5.3; 6.1
6	WSPT	w_j, p_j	$Pm \mid\mid \sum w_j C_j$	3.1; 5.3
7	LPT	p_j	$Pm \mid\mid C_{\max}$	5.1
8	SPT-LPT	p_j	$Fm \mid block, p_{ij} = p_j \mid C_{\max}$	6.2
9	CP	p_j, *prec*	$Pm \mid prec \mid C_{\max}$	5.1
10	LNS	p_j, *prec*	$Pm \mid prec \mid C_{\max}$	5.1
11	SST	s_{jk}	$1 \mid s_{jk} \mid C_{\max}$	4.4
12	LFJ	M_j	$Pm \mid M_j \mid C_{\max}$	5.1
13	LAPT	p_{ij}	$O2 \mid\mid C_{\max}$	8.1
14	SQ	–	$Pm \mid\mid \sum C_j$	14.1
15	SQNO	–	$Jm \mid\mid \gamma$	14.1

Dispatching rules are useful when one attempts to find a reasonably good schedule with regard to a single objective such as the makespan, the total completion time or the maximum lateness.

However, objectives in the real world are often more complicated. For example, a realistic objective may be a combination of several basic objectives and also a function of time or a function of the set of jobs waiting for processing. Sorting the jobs on the basis of one or two parameters may not yield acceptable schedules. More elaborate dispatching rules, that take into account several different parameters, can address more complicated objective functions. Some of these more elaborate rules are basically a combination of a number of the elementary dispatching rules listed above. These more elaborate rules are referred to as composite dispatching rules and are described in the next section.

14.2 Composite Dispatching Rules

To explain the structure and the construction of these composite dispatching rules, a general framework has to be introduced. Subsequently, two of the more widely used composite rules are described.

A composite dispatching rule is a ranking expression that combines a number of elementary dispatching rules. An elementary rule is a function of attributes of the jobs and/or the machines. An attribute may be any property associated with either a job or a machine, that may be either constant or time dependent.

Examples of job attributes are weight, processing time and due date; examples of machine attributes are speed, the number of jobs waiting for processing and the total amount of processing that is waiting in queue. The extent to which a given attribute affects the overall priority of a job is determined by the elementary rule that uses it as well as a scaling parameter. Each elementary rule in the composite dispatching rule has its own scaling parameter that is chosen to properly scale the contribution of the elementary rule to the total ranking expression. The scaling parameters are either fixed by the designer of the rule, or variable and a function of time or of the particular job set to be scheduled. If they depend on the particular job set to be scheduled, they require the computation of some job set statistics that characterize the particular scheduling instance at hand as accurately as possible (for example, whether the due dates in the particular instance are tight or not). These statistics, which are also called *factors*, usually do not depend on the schedule and can be computed easily from the given job and machine attributes.

The functions that map the statistics into the scaling parameters have to be determined by the designer of the rule. Experience may offer a reasonable guide, but extensive computer simulation may also be required. These functions are usually determined only once, before the rule is made available for regular use.

Each time the composite dispatching rule is used for generating a schedule, the necessary statistics are computed. Based on the values of these statistics, the values of the scaling parameters are set by the predetermined functions. After the scaling parameters have been fixed the dispatching rule is applied to the job set.

One example of a composite dispatching rule is a rule that is often used for the $1 \mid\mid \sum w_j T_j$ problem. As stated in Chapter 3, the $1 \mid\mid \sum w_j T_j$ problem is strongly NP-hard. As branch-and-bound methods are prohibitively time consuming even for only 30 jobs, it is important to have a heuristic that provides a reasonably good schedule with a reasonable computational effort. Some heuristics come immediately to mind; namely, the WSPT rule (that is optimal when all release dates and due dates are zero) and the EDD rule or the MS rule (which are optimal when all due dates are sufficiently loose and spread out). It is natural to seek a heuristic or priority rule that combines the characteristics of these dispatching rules. The *Apparent Tardiness Cost (ATC)* heuristic is a composite dispatching rule that combines the WSPT rule and the MS rule. (Recall that under the MS rule the slack of job j at time t, i.e., $\max(d_j - p_j - t, 0)$, is computed and the job with the minimum slack is scheduled.) Under the ATC rule jobs are scheduled one at a time; that is, every time the machine becomes free a ranking index is computed for each remaining job. The job with the highest ranking index is then selected to be processed next. This ranking index is a function of the time t at which the machine became free as well as of the p_j, the w_j and the d_j of the remaining jobs. The index is defined as

$$I_j(t) = \frac{w_j}{p_j} \exp\Big(- \frac{\max(d_j - p_j - t, 0)}{K\bar{p}} \Big),$$

where K is the scaling parameter, that can be determined empirically, and $\bar{p}$ is the average of the processing times of the remaining jobs. If K is very large the ATC rule reduces to the WSPT rule. If K is very small the rule reduces to the MS rule when there are no overdue jobs and to the WSPT rule for the overdue jobs otherwise.

In order to obtain good schedules, the value of K (sometimes referred to as the look-ahead parameter) must be appropriate for the particular instance of the problem. This can be done by first performing a statistical analysis of the particular scheduling instance under consideration. There are several statistics that can be used to help characterize scheduling instances. The *due date tightness* factor τ is defined as

$$\tau = 1 - \frac{\sum d_j}{nC_{\max}},$$

where $\sum d_j/n$ is the average of the due dates. Values of τ close to 1 indicate that the due dates are tight and values close to 0 indicate that the due dates are loose. The due date range factor R is defined as

$$R = \frac{d_{\max} - d_{min}}{C_{\max}}.$$

A high value of R indicates a wide range of due dates, while a low value indicates a narrow range of due dates. A significant amount of research has been done to establish the relationships between the scaling parameter K and the factors τ and R.

Thus, when one wishes to minimize $\sum w_j T_j$ in a single machine or in a more complicated machine environment (machines in parallel, flexible flow shops), one first characterizes the particular problem instance through the two statistics. Then one determines the value of the look-ahead parameter K as a function of these characterizing factors and of the particular machine environment. After fixing K, one applies the rule. This rule could be used, for example, in the paper bag factory described in Example 1.1.1.

Several generalizations of the ATC rule have been developed in order to take release dates and sequence dependent setup times into account. Such a generalization, the *Apparent Tardiness Cost with Setups (ATCS)* rule, is designed for the $1 \mid s_{jk} \mid \sum w_j T_j$ problem. The objective is once again to minimize the total weighted tardiness, but now the jobs are subject to sequence dependent setup times. This implies that the priority of any job j depends on the job just completed on the machine freed. The ATCS rule combines the WSPT rule, the MS rule and the SST rule in a single ranking index. The rule calculates the index of job j at time t when job l has completed its processing on the machine as

$$I_j(t,l) = \frac{w_j}{p_j} \exp\Big(- \frac{\max(d_j - p_j - t, 0)}{K_1 \bar{p}} \Big) \exp\Big(- \frac{s_{lj}}{K_2 \bar{s}} \Big),$$

where $\bar{s}$ is the average of the setup times of the jobs remaining to be scheduled, K_1 the due date related scaling parameter and K_2 the setup time related scaling parameter. Note that the scaling parameters are dimensionless quantities to make them independent of the units used to express various quantities.

The two scaling parameters, K_1 and K_2, can be regarded as functions of three factors:

(i) the due date tightness factor τ,
(ii) the due date range factor R,
(iii) the setup time severity factor $\eta = \bar{s}/\bar{p}$.

These statistics are not as easy to determine as in the previous case. Even with a single machine the makespan is now schedule dependent because of the setup times. Before computing the τ and R factors the makespan has to be estimated. A simple estimate for the makespan on a single machine can be

$$\widehat{C}_{\max} = \sum_{j=1}^{n} p_j + n\bar{s}.$$

This estimate most likely will overestimate the makespan as the final schedule will take advantage of setup times that are shorter than average. The definitions of τ and R have to be modified by replacing the makespan with its estimate.

An experimental study of the ATCS rule, although inconclusive, has suggested some guidelines for selecting the two parameters K_1 and K_2 when $m = 1$. The following rule can be used for selecting a proper value of K_1:

$$\begin{aligned} K_1 &= 4.5 + R, && R \leq 0.5 \\ K_1 &= 6 - 2R, && R \geq 0.5. \end{aligned}$$

The following rule can be used for selecting a proper value of K_2:

$$K_2 = \tau/(2\sqrt{\eta}).$$

Example 14.2.1 (Application of the ATCS Rule)

Consider an instance of $1 \mid s_{jk} \mid \sum w_j T_j$ with the following four jobs.

jobs	1	2	3	4
p_j	13	9	13	10
d_j	12	37	21	22
w_j	2	4	2	5

The setup times s_{0j} of the first job in the sequence are presented in the table below.

jobs	1	2	3	4
s_{0j}	1	1	3	4

The sequence dependent setup times of the jobs following the first job are the following:

jobs	1	2	3	4
s_{1j}	–	4	1	3
s_{2j}	0	–	1	0
s_{3j}	1	2	–	3
s_{4j}	4	3	1	–

In order to use the ATCS rule the average processing time $\bar{p}$ and the average setup time $\bar{s}$ have to be determined. The average processing time is approximately 11 while the average setup time is approximately 2. An estimate for the makespan is

$$\widehat{C}_{\max} = \sum_{j=1}^{n} p_j + n\bar{s} = 45 + 4 \times 2 = 53.$$

The due date range factor $R = 25/53 \approx 0.47$, the due date tightness coefficient $\tau = 1 - 23/53 \approx 0.57$ and the setup time severity coefficient is $\eta = 2/11 \approx 0.18$. Using the given formulae, the parameter K_1 is chosen to be 5 and the parameter K_2 is chosen to be 0.7. In order to determine which job goes first $I_j(0,0)$ has to be computed for $j = 1, \ldots, 4$:

$$I_1(0,0) = \frac{2}{13} \exp\Big(-\frac{(12-13)^+}{55}\Big) \exp\Big(-\frac{1}{1.4}\Big) \approx 0.15 \times 1 \times 0.51 = 0.075$$

$$I_2(0,0) = \frac{4}{9} \exp\Big(-\frac{(37-9)^+}{55}\Big) \exp\Big(-\frac{1}{1.4}\Big) \approx 0.44 \times 0.6 \times 0.47 = 0.131$$

$$I_3(0,0) = \frac{2}{13} \exp\Big(-\frac{(21-13)^+}{55}\Big) \exp\Big(-\frac{3}{1.4}\Big) \approx 0.15 \times 0.86 \times 0.103 = 0.016$$

$$I_4(0,0) = \frac{5}{10} \exp\Big(-\frac{(22-10)^+}{55}\Big) \exp\Big(-\frac{4}{1.4}\Big) \approx 0.50 \times 0.80 \times 0.05 = 0.020.$$

Job 2 has the highest priority (in spite of the fact that its due date is the latest). As its setup time is 1, its completion time is 10. So at the second iteration $I_1(10,2)$, $I_3(10,2)$ and $I_4(10,2)$ have to be computed. To simplify the computations the values of $K_1\bar{p}$ and $K_2\bar{s}$ can be kept the same. Continuing the application of the ATCS rule results in the sequence $2, 4, 3, 1$ with the total weighted tardiness equal to 98. Complete enumeration shows that

this sequence is optimal. Note that this sequence always selects, whenever the machine is freed, one of the jobs with the smallest setup time. ||

The ATCS rule can be applied easily to $Pm \mid s_{jk} \mid \sum w_j T_j$ as well. Of course, the look-ahead parameters K_1 and K_2 have to be determined as a function of τ, R, η and m. The ATCS rule has been used in a scheduling system for a paper bag factory such as the one described in Example 1.1.1 (see also Section 16.4).

14.3 Local Search: Simulated Annealing and Tabu-Search

The heuristics described in the first two sections of this chapter are of the constructive type. They start without a schedule and gradually construct a schedule by adding one job at a time.

This section as well as the next section consider algorithms of the improvement type. Algorithms of the improvement type are conceptually completely different from algorithms of the constructive type. They start out with a complete schedule, which may be selected arbitrarily, and then try to obtain a better schedule by manipulating the current schedule. An important class of improvement type algorithms are the local search procedures. A local search procedure does not guarantee an optimal solution. It usually attempts to find a schedule that is better than the current one in the *neighbourhood* of the current one. Two schedules are *neighbors,* if one can be obtained through a well defined modification of the other. At each iteration, a local search procedure performs a search within the neighbourhood and evaluates the various neighbouring solutions. The procedure either accepts or rejects a candidate solution as the next schedule to move to, based on a given acceptance-rejection criterion.

One can compare the various local search procedures with respect to the following four design criteria:

(i) The schedule representation needed for the procedure.
(ii) The neighbourhood design.
(iii) The search process within the neighbourhood.
(iv) The acceptance-rejection criterion.

The representation of a schedule may be at times nontrivial. A nonpreemptive single machine schedule can be specified by a simple permutation of the n jobs. A nonpreemptive job shop schedule can be specified by m consecutive strings, each one representing a permutation of n operations on a specific machine. Based on this information, the starting and completion times of all operations can be computed. However, when preemptions are allowed the format of the schedule representation becomes significantly more complicated.

The design of the neighbourhood is a very important aspect of a local search procedure. For a single machine a neighbourhood of a particular schedule may be simply defined as all schedules that can be obtained by doing a single adjacent pairwise interchange. This implies that there are $n - 1$ schedules in the

neighbourhood of the original schedule. A larger neighbourhood of a single machine schedule may be defined by taking an arbitrary job in the schedule and inserting it in another position in the schedule. Clearly, each job can be inserted in $n-1$ other positions. The entire neighbourhood contains less than $n(n-1)$ neighbors as some of these neighbors are identical. The neighbourhood of a schedule in a more complicated machine environment is usually more complex.

An interesting example is a neighbourhood designed for the job shop problem with the makespan objective. In order to describe this neighbourhood, the concept of a critical path has to be used. A critical path in a job shop schedule consists of a set of operations of which the first one starts out at time $t = 0$ and the last one finishes at time $t = C_{\max}$. The completion time of each operation on a critical path is equal to the starting time of the next operation on that path; two successive operations either belong to the same job or are processed on the same machine (see Chapter 7). A schedule may have multiple critical paths that may overlap. Finding the critical path(s) in a given schedule for a job shop problem with the makespan objective is relatively straightforward. It is clear that in order to reduce the makespan, changes have to be made in the sequence(s) of the operations on the critical path(s). A simple neighbourhood of an existing schedule can be designed as follows: the set of schedules whose corresponding sequences of operations on the machines can be obtained by interchanging a pair of adjacent operations on the critical path of the current schedule. Note that in order to interchange a pair of operations on the critical path, the operations must be on the same machine and belong to different jobs. If there is a single critical path, then the number of neighbors within the neighbourhood is at most the number of operations on the critical path minus 1.

Experiments have shown that this type of neighbourhood for the job shop problem is too simple to be effective. The number of neighbouring schedules that are better than the existing schedule tends to be very limited. More sophisticated neighbourhoods have been designed that perform better. One of these is referred to as the *One Step Look-Back Adjacent Interchange*.

Example 14.3.1 (Neighbourhood of a job shop schedule)

A neighbor of a current schedule is obtained by first performing an adjacent pairwise interchange between two operations (i, j) and (i, k) on the critical path. After the interchange operation (i, k) is processed before operation (i, j) on machine i. Consider job k to which operation (i, k) belongs and refer to the operation of job k immediately preceding operation (i, k) as operation (h, k) (it is processed on machine h). On machine h, interchange operation (h, k) and the operation preceding (h, k) on machine h, say operation (h, l) (see Figure 14.1). From the figure it is clear that, even if the first interchange (between (i, j) and (i, k)) does not result in an improvement, the second interchange between (h, k) and (h, l) may lead to an overall improvement.

Actually, this design can be made more elaborate by backtracking more than one step. These types of interchanges are referred to as *Multi-Step Look-Back Interchanges*.

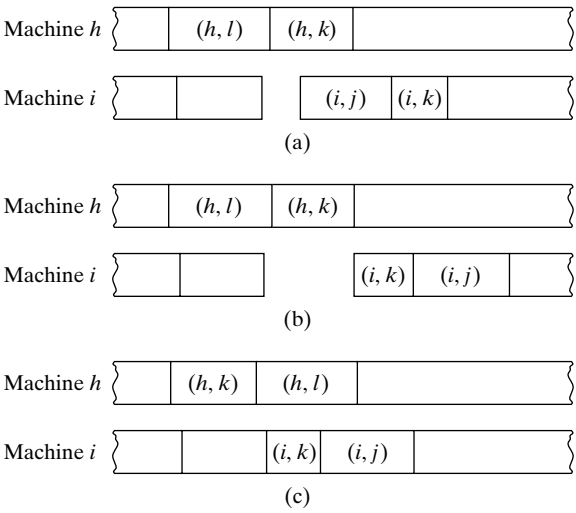


Fig. 14.1 One-step look-back interchange for $Jm \mid\mid C_{\max}$ (a) current schedule, (b) schedule after interchange of (i, j) and (i, k) (c) schedule after interchange of (h, l) and (h, k)

In exactly the same way, one can construct the *One Step Look-Ahead Interchange* and the *Multi-Step Look-Ahead Interchanges.* ||

The search process within a neighbourhood can be done in a number of ways. A simple way is to select schedules in the neighbourhood at random, evaluate these schedules and decide which one to accept. However, it may pay to do a more organized search and select first schedules that appear promising. One may want to consider swapping those jobs that affect the objective the most. For example, when the total weighted tardiness has to be minimized, one may want to move jobs that are very tardy more towards the beginning of the schedule.

The acceptance-rejection criterion is usually the design aspect that distinguishes a local search procedure the most. The difference between the two procedures discussed in the remaining part of this section, simulated annealing and tabu-search, lies mainly in their acceptance-rejection criteria. In simulated annealing the acceptance-rejection criterion is based on a probabilistic process while in tabu-search it is based on a deterministic process.

Simulated annealing is a search process that has its origin in the fields of material science and physics. It was first developed as a simulation model for describing the physical annealing process of condensed matter.

The simulated annealing procedure goes through a number of iterations. In iteration k of the procedure, there is a current schedule $\mathcal{S}_k$ as well as a best schedule found so far, $\mathcal{S}_0$. For a single machine problem these schedules are sequences (permutations) of the jobs. Let $G(\mathcal{S}_k)$ and $G(\mathcal{S}_0)$ denote the corresponding values of the objective function. Note that $G(\mathcal{S}_k) \geq G(\mathcal{S}_0)$. The value

of the best schedule obtained so far, $G(\mathcal{S}_0)$, is often referred to as the aspiration criterion. The algorithm, in its search for an optimal schedule, moves from one schedule to another. At iteration k, a search for a new schedule is conducted within the neighbourhood of $\mathcal{S}_k$. First, a so-called *candidate* schedule, say $\mathcal{S}_c$, is selected from the neighbourhood. This selection of a candidate schedule can be done at random or in an organized, possibly sequential, way. If $G(\mathcal{S}_c) < G(\mathcal{S}_k)$, a move is made, setting $\mathcal{S}_{k+1} = \mathcal{S}_c$. If $G(\mathcal{S}_c) < G(\mathcal{S}_0)$, then $\mathcal{S}_0$ is set equal to $\mathcal{S}_c$. However, if $G(\mathcal{S}_c) \geq G(\mathcal{S}_k)$, a move to $\mathcal{S}_c$ is made only with probability

$$P(\mathcal{S}_k, \mathcal{S}_c) = \exp\Big(\frac{G(\mathcal{S}_k) - G(\mathcal{S}_c)}{\beta_k}\Big);$$

with probability $1 - P(\mathcal{S}_k, \mathcal{S}_c)$ schedule $\mathcal{S}_c$ is rejected in favor of the current schedule, setting $\mathcal{S}_{k+1} = \mathcal{S}_k$. Schedule $\mathcal{S}_0$ does not change when it is better than schedule $\mathcal{S}_c$. The $\beta_1 \geq \beta_2 \geq \beta_3 \geq \cdots > 0$ are control parameters referred to as cooling parameters or temperatures (in analogy with the annealing process mentioned above). Often β_k is chosen to be a^k for some a between 0 and 1.

From the above description of the simulated annealing procedure it is clear that moves to worse solutions are allowed. The reason for allowing such moves is to give the procedure the opportunity to move away from a local minimum and find a better solution later on. Since β_k decreases with k, the acceptance probability for a non-improving move is lower in later iterations of the search process. The definition of the acceptance probability also ensures that if a neighbor is significantly worse, its acceptance probability is very low and a move is unlikely to be made.

Several stopping criteria are used for this procedure. One way is to let the procedure run for a prespecified number of iterations. Another is to let the procedure run until no improvement has been achieved during a predetermined number of iterations.

The method can be summarized as follows:

Algorithm 14.3.2 (Simulated Annealing)

Step 1.

Set $k = 1$ and select β_1.
Select an initial sequence $\mathcal{S}_1$ using some heuristic.
Set $\mathcal{S}_0 = \mathcal{S}_1$.

Step 2.

Select a candidate schedule $\mathcal{S}_c$ from the neighbourhood of $\mathcal{S}_k$.
If $G(\mathcal{S}_0) < G(\mathcal{S}_c) < G(\mathcal{S}_k)$, set $\mathcal{S}_{k+1} = \mathcal{S}_c$ and go to Step 3.
If $G(\mathcal{S}_c) < G(\mathcal{S}_0)$, set $\mathcal{S}_0 = \mathcal{S}_{k+1} = \mathcal{S}_c$ and go to Step 3.
If $G(\mathcal{S}_c) > G(\mathcal{S}_k)$ generate a random number U_k from a Uniform(0,1) distribution;
If $U_k \leq P(\mathcal{S}_k, \mathcal{S}_c)$ set $\mathcal{S}_{k+1} = \mathcal{S}_c$ otherwise set $\mathcal{S}_{k+1} = \mathcal{S}_k$ and go to Step 3.

Step 3.

Select $\beta_{k+1} \leq \beta_k$.
Increment k *by 1.*
If $k = N$ *then STOP, otherwise go to Step 2.* ||

The effectiveness of simulated annealing depends on the design of the neighbourhood as well as on how the search is conducted within this neighbourhood. If the neighbourhood is designed in a way that facilitates moves to better solutions and moves out of local minima, then the procedure will perform well. The search within a neighbourhood can be done randomly or in a more organized way. For example, the contribution of each job to the objective function can be computed and the job with the highest impact on the objective can be selected as a candidate for an interchange.

Over the last two decades simulated annealing has been applied to many scheduling problems, in academia as well as in industry, with considerable success.

The remainder of this section focuses on the tabu-search procedure. Tabu-search is in many ways similar to simulated annealing in that it also moves from one schedule to another with the next schedule being possibly worse than the one before. For each schedule, a neighbourhood is defined as in simulated annealing. The search within the neighbourhood for a potential candidate to move to is again a design issue. As in simulated annealing, this can be done randomly or in an organized way. The basic difference between tabu-search and simulated annealing lies in the mechanism that is used for approving a candidate schedule. In tabu-search the mechanism is not probabilistic but rather of a deterministic nature. At any stage of the process a tabu-list of mutations, which the procedure is *not* allowed to make, is kept. A mutation on the tabu-list can be, for example, a pair of jobs that may not be interchanged. The tabu-list has a fixed number of entries (usually between 5 and 9), that depends upon the application. Every time a move is made through a certain mutation in the current schedule, the *reverse* mutation is entered at the top of the tabu-list; all other entries in the tabu-list are pushed down one position and the bottom entry is deleted. The reverse mutation is put on the tabu-list to avoid returning to a local minimum that has been visited before. Actually, at times a reverse mutation that is tabu could actually have led to a new schedule, not visited before, that is better than any one generated so far. This may happen when the mutation is close to the bottom of the tabu-list and a number of moves have already been made since the mutation was entered in the list. Thus, if the number of entries in the tabu-list is too small cycling may occur; if it is too large the search may be unduly constrained. The method can be summarized as follows:

Algorithm 14.3.3 (Tabu-Search)

Step 1.

Set $k = 1$.

Select an initial sequence $\mathcal{S}_1$ using some heuristic.
Set $\mathcal{S}_0 = \mathcal{S}_1$.

Step 2.

Select a candidate schedule $\mathcal{S}_c$ from the neighbourhood of $\mathcal{S}_k$.
If the move $\mathcal{S}_k \rightarrow \mathcal{S}_c$ is prohibited by a mutation on the tabu-list,
then set $\mathcal{S}_{k+1} = \mathcal{S}_k$ and go to Step 3.
If the move $\mathcal{S}_k \rightarrow \mathcal{S}_c$ is not prohibited by any mutation on the tabu-list,
then set $\mathcal{S}_{k+1} = \mathcal{S}_c$ and
enter reverse mutation at the top of the tabu-list.
Push all other entries in the tabu-list one position down and
delete the entry at the bottom of the tabu-list.
If $G(\mathcal{S}_c) < G(\mathcal{S}_0)$, set $\mathcal{S}_0 = \mathcal{S}_c$;
Go to Step 3.

Step 3.

Increment k by 1.
If $k = N$ then STOP,
otherwise go to Step 2. ||

The following example illustrates the method.

Example 14.3.4 (Application of Tabu-Search)
Consider the following instance of $1 \,||\, \sum w_j T_j$.

jobs	1	2	3	4
p_j	10	10	13	4
d_j	4	2	1	12
w_j	14	12	1	12

The neighbourhood of a schedule is defined as all schedules that can be obtained through adjacent pairwise interchanges. The tabu-list is a list of pairs of jobs (j, k) that were swapped within the last two moves and cannot be swapped again. Initially, the tabu-list is empty.

Sequence $\mathcal{S}_1 = 2, 1, 4, 3$ is chosen as a first schedule. The value of the objective function is

$$\sum w_j T_j(2, 1, 4, 3) = 500.$$

The aspiration criterion is therefore 500. There are three schedules in the neighbourhood of $\mathcal{S}_1$, namely $1, 2, 4, 3$; $2, 4, 1, 3$ and $2, 1, 3, 4$. The respective values of the objective function are 480, 436 and 652. Selection of the best non-tabu sequence results in $\mathcal{S}_2 = 2, 4, 1, 3$. The aspiration criterion is changed to 436. The tabu-list is updated and contains now the pair (1,4). The values of the objective functions of the neighbors of $\mathcal{S}_2$ are:

sequence	4,2,1,3	2,1,4,3	2,4,3,1
$\sum w_j T_j$	460	500	608

Note that the second move is tabu. However, the first move is anyhow better than the second. The first move results in a schedule that is worse than the best one so far. The best one is the current one, which therefore is a local minimum. Nevertheless, $\mathcal{S}_3 = 4, 2, 1, 3$ and the tabu-list is updated and contains now $\{(2, 4), (1, 4)\}$. Neighbors of $\mathcal{S}_3$ with the corresponding values of the objective functions are:

sequence	2,4,1,3	4,1,2,3	4,2,3,1
$\sum w_j T_j$	436	440	632

Now, although the best move is to $2, 4, 1, 3$ ($\mathcal{S}_2$), this move is tabu. Therefore $\mathcal{S}_4$ is chosen to be $4, 1, 2, 3$. Updating the tabu-list results in $\{(1, 2), (2, 4)\}$ and the pair (1,4) drops from the tabu-list as the length of the list is kept to 2. Neighbors of $\mathcal{S}_4$ and their corresponding objective function values are:

sequence	1,4,2,3	4,2,1,3	4,1,3,2
$\sum w_j T_j$	408	460	586

The schedule $4, 2, 1, 3$ is tabu, but the best move is to the schedule $1, 4, 2, 3$. So $\mathcal{S}_5 = 1, 4, 2, 3$. The corresponding value of the objective is better than the aspiration criterion. So the aspiration criterion becomes 408. The tabu-list is updated by adding (1,4) and dropping (2,4). Actually, $\mathcal{S}_5$ is a global minimum, but tabu-search, being unaware of this, continues. ||

The information carried along in tabu-search consists of the tabu-list as well as the best solution obtained so far in the search process. Recently, more powerful versions of tabu-search have been proposed; these versions retain more information. One version uses a so-called tabu-tree. In this tree each node represents a solution or schedule. While the search process goes from one solution to another (with each solution having a tabu-list), the process generates additional nodes. Certain solutions that appear promising may not be used as a take-off point immediately but are retained for future use. If at a certain point during the search process the current solution does not appear promising as a take-off point, the search process can return within the tabu-tree to another node (solution) that had been retained before and take off again, but now in a different direction.

14.4 Local Search: Genetic Algorithms

Genetic algorithms are more general and abstract than simulated annealing and tabu-search. Simulated annealing and tabu-search may, in a certain way, be viewed as special cases of genetic algorithms.

Genetic algorithms, when applied to scheduling, view sequences or schedules as *individuals* or members of a *population.* Each individual is characterized by its *fitness.* The fitness of an individual is measured by the associated value of the objective function. The procedure works iteratively, and each iteration is referred to as a *generation.* The population of one generation consists of survivors from the previous generation plus the new schedules, i.e., the *offspring (children)* of the previous generation. The population size usually remains constant from one generation to the next. The offspring is generated through reproduction and mutation of individuals that were part of the previous generation (the *parents*). Individuals are sometimes also referred to as *chromosomes.* In a multi-machine environment a chromosome may consist of sub-chromosomes, each one containing the information regarding the job sequence on a machine. A mutation in a parent chromosome may be equivalent to an adjacent pairwise interchange in the corresponding sequence. In each generation the fittest individuals reproduce while the least fit die. The birth, death and reproduction processes that determine the composition of the next generation can be complex, and usually depend on the fitness levels of the individuals in the current generation.

A genetic algorithm, as a search process, differs in one important aspect from simulated annealing and tabu-search. At each iterative step a number of different schedules are generated and carried over to the next step. In simulated annealing and tabu-search only a single schedule is carried over from one iteration to the next. Hence simulated annealing and tabu-search may be regarded as special cases of genetic algorithms with a population size that is equal to 1. This diversification scheme is an important characteristic of genetic algorithms. In genetic algorithms the neighbourhood concept is also not based on a single schedule, but rather on multiple schedules. The design of the neighbourhood of the current population of schedules is therefore based on more general techniques than those used in simulated annealing and tabu-search. A new schedule can be generated by combining parts of different schedules from the current population. A mechanism that creates such a new schedule is often referred to as a *crossover* operator.

Example 14.4.1 (Linear Order Crossover (LOX) Operator)

One popular crossover operator is often referred to as the linear order crossover or LOX. This crossover operator, which creates a new member of the next generation from two members of the current generation, can be applied to single machine scheduling problems as well as to more complicated shop scheduling problems. It typically follows four steps, namely:

Step 1: Select at random a subsequence of jobs from one parent.

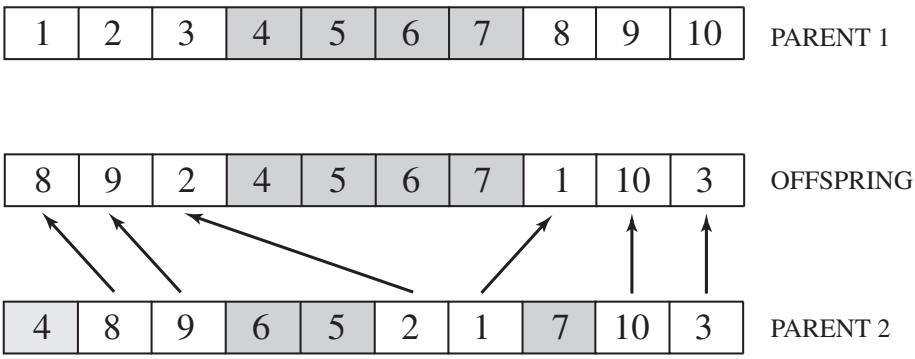


Fig. 14.2 Application of the Linear Crossover Operator

Step 2: Start generating a new offspring by copying the subsequence into the corresponding positions of the new offspring.

Step 3: Delete the jobs that are already in this subsequence from the second parent. The remaining subsequence in the second parent contains the jobs that the new offspring still needs.

Step 4: Place the jobs in this remaining subsequence in the unfilled positions of the new offspring from left to right in the order that they appeared in the second parent.

An example of such a crossover operation is depicted in Figure 14.2. ||

Of course, crossover operators have been applied to job shop schedules as well. In a job shop scheduling problem a new schedule can be generated by combining the sequence of operations on one machine in one parent's schedule with a sequence of operations on another machine in another parent's schedule.

A very simplified version of a genetic algorithm can now be described as follows.

Algorithm 14.4.2 (Genetic algorithm)

Step 1.

Set $k = 1$.
Select ℓ *initial sequences* $\mathcal{S}_{1,1}, \ldots, \mathcal{S}_{1,\ell}$ *using some heuristic.*

Step 2.

Select the two best schedules among $\mathcal{S}_{k,1}, \ldots, \mathcal{S}_{k,\ell}$
and call these $\mathcal{S}_k^+$ *and* $\mathcal{S}_k^{++}$.
Select the two worst schedules among $\mathcal{S}_{k,1}, \ldots, \mathcal{S}_{k,\ell}$
and call these $\mathcal{S}_k^-$ *and* $\mathcal{S}_k^{--}$.
Generate two offspring $\mathcal{S}^*$ *and* $\mathcal{S}^{**}$ *from parents* $\mathcal{S}_k^+$ *and* $\mathcal{S}_k^{++}$.
Replace $\mathcal{S}_k^-$ *and* $\mathcal{S}_k^{--}$ *with* $\mathcal{S}^*$ *and* $\mathcal{S}^{**}$.
Keep all other schedules the same and go to Step 3.

Step 3.

Increment k by 1.
If $k = N$ then STOP,
otherwise go to Step 2. ||

The use of genetic algorithms has its advantages and disadvantages. One advantage is that they can be applied to a problem without having to know much about the structural properties of the problem. They can be very easily coded and they often give fairly good solutions. However, the computation time needed to obtain a good solution may be somewhat long in comparison with the more rigorous problem specific approaches.

14.5 Ant Colony Optimization

Ant Colony Optimization (ACO) algorithms combine local search techniques, dispatching rules, and other techniques within one framework. The ACO paradigm is inspired by the trail following behavior of ant colonies. Ants, when moving along a path to a destination, leave along their path a chemical called pheromone as a signal for other ants to follow. An ACO algorithm assumes that a colony of (artificial) ants iteratively construct solutions for the problem at hand using (artificial) pheromone trails that are related to previously found solutions as well as to heuristic information. The ants communicate with one another only indirectly through changes in the amounts of pheromone they deposit on their trails during the algorithm's execution. Because the solutions constructed by the ants may not be locally optimal, many ACO algorithms allow the ants to improve their solutions through a local search procedure. The basic ACO framework consists of four steps.

Algorithm 14.5.1 (Ant Colony Optimization)

Step 1. (Initialization)

Set parameters and initialize pheromone trails.

Step 2. (Generate Solutions)

Generate ℓ solutions using a combination of pheromone trails and dispatching rules.

Step 3. (Improve Solutions)

Apply a local search procedure to each one of the ℓ solutions.

Step 4. (Update Settings)

If the best of the ℓ solutions produced in Step 3 is better than the best solution generated so far,
replace best solution obtained so far with the current solution and store the corresponding value of the objective function.
Update pheromone trail values and return to Step 2 to start next iteration.
||

An application of an ACO algorithm to $1 \mid\mid \sum w_j T_j$ can be described as follows. When an ant constructs a sequence in iteration k, it starts out with an empty sequence and iteratively appends a job from the set of jobs remaining to be scheduled to the partial sequence generated so far. With probability Q, the not yet scheduled job j that maximizes $\phi_{ij}(k) \cdot \eta_{ij}^{\beta}$ is put in position i of the current sequence. Here, $\phi_{ij}(k)$ is the pheromone trail associated with the assignment of job j to position i, the k indicates the dependence of the pheromone trail on the iteration count, the η_{ij} is the heuristic desirability of putting job j in position i, and β is a parameter that determines the influence of the heuristic desirability. With probability $1-Q$, job j is selected randomly with probability

$$P_{ij} = \frac{\phi_{ij}(k) \cdot \eta_{ij}^{\beta}}{\sum_{l \in J} \phi_{il}(k) \cdot \eta_{il}^{\beta}},$$

where J refers to the set of jobs that have not yet been scheduled. So, with probability Q the ant makes the best decision according to pheromone trails and heuristic desirability, and with probability $1-Q$ it makes a random selection with a certain bias. The heuristic desirability η_{ij} may be determined either via the EDD rule or via the ATC rule. If it is determined via the EDD rule, then $\eta_{ij} = 1/d_j$. If it is determined via the ATC rule, then $\eta_{ij} = 1/I_j(t)$, where

$$I_j(t) = \frac{w_j}{p_j} \exp\Big(- \frac{\max(d_j - p_j - t, 0)}{K\bar{p}} \Big).$$

The updating of the pheromone trails can be done in two ways, namely through immediate updating or through delayed updating. Immediate updating is a form of local updating that is applied every time an ant has added a new job to a partial sequence. If job j has been put into a position i, then the pheromone trail is modified as follows: an updated $\phi_{ij}(k)$ is obtained by multiplying the current $\phi_{ij}(k)$ with ξ and adding $(1-\xi)\phi_0$. Or, in a more algorithmic notation,

$$\phi_{ij}(k) \leftarrow \xi \phi_{ij}(k) + (1-\xi)\phi_0$$

The ξ, $0 < \xi \leq 1$, and the ϕ_0 (which is a small number) are the two parameters in this updating process. The effect of the immediate updating is to make the decision of putting job j in position i less desirable for the other ants encouraging thereby the exploration of other sequences within the same iteration k.

Delayed updating is a form of global updating that is done at the end of each iteration. At the end of iteration k the delayed updating procedure first evaporates, for each combination of i and j, some of the pheromone according to the formula

$$\phi_{ij}(k+1) = (1-\rho) \cdot \phi_{ij}(k),$$

where ρ, $0 < \rho \leq 1$, is a parameter that represents the pheromone evaporation rate. Then, also at the end of iteration k, it adds more pheromone to some (but

not all) combinations of i and j, i.e., only to those combinations that correspond to the best solution found in iteration k. If in the best solution generated in iteration k job j is put in position i, then

$$\phi_{ij}(k+1) = (1-\rho) \cdot \phi_{ij}(k) + \rho \cdot \Delta\phi_{ij}(k),$$

where

$$\Delta\phi_{ij}(k) = \frac{1}{\sum w_j T_j^*},$$

and $\sum w_j T_j^*$ is the total weighted tardiness of the overall best solution found so far.

Summarizing, an ACO algorithm combines several of the techniques described in this chapter within one framework. Dispatching rules as well as local search procedures play an important role within the framework. An ACO algorithm is, in a sense, also similar to a genetic algorithm since both types of procedures consider in each iteration multiple schedules ($\ell > 1$). However, the manner in which a population of schedules is generated in an iteration of a genetic algorithm is different from the manner in which a colony of ℓ ants generate their sequences in an iteration of an ACO algorithm.

14.6 Discussion

For many scheduling problems one can design all kinds of procedures that combine elements of the different techniques presented in this chapter.

For example, the following three phase approach has proven fairly useful for solving scheduling problems in practice. It combines composite dispatching rules with simulated annealing or tabu-search.

Phase 1: Values of a number of statistics are computed, such as the due date tightness, the setup time severity, and so on.

Phase 2: Based on the outcome of Phase 1 a number of scaling parameters for a composite dispatching rule are determined and the composite dispatching rule is applied on the scheduling instance.

Phase 3: The schedule developed in Phase 2 is used as an initial solution for a tabu-search or simulated annealing procedure that tries to generate a better schedule.

This three phase framework would only be useful if the routine would be used frequently (a new instance of the same problem has to be solved every day). The reason is that the empirical procedure that determines the functions that map values of the job statistics into appropriate values for scaling parameters constitutes a major investment of time. Such an investment pays off only when a routine is subject to heavy use.

Exercises (Computational)

14.1. Consider the instance in Example 14.2.1.

(a) How many different schedules are there?

(b) Compute the value of the objective in case, whenever the machine is freed, the job with the highest w_j/p_j ratio is selected to go next.

(c) Compute the value of the objective in case, whenever the machine is freed, the job with the minimum slack is selected to go next.

(d) Explain why in this instance under the optimal schedule the job with the latest due date has to go first.

14.2. Consider the instance in Example 14.2.1 and determine the schedule according to the ATCS rule with

(a) $K_1 = 5$ and $K_2 = \infty$;
(b) $K_1 = 5$ and $K_2 = 0.0001$;
(c) $K_1 = \infty$ and $K_2 = 0.7$.
(d) $K_1 = 0.0001$ and $K_2 = 0.7$.
(e) $K_1 = \infty$ and $K_2 = \infty$.

14.3. Consider the instance of $P2 \mid\mid \sum w_j T_j$ with the following 5 jobs.

jobs	1	2	3	4	5
p_j	13	9	13	10	8
d_j	6	18	10	11	13
w_j	2	4	2	5	4

(a) Apply the ATC heuristic on this instance with the look-ahead parameter $K = 1$.

(b) Apply the ATC heuristic on this instance with the look-ahead parameter $K = 5$.

14.4. Consider the instance in Example 14.3.4. Apply the tabu-search technique once more, starting out with the same initial sequence, under the following conditions.

(a) Make the length of the tabu-list 1, i.e., only the pair of jobs that was swapped during the last move cannot be swapped again. Apply the technique for four iterations and determine whether the optimal sequence is reached.

(b) Make the length of the tabu-list 3, i.e., the pairs of jobs that were swapped during the last three moves cannot be swapped again. Apply the technique for four iterations and determine whether the optimal sequence is reached.

14.5. Apply the ATC dispatching rule to the following instance of $F3 \mid prmu, p_{ij} = p_j \mid \sum w_j T_j$.

jobs	1	2	3	4
p_j	9	9	12	3
d_j	10	8	5	28
w_j	14	12	1	12

What is the best value for the scaling parameter?

14.6. Apply tabu-search to the instance of $F3 \mid prmu, p_{ij} = p_j \mid \sum w_j T_j$ in Exercise 14.5. Choose as the neighbourhood again all schedules that can be obtained through adjacent pairwise interchanges. Start out with sequence $3, 1, 4, 2$ and apply the technique for four iterations. Keep the length of the tabu-list equal to 2. Determine whether the optimal sequence is reached.

14.7. Consider the same instance as in Exercise 14.5. Now apply simulated annealing to this instance. Adopt the same neighbourhood structure and select neighbors within the neighbourhood at random. Choose $\beta_k = (0.9)^k$. Start with $3, 1, 4, 2$ as the initial sequence. Terminate the procedure after two iterations and compare the result with the result obtained in the previous exercise. Use the following numbers as uniform random numbers:

$$U_1 = 0.91, \;\; U_2 = 0.27, \;\; U_3 = 0.83, \;\; U_4 = 0.17.$$

14.8. Consider the same instance as in Exercise 14.5. Now apply the Genetic Algorithm 14.4.2 to the instance.

(a) Start with a population of the three sequences $3, 4, 1, 2, \quad 4, 3, 1, 2$ and $3, 2, 1, 4$ and perform three iterations.

(b) Replace one of the sequences in the initial population under (a) with the sequence obtained in Exercise 14.5 and perform three iterations.

14.9. Compare the results obtained with the four different approaches in Exercises 14.5, 14.6, 14.7, and 14.8 with one another.

(a) Which one of the four approaches seems to be the most effective for this particular problem?

(b) What type of hybrid technique would be the most effective for this problem?

14.10. Consider the following instance of $1 \mid s_{jk} \mid C_{\max}$ with 6 jobs. The sequence dependent setup times are specified in the table below.

k	0	1	2	3	4	5	6
s_{0k}	-	1	$1+\epsilon$	D	$1+\epsilon$	$1+\epsilon$	D
s_{1k}	D	-	1	$1+\epsilon$	D	$1+\epsilon$	$1+\epsilon$
s_{2k}	$1+\epsilon$	D	-	1	$1+\epsilon$	D	$1+\epsilon$
s_{3k}	$1+\epsilon$	$1+\epsilon$	D	-	1	$1+\epsilon$	D
s_{4k}	D	$1+\epsilon$	$1+\epsilon$	D	-	1	$1+\epsilon$
s_{5k}	$1+\epsilon$	D	$1+\epsilon$	$1+\epsilon$	D	-	1
s_{6k}	1	$1+\epsilon$	D	$1+\epsilon$	$1+\epsilon$	D	-

Assume D to be very large. Define as the neighbourhood of a schedule all schedules that can be obtained through an adjacent pairwise interchange.

(a) Find the optimal sequence.

(b) Determine the makespans of all schedules that are neighbors of the optimal schedule.

(c) Find a schedule with a makespan less than D of which all the neighbors have the same makespan. (The optimal sequence may be described as a "brittle" sequence, while the last sequence may be described as a more "robust" sequence.)

Exercises (Theory)

14.11. What does the ATCS rule reduce to

(a) if both K_1 and K_2 go to ∞,
(b) if K_1 is very close to zero and $K_2 = 1$,
(c) and if K_2 is very close to zero and $K_1 = 1$?

14.12. Consider $Pm \mid\mid \sum w_j C_j$ and the dispatching rule that releases jobs in decreasing order of $w_j/(p_j^k)$. Give an argument for setting the parameter k between 0.75 and 1. Describe the relationship between an appropriate value of k and the number of machines m.

14.13. Consider $Fm \mid p_{ij} = p_j \mid \sum w_j C_j$ and the dispatching rule that releases jobs in decreasing order of $w_j/(p_j^k)$. Give an argument for setting the k larger than 1. Describe the relationship between an appropriate value of k and the number of machines m.

14.14. Consider the following basic mutations that can be applied to a sequence:

(i) An insertion (a job is selected and put elsewhere in the sequence).
(ii) A pairwise interchange of two adjacent jobs.
(iii) A pairwise interchange of two nonadjacent jobs.

(iv) A sequence interchange of two adjacent subsequences of jobs.
(v) A sequence interchange of two nonadjacent subsequences of jobs.
(vi) A reversal of a subsequence of jobs.

Some of these mutations are special cases of others and some mutations can be achieved through repeated applications of others. Taking this into account explain how these six types of mutations are related to one another.

14.15. Show that if the optimality of a rule can be shown through an adjacent pairwise interchange argument applied to an arbitrary sequence, then

(a) the sequence that minimizes the objective is *monotone* in a function of the parameters of the jobs and
(b) the reverse sequence *maximizes* that same objective.

14.16. Determine the number of neighbors of a permutation schedule if the neighbourhood consists of all schedules that can be reached through

(a) any adjacent pairwise interchange;
(b) any pairwise interchange.

14.17. Consider the $1 \mid\mid \sum w'_j E_j + \sum w''_j T_j$ problem. Design a composite dispatching rule for the minimization of the sum of the weighted earliness and tardiness penalties. (Consider first the case where all due dates are equal to $C_{\max}$).

14.18. Describe a neighbourhood and a neighbourhood search technique for a local search procedure that is applicable to a permutation flow shop scheduling problem with the makespan as objective.

14.19. Describe a neighbourhood and a neighbourhood search procedure for the problem $1 \mid r_j, prmp \mid \sum w_j C_j$.

14.20. Design a multiphase procedure for $Fm \mid block \mid \sum w_j T_j$. with zero intermediate storage and blocking. Give proper statistics to characterize instances. Present a composite dispatching rule and design an appropriate neighbourhood for a local search procedure. (*Hint:* the goodness of fit of an additional job to be included in a partial sequence may be considered to be similar to a sequence dependent setup time; the structure of a composite dispatching rule may in some respects look like the ATCS rule).

14.21. Design a scheduling procedure for the problem $Pm \mid r_j, M_j \mid \sum w_j T_j$. Let the procedure consist of three basic steps:

(i) a statistics evaluation step,
(ii) a composite dispatching rule step and
(iii) a simulated annealing step.

Consider the WSPT, LFJ, LFM, EDD and MS rule as rules for possible inclusion in a composite dispatching rule. What factors should be defined for characterization of scheduling instances? What kind of input can the scheduler provide to the three modules?

Comments and References

Morton and Pentico (1994) provide an excellent treatise on general purpose procedures and heuristic techniques for scheduling problems; they cover most of the techniques described in this chapter.

One of the first studies on dispatching rules is due to Conway (1965a, 1965b). A fairly complete list of the most common dispatching rules is given by Panwalkar and Iskander (1977) and a detailed description of composite dispatching rules is given by Bhaskaran and Pinedo (1992). Special examples of composite dispatching rules are the COVERT rule developed by Carroll (1965) and the ATC rule developed by Vepsalainen and Morton (1987). The ATCS rule is due to Lee, Bhaskaran and Pinedo (1997). Ow and Morton (1989) describe a rule for scheduling problems with earliness and tardiness penalties.

A great deal of research has been done on simulated annealing, tabu-search and genetic algorithms with applications to scheduling. The book edited by Aarts and Lenstra (1997) and the book by Hoos and Stützle (2005) give excellent overviews of local search in general. The monograph by Deb (2001) focuses solely on genetic algorithms. For an overview of search spaces for scheduling problems, see Storer, Wu and Vaccari (1992). For studies on simulated annealing, see Kirkpatrick, Gelatt and Vecchi (1983), Van Laarhoven, Aarts and Lenstra (1992) and Matsuo, Suh and Sullivan (1988). For tabu-search, see Glover (1990), Dell'Amico and Trubian (1991) and Nowicki and Smutnicki (1996). For genetic algorithms applied to scheduling, see Lawton (1992), Della Croce, Tadei and Volta (1992) and Bean (1994). The example of the crossover operator that is presented in Section 14.4 is due to Liaw (2000). Aytug, Khouja and Vergara (2003) give an excellent overview of all the different types of crossover operators which are used in genetic algorithms applied to scheduling problems.

A great deal of research has been done on Ant Colony Optimization (ACO) algorithms. For an excellent overview of this area of research, see Dorigo and Stützle (2004). The ACO algorithm for the single machine total weighted tardiness problem, described in Section 14.5, is due to Den Besten, Stützle and Dorigo (2000). A different type of ACO algorithm developed for the same problem is due to Merkle and Middendorf (2003).

The three step procedure described in the discussion section is due to Lee, Bhaskaran and Pinedo (1997).

Chapter 15

More Advanced General Purpose Procedures

The previous chapter covered the more established and the more widely used generic procedures. This chapter focuses on techniques that are more specialized and not as widely used.

The first section focuses on a technique that is a modification of the branch-and-bound procedure. It is referred to as beam search. It tries to eliminate branches in an intelligent way so that not all branches have to be examined. The second section covers decomposition procedures. Chapter 7 already considered a very well-known decomposition technique, namely the shifting bottleneck heuristic. The shifting bottleneck technique is a classical example of a so-called machine-based decomposition procedure. The second section of this chapter describes several other types of decomposition techniques. The third section discusses constraint guided heuristic search procedures which have been developed in the artificial intelligence community. Constraint guided heuristic search is often also referred to as constraint-based programming. A constraint guided heuristic search procedure attempts to find a feasible schedule given all the constraints in the scheduling environment. The fourth section discusses a class of techniques that also originated in the artificial intelligence community. These techniques assume that the scheduling process is based on a market mechanism in which each job has to make bids and pay for machine time. The fifth section focuses on procedures for scheduling problems with multiple objectives.

M.L. Pinedo, *Scheduling*, DOI: 10.1007/978-0-387-78935-4_15,

In practice, most scheduling systems have to deal with multiple objectives and have to be able to do some form of parametric or sensitivity analysis. The discussion section explains the role of general purpose procedures in the design and development of engines for scheduling systems.

15.1 Beam Search

Filtered beam search is based on the ideas of branch-and-bound. Enumerative branch-and-bound methods are currently the most widely used methods for obtaining optimal solutions to NP-hard scheduling problems. The main disadvantage of branch-and-bound is that it usually is extremely time consuming, as the number of nodes that have to be considered is very large.

Consider, for example, a single machine problem with n jobs. Assume that for each node at level k jobs have been selected for the first k positions. There is a single node at level 0, with n branches emanating to n nodes at level 1. Each node at level 1 branches out into $n-1$ nodes at level 2, resulting in a total of $n(n-1)$ nodes at level 2. At level k there are $n!/(n-k)!$ nodes. At the bottom level, level n, there are $n!$ nodes.

Branch-and-bound attempts to eliminate a node by determining a lower bound on the objective for all partial schedules that sprout out of that node. If the lower bound is higher than the value of the objective under a known schedule, then the node may be eliminated and its offspring disregarded. If one could obtain a reasonably good schedule through some clever heuristic before going through the branch-and-bound procedure, then it might be possible to eliminate many nodes. Other elimination criteria (see Chapter 3) may also reduce the number of nodes to be investigated. However, even after these eliminations there are usually still too many nodes that have to be evaluated. For example, it may require several weeks on a workstation to find an optimal schedule for an instance of the $1 \mid\mid \sum w_j T_j$ problem with 40 jobs. The main advantage of branch-and-bound is that, after evaluating all nodes, the final solution is known with certainty to be optimal.

Filtered beam search is an adaptation of branch-and-bound in which not all nodes at any given level are evaluated. Only the most promising nodes at level k are selected as nodes to branch from. The remaining nodes at that level are discarded *permanently*. The number of nodes retained is called the *beam width* of the search. The evaluation process that determines which nodes are the promising ones is a crucial element of this method. Evaluating each node carefully, in order to obtain an estimate for the potential of its offspring, is time consuming. Here a trade-off has to be made: a crude prediction is quick, but may lead to discarding good solutions, while a more thorough evaluation may be prohibitively time consuming. Here is where the filter comes in. For all the nodes generated at level k, a crude prediction is done. Based on the outcome of these crude predictions a number of nodes are selected for a more thorough evaluation, while the remaining nodes are discarded permanently. The number

of nodes selected for a more thorough evaluation is referred to as the *filter width.* Based on the outcome of the more thorough evaluation of the nodes that pass the filter, a subset of these nodes (the number being equal to the beam width which therefore cannot be greater than the filter width) is selected from where further branches are generated.

A simple example of a crude prediction is the following. The contribution of the partial schedule to the objective and the due date tightness or some other statistic of the jobs that remain to be scheduled are computed; based on these values the nodes at a given level are compared to one another and an overall assessment is made.

Every time a node has to undergo a thorough evaluation, all the jobs not yet scheduled are scheduled according to a composite dispatching rule. Such a schedule can still be generated reasonably fast as it only requires a sort. The objective value of such a schedule is an indication of the promise of that node. If a very large number of jobs are involved, nodes may be filtered out by examining a partial schedule that is generated by scheduling only a subset of the remaining jobs with a dispatching rule. This extended partial schedule may be evaluated and based on its value a node may be discarded or retained. If a node is retained, it may be analyzed more thoroughly by having all its remaining jobs scheduled using the composite dispatching rule. The value of this schedule's objective then represents an upper bound on the best schedule among the offspring of that node. The following example illustrates a simplified version of beam search.

Example 15.1.1 (Application of Beam Search)

Consider the following instance of $1 \mid\mid \sum w_j T_j$ (which is the same instance as the one considered in Example 14.3.4).

jobs	1	2	3	4
p_j	10	10	13	4
d_j	4	2	1	12
w_j	14	12	1	12

As the number of jobs is rather small only one type of prediction is made for the nodes at any particular level. No filtering mechanism is used. The beam width is chosen to be 2, which implies that at each level only two nodes are retained. The prediction at a node is made by scheduling the remaining jobs according to the ATC rule. With the due date range factor R being 11/37 and the due date tightness factor τ being 32/37, the look-ahead parameter is chosen to be 5.

A branch-and-bound tree is constructed assuming the sequence is developed starting out from $t = 0$. So, at the jth level of the tree jobs are put into the jth position. At level 1 of the tree there are four nodes: $(1, *, *, *)$, $(2, *, *, *)$, $(3, *, *, *)$ and $(4, *, *, *)$, see Figure 15.1. Applying the ATC rule to the three remaining jobs at each one of the four nodes results in the four

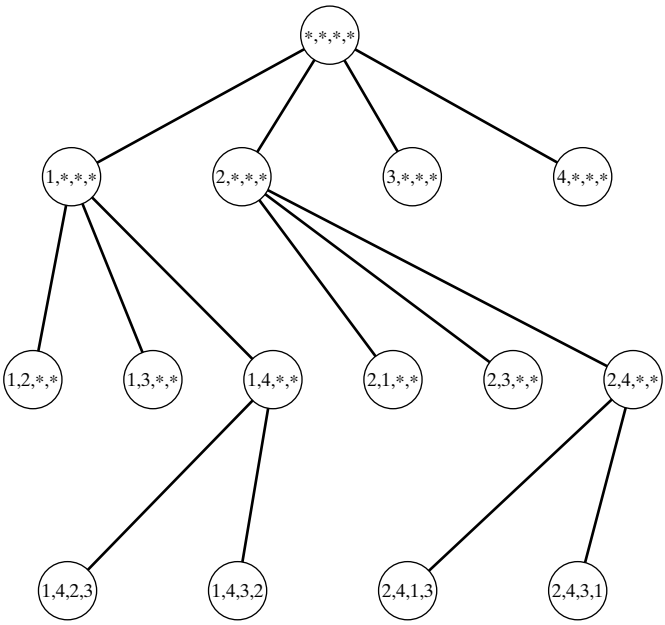


Fig. 15.1 Beam search applied to $1 \mid\mid \sum w_j T_j$

sequences: (1,4,2,3), (2,4,1,3), (3,4,1,2) and (4,1,2,3) with objective values 408, 436, 771 and 440. As the beam width is 2, only the first two nodes are retained.

Each of these two nodes leads to three nodes at level 2. Node $(1,*,*,*)$ leads to nodes $(1,2,*,*)$, $(1,3,*,*)$ and $(1,4,*,*)$ and node $(2,*,*,*)$ leads to nodes $(2,1,*,*)$, $(2,3,*,*)$ and $(2,4,*,*)$. Applying the ATC rule to the remaining two jobs in each one of the 6 nodes at level 2 results in nodes $(1,4,*,*)$ and $(2,4,*,*)$ being retained and the remaining four being discarded.

The two nodes at level 2 lead to four nodes at level 3 (the last level), namely nodes (1,4,2,3), (1,4,3,2), (2,4,1,3) and (2,4,3,1). Of these four sequences sequence (1,4,2,3) is the best with a total weighted tardiness equal to 408. It can be verified through complete enumeration that this sequence is optimal. ||

15.2 Decomposition Methods and Rolling Horizon Procedures

There are several classes of decomposition methods. The best known class of decomposition methods is usually referred to as machine-based decomposition. A prime example of machine-based decomposition is the shifting bottleneck

technique described in Chapter 7. Another class of decomposition methods is referred to as job-based decomposition. A job-based decomposition method is useful when there are constraints with regard to the timing of the various operations of any given job, e.g., when there are minimum and maximum time delays between consecutive operations of a job. A third class consists of the time-based decomposition methods, which are also known as rolling horizon procedures. According to these methods a schedule is first determined for all machines up to a given point in time, ignoring everything that could happen afterwards. After a schedule has been generated up to that given point in time, a schedule is generated for the next time period, and so on. A fourth class of decomposition methods consists of the hybrid methods. Hybrid methods may combine either machine-based or job-based decomposition with time-based decomposition.

Machine-based decomposition is often used in (flexible) flow shop, (flexible) job shop, and (flexible) open shop environments. The decomposition procedure solves the problem by scheduling the machines one at a time, in decreasing order of their criticality indices. That is, the procedure schedules first the machine that is the most difficult to schedule; after having scheduled that machine, it proceeds with scheduling the second most difficult one, and so on. In order to determine the degree of difficulty in scheduling a machine, i.e., the criticality of a machine, a subproblem has to be defined and solved for each machine. The objective of the overall problem determines the type of objective in the single machine subproblem. However, the objective of the single machine (or parallel machines) subproblem is typically a more difficult objective than the objective of the main problem. The single machine subproblem may still be NP-hard. In Chapter 7, it is shown how the $C_{\max}$ objective in the main problem leads to the $L_{\max}$ objective in the subproblem and how the $\sum w_j T_j$ objective in the main problem leads to the $\sum \sum h_{ij}(C_{ij})$ objective in the subproblem, where h_{ij} is piecewise linear convex. It is often not clear how much of an investment in computing time one should make in the search for a solution to the single machine subproblem. It is also not clear how the quality of the solutions of the subproblems affects the quality of the overall solution.

An important aspect of machine-based decomposition is its so-called control structure. The control structure is the framework that determines which subproblem has to be solved when. A control structure may typically lead to a significant amount of reoptimization: After a schedule for an additional machine has been generated, the procedure reoptimizes all the machines that had been scheduled earlier, taking into account the sequence on the machine just scheduled. The sequence of the operations on the machine just scheduled may lead to adjustments in the sequences of operations on the machines scheduled earlier. This reoptimization feature has proven to be crucial with regard to the effectiveness of machine-based decomposition procedures.

In a job-based decomposition method, a subproblem consists of all the operations associated with a given job. The jobs are prioritized and inserted in the schedule one at a time. So the solution of a given subproblem involves the

insertion of the operations of a given job into a partial schedule in such a way that the new schedule is feasible and that the increase in the overall objective caused by the insertion of the new job is minimal. If an insertion of the new job is not feasible, then the jobs inserted before have to be rescheduled.

Time-based decomposition can be applied in any machine environment. There are various forms of time-based decomposition methods. First, a problem can be decomposed by introducing time intervals of a fixed length and by considering in each iteration only jobs that are released during a particular interval. Or, one can partition the problem by considering each time a fixed number of jobs that are released consecutively.

However, there are also time-based decomposition methods that are based on more "natural" partitioning schemes. A natural partitioning point can be defined as a time t with the property that the schedule of the jobs completed before t does not have an effect on (i.e., is independent of) the schedule of the jobs completed after t. For example, consider an instance of the $1 \mid r_j \mid \sum C_j$ problem with a very large number of jobs. This problem is well-known to be strongly NP-hard. Assume, for the time being, that the machine is always processing a job when there are jobs available for processing. Suppose that, at a certain point in time, the machine is idle and there are no jobs waiting for processing; the machine remains idle until the next job is released. It is clear that such a point in time is a natural partitioning point. More formally, let $V(t)$ denote the total amount of processing that remains to be done on the jobs released prior to t. If $V(t)$ is zero or close to zero for some t, then such a t would be an appropriate partitioning point.

Example 15.2.1 (Application of Time-Based Decomposition)
Consider the following instance of $1 \mid r_j \mid \sum C_j$ with 9 jobs.

jobs	1	2	3	4	5	6	7	8	9
r_j	7	7	9	17	33	35	39	40	42
p_j	9	10	6	3	10	8	6	2	2

The graph of $V(t)$ is presented in Figure 15.2. At $t = 33$ the $V(t)$ jumps from 2 to 12. So just before time 33 the $V(t)$ reaches a minimum. One possible partition would consider jobs 1, 2, 3, 4 as one problem and jobs 5, 6, 7, 8, 9 as a second problem. Minimizing $\sum C_j$ in the first problem results in schedule 1, 3, 4, 2. The second problem starts at time 35 and the optimal schedule is 6, 8, 9, 7, 5.

If the jobs have different weights, i.e., the objective is $\sum w_j C_j$, then the partitioning scheme is not as straightforward any more. For example, consider a job with weight 0 (or an extremely low weight). Such a job, most likely, will appear more towards the end of the schedule. ||

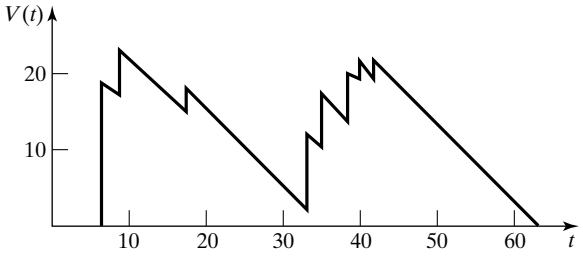


Fig. 15.2 Total amount of processing remaining to be done in system (Example 15.2.1)

Partitioning schemes for problems with due dates may be designed completely differently. Due dates may appear in clusters. Some intervals may have a large number of due dates while other intervals may have a small number. A partitioning of the job set may be done immediately at the end of a cluster of due dates; the subset of jobs to be considered includes all the jobs that have due dates in that cluster.

Example 15.2.2 (Application of Time-Based Decomposition)
Consider the following instance of $1 \mid\mid \sum T_j$ with 9 jobs that are all released at time 0.

jobs	1	2	3	4	5	6	7	8	9
p_j	7	6	7	6	5	5	8	9	4
d_j	22	24	24	25	33	51	52	52	55

There are two clusters of due dates. One cluster lies in the range $[22, 25]$, and the second cluster lies in the range $[51, 55]$. There is a single due date, d_5, that lies in the middle. According to a partitioning scheme that is based on clusters of due dates it makes sense to consider jobs 1, 2, 3, 4 as one subproblem and jobs 5, 6, 7, 8, 9 as a second subproblem.

Any sequence in the first subproblem that schedules job 4 last minimizes $\sum T_j$. Any sequence in the second subproblem that schedules job 9 last and either job 7 or job 8 second to last minimizes $\sum T_j$ in that problem. ||

Control structures can play an important role in time-based decomposition procedures as well. After a time-based decomposition procedure has generated a schedule, a post-processing procedure can be done. In such a post-processing procedure the last couple of jobs of one job set are combined with the first couple of jobs of the next job set. This new set of jobs is then reoptimized (see Figure 15.3).

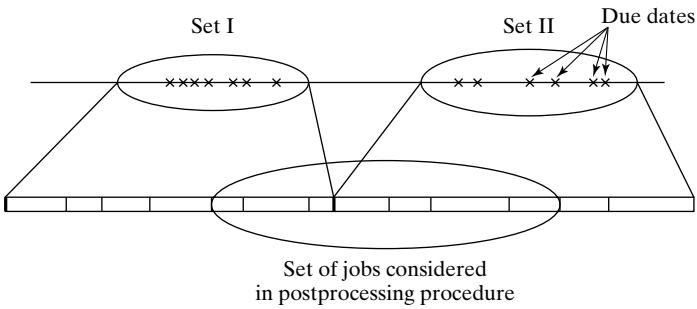


Fig. 15.3 Set of jobs considered in postprocessing procedure

Example 15.2.3 (Application of Time-Based Decomposition)
Consider the following instance of the $1 \mid\mid \sum w_j T_j$. The processing times and due dates are the same as those in Example 15.2.2.

jobs	1	2	3	4	5	6	7	8	9
p_j	7	6	7	6	5	5	8	9	4
w_j	0	1	1	2	1	1	2	1	4
d_j	22	24	24	25	33	51	52	52	55

If the time based decomposition is done in the same way as in Example 15.2.2, then the first subproblem consists again of jobs $1, 2, 3, 4$ and the second subproblem of jobs $5, 6, 7, 8, 9$.

In the first subproblem, any schedule that puts job 1 in the fourth position (i.e., the last slot) minimizes the total weighted tardiness. The total weighted tardiness of the first subproblem is then zero. In the second subproblem any schedule that puts job 8 in the last position and job 5 in the first position minimizes the total weighted tardiness.

What happens now on the borderline between the first and the second subproblem? Job 1 is last in the first subproblem and job 5 is first in the second subproblem. It is clear that an interchange between these two jobs results in a schedule with the same objective value. Are these two schedules optimal? That is clearly not the case. If job 1 is processed all the way at the end of the schedule, i.e., completing its processing at $C_1 = 57$, then the total weighted tardiness of the entire schedule is zero. This last schedule is therefore optimal. ||

Hybrid decomposition methods are more complicated procedures. There are several types of hybrid methods. Some hybrid methods do a machine-based decomposition first, and solve the single machine subproblems using a rolling

horizon procedure. Other hybrid methods do a time-based decomposition first and then find a schedule for each time interval via a machine decomposition method.

The type of decomposition that has to be done first depends on the characteristics of the instance. For example, if there are significant differences between the criticality indices of the different machines, then it may make sense to do a machine-based decomposition first; the scheduling procedure for the subproblems within the machine-based decomposition procedure may be done via time-based decomposition. However, if there are natural partitioning points that are clear and pronounced, then it may make sense to do a time-based decomposition first and solve the subproblems via a machine-based decomposition procedure.

In practice, decomposition methods are often combined with other methods, e.g., local search procedures (see Section 17.3).

15.3 Constraint Programming

Constraint programming is a technique that originated in the Artificial Intelligence (AI) community. In recent years, it has often been implemented in combination with Operations Research (OR) techniques in order to improve its effectiveness.

Constraint programming, according to its original design, only tries to find a good solution that is feasible and that satisfies all the given constraints. However, these solutions may not necessarily be optimal. The constraints typically include release dates and due dates. It is possible to embed a constraint programming technique in a framework that is designed for minimizing a due date related objective function.

In order to illustrate the use of the technique, consider the problem $Pm \mid r_j \mid \sum w_j U_j$. Job j has a weight w_j, a release date r_j and a due date d_j; there are m identical machines in parallel. The objective is to minimize the weighted number of late jobs. This problem can serve as a model for the airport gate assignment problem described in Example 1.1.3 and can be formulated as a Mixed Integer Program (MIP). However, because of the size of the MIP it is impossible to consider instances with more than 20 jobs. Several branch-and-bound schemes have been designed for the MIP with lower bounds generated through various forms of relaxations.

From a constraint programming point of view this nonpreemptive scheduling problem can be described as a search for an assignment of start times for the jobs such that (i) at each point in time not more than m jobs are being processed, (ii) each job starts after its release date, (iii) the weighted number of jobs completed after their respective due dates is less than or equal to a given value W.

Let $\mathcal{U}$ denote an upper bound on the completion times of all jobs. Such an upper bound can be found by taking the last release date and adding the sum of all processing times. The initial domains of the start and completion time

variables can be set equal to $[r_j, \mathcal{U} - p_j]$ and $[r_j + p_j, \mathcal{U}]$. The domain of the value of the objective function can be specified as $[0, W]$, where

$$W = \sum_{j=1}^{n} w_j.$$

Note that within these domain sets a job may end up either late or on time. If a given job *must* be completed on time, then its domain must be made narrower. The start and completion time variables of such a job must then lie within $[r_j, d_j - p_j]$ and $[r_j + p_j, d_j]$.

The constraint programming procedure generates a schedule in a constructive manner, one job at a time. After the start and completion times of some jobs have been determined, the time windows in which the remaining jobs can be scheduled (i.e., their current domains) can be made narrower by propagating constraints that are induced by the partial schedules already put in place. So, after positioning an additional job in the schedule, the domains of the remaining jobs have to be recomputed. Let S'_j (S''_j) denote in each step of the procedure the current earliest (latest) possible starting time of job j. Clearly, $S'_j \geq r_j$ and $S''_j \leq d_j - p_j$. Let C'_j (C''_j) denote the current earliest (latest) possible completion time of job j. Clearly, $C'_j \geq r_j + p_j$ and $C''_j \leq d_j$.

An implementation of constraint programming can usually be made significantly more effective through the use of dominance rules. In order to describe one such rule a definition is needed.

Definition 15.3.1 (Job Preference). *Job j is preferred over job k if*

$$\begin{aligned} p_j &\leq p_k, \\ w_j &\geq w_k, \\ r_j + p_j &\leq r_k + p_k, \\ d_j - p_j &\geq d_k - p_k. \end{aligned}$$

It is clear that this preference ordering is transitive, i.e., if job j is preferred over job k and job k is preferred over job l, then job j is preferred over job l.

Lemma 15.3.2. *If job j is preferred over job k and there exists a feasible schedule in which job k is on time and job j late, then there exists another feasible schedule that is at least as good in which job j is on time and job k late.*

Proof. By contradiction. Consider an optimal schedule σ^* with job j being late, job k on time, and the lemma being violated. Job j is more preferable than job k, $C_k \leq d_k$, and $C_j > d_j$. Now, perform an interchange between jobs j and k. More precisely, job j is started at the starting time of job k in schedule σ^* and job k is moved all the way to the end of the schedule (it will be late). It is easy to see that the new schedule is feasible and still optimal. Job j is now on time and if $w_j > w_k$ (strictly), then the value of the objective function is lower. □

For other examples of dominance rules, see Lemmas 3.4.1 and 3.6.1. Another set of rules that are useful in constraint programming are based on problem decomposition. These rules enable the constraint program to partition the problem of scheduling the remaining jobs into two or more independent subproblems that do not affect one another and that therefore can be solved independently. A decomposition rule may be applied in a constraint guided search procedure any time after a subset of the jobs already have been put in place. The next result provides a basis for decomposing and concatenating schedules.

Lemma 15.3.3. *Let t^* be a point in time such that for any job j that remains to be scheduled either $S''_j + p_j \leq t^*$ or $S'_j \geq t^*$. Any optimal schedule is then a concatenation of an optimal schedule for the problem with those jobs that have due dates less than or equal to t^* and an optimal schedule for the problem with those jobs that have release dates larger than or equal to t^*.*

Proof. The proof is left as an exercise. □

It is clear that after each iteration in a constraint guided search procedure (i.e., after one or more additional jobs have been put in place) the domains of the remaining jobs become more restricted, thereby creating more opportunities for decomposing the remaining problem into smaller subproblems that are easier to solve.

The next result provides a basis for another form of decomposition. In order to state the result, let $[t_1,\ t_2]$ denote an arbitrary time interval and let $J_{[t_1,t_2]}$ denote the subset of jobs among those that are not yet late (i.e., $S'_j + p_j \leq d_j$) and that may end up being processed (at least partially) during $[t_1,\ t_2]$ provided they end up on time (i.e., $t_1 < d_j$ and $t_2 > r_j$).

Lemma 15.3.4. *If there exists a feasible schedule σ for the job set $J_{[t_1,t_2]}$ with all machines being idle just before t_1 and immediately after t_2 and all jobs in $J_{[t_1,t_2]}$ being on time, then there exists an overall optimal schedule σ^* for all n jobs such that between t_1 and t_2 the schedules σ and σ^* are the same.*

Proof. The proof is easy and left as an exercise. □

Consider now any point in time t_1 and let J_{t_1} denote a subset of the remaining jobs that do not have to be late and that can be completed after t_1. A schedule can be generated for set J_{t_1}. Let t_2 ($t_2 > t_1$) denote a time point during the generation of the schedule. If at t_2 a job has been completed after its due date, then t_1 cannot serve as a starting point for an application of Lemma 15.3.4; however, if at t_2 J_{t_2} has become empty, then a valid schedule has been generated for $J_{[t_1,t_2]}$ with all jobs completed on-time.

The following algorithm is based on the ideas described above. A search tree is built. While there are still jobs remaining to be scheduled which may end up either late or on-time, one of these jobs is selected. When generating a schedule, it may turn out (with the partial schedule that is already in place) that it is not possible to generate a complete schedule that is feasible. If that is the case,

then the algorithm must *backtrack*, which implies that some segment(s) of the already established partial schedule have to be annulled.

Algorithm 15.3.5 (Constraint Guided Heuristic Search Procedure)

Step 1. (Job Selection)

Apply a job selection heuristic to choose a job with $C'_j \leq d_j$.
Constrain job j *to be on time.*
If no such job is found and
the weighted number of late jobs is larger than W,
then BACKTRACK
(i.e., annul a segment of the partial schedule and return to Step 1).

Step 2. (Application of Dominance Rules and Propagation of Constraints)

Apply dominance rules and propagate constraints.
Adjust S'_j, S''_j, C'_j *and* C''_j.

Step 3. (Feasibility Check)

Check if there still exists a feasible schedule (including all jobs)
that has a weighted number of late jobs less than or equal to W
(by applying decomposition and interval scheduling rules as well as
heuristics on the jobs remaining to be scheduled).
If no feasible schedule is found, then BACKTRACK.

Step 4. (Stopping Criterion)

If there are still jobs remaining that can end up either on time or late,
then return to Step 1;
Otherwise STOP. ||

The job selection heuristic in Step 1 can be designed in many ways. For example, let J denote the set of jobs remaining to be scheduled, i.e., jobs that have not been put in the schedule yet and that may end up in the final schedule either on time or late. Let

$$\left(\frac{w_j}{p_j}\right)^* = \max\{w_j/p_j \mid j \in J\}$$

Let J' denote all jobs from set J that have a w_j/p_j ratio higher than $0.9 \times (w_j/p_j)^*$. The heuristic selects among these jobs the job that, if it is completed on time, has the widest interval $[S'_j, C''_j]$. Basically, the heuristic "bets" that it is better to schedule a job that has a high w_j/p_j ratio and a wide domain, i.e., a more preferred job. The reasoning behind this is that when later on in the process more jobs are added to the schedule, there is still some freedom that allows this job to be moved either forward or backward in time.

The feasibility check in Step 3 is a hard problem. A scheduling heuristic has to be used here. Such a heuristic may work either forward or backward in time. If such a heuristic works forward in time (from left to right), then it typically would try to schedule the jobs through a combination of the EDD rule and the

WSPT rule, e.g., the ATC rule. If it works backwards in time, then it would try to apply similar rules in a backwards manner.

15.4 Market-Based and Agent-Based Procedures

Consider a flexible job shop with c workcenters. Each workcenter has a number of identical machines in parallel. The jobs have different release dates and the objective is to minimize the total weighted tardiness $\sum w_j T_j$. This problem can be referred to as $FJc \mid r_j \mid \sum w_j T_j$.

Each job has a Job Agent (JA) who receives a budget at the release of the job. The budget of the JA is a function of the weight of the job, the tightness of its due date, and the total amount of processing it requires at the various workcenters. At each point in time, when an operation of a job has been completed at a workcenter, the JA sends out a call for bids from the machines at the next workcenter on the job's route. The information specified in this call for bids includes the preferred time frame in which the JA wants the next operation to be done. The JA determines the desired time frame by taking the job's final due date into account, as well as an estimate of the remaining time it needs to complete all remaining operations.

Each machine has a Machine Agent (MA). The cost structure of the machine consists of a fixed cost per unit time, that is incurred it all times (whether or not the machine is processing a job). The cost per unit time of each machine is known to all parties. If a JA issues a call for bids, then the machines that can process this particular operation of that job submit their bids. The information in a bid includes the time that the machine intends to process that operation as well as the price. The MA, of course, would like the price to be higher than the fixed cost, because then he makes a profit. A machine is allowed to submit multiple bids to the same JA for different time periods with different prices. The rules and heuristics an MA uses to generate bids can be complicated.

It may happen that an MA receives at the same point in time several calls for bids from different JAs. The MA may bid on any subset as long as the time periods specified in the different bids do not overlap. The MA's goal is to make as much money as possible on all the processing it does.

The JA receives, as a result of his call for bids, a number of bids and compares them according to a decision rule that takes the timing of the processing as well as the pricing into account. It then decides to whom to give the award. The decision rules and heuristics the JA uses when making awards take into account the times and the prices stated in the bids, as well as the characteristics of the job (its weight, its due date, and the processing times of all remaining operations). The goal the JA tries to optimize when making an award is a composite objective that includes the estimated weighted tardiness of the job if the next operation is done in the given time frame as well as the amount it has to pay for that operation and estimates for the amounts it has to pay other machines for the subsequent operations. The JA cannot spend more on awards

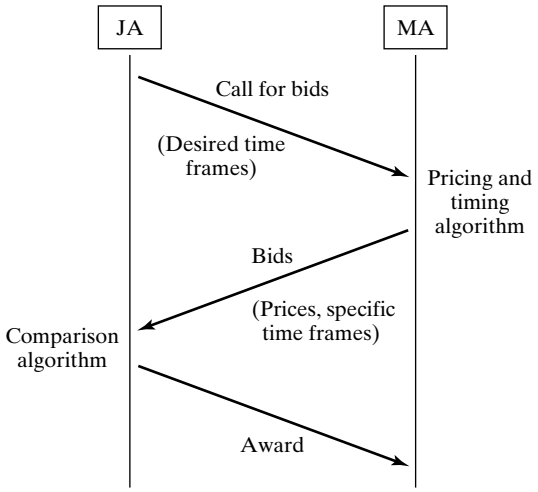


Fig. 15.4 Overview of bidding and pricing process

than what he has in his budget. The total budget of a JA is the total cost of processing all the operations on the various machines (including fixed costs as well as running costs) multiplied by a constant that depends on the weight of the job.

However, it may occur that a JA, after sending out a call, does not receive any bid at all. In such a case, the JA must issue a new call for bids, and change the relevant information in the call. For example, the JA may ask for a time frame that is later than the time frame originally requested. This implies that at one point in time, several iterations may take place going back and forth between the JA and the various MAs. An overview of the bidding and pricing process is depicted in Figure 15.4.

Scheduling that is based on contract negotiations is characterized by the rules used by the various Machine Agents to determine their bids as well as by the rules used by the Job Agents to make the awards. These decision rules depend strongly on budgetary constraints as well as on the level of information available to the JA and MA.

Summarizing, each bidding and pricing mechanism has its own characteristics; they are mainly determined by

(i) the Budgetary Constraints,
(ii) the Information Infrastructure, and
(iii) the Bidding and Pricing Rules.

The Budgetary Constraints: The budget of the JA of job j, B_j, is a function of the total amount of processing of all the operations of the job, the amount

of slack there is with regard to the due date, and the weight of the job, i.e.,

$$B_j = f\Big(w_j, (d_j - r_j - \sum_{i=1}^{n} p_{ij}), \sum_{i=1}^{n} p_{ij}\Big)$$

The B_j may also be regarded as a value that is assigned to the job by a higher level manager or coordinator and that gives an indication regarding the overall priority of job j.

The Information Infrastructure: A framework with a very high level of information assumes that the JA's as well as the MA's have complete information: that is, they know exactly what has happened up to now anywhere in the entire shop, which jobs are still circulating in the shop, their remaining operations and their due dates. They also know everything about the job budgets and about the amounts already spent. However, nobody has any information about future contracts.

A framework with a very low level of information is a setting where a JA only has information concerning the processing of its own operations in the past (pricing, awards, processing times, and so on), but no information concerning the processing of operations of other jobs. An MA only has information concerning the processing of operations on his own machine in the past, and what he himself has committed to for the future (including the prices agreed upon). The MA does not have any detailed information concerning the processing of operations on other machines. (However, an MA may have some limited information concerning past processing on other machines; he may have lost a bid for a job to another machine in the past and may conclude that the other machine must have processed the operation in question at a lower price than he himself had quoted in his bid.) In such an environment the levels of the bids depend on past experiences of both the JA and the MA. This implies that in the beginning of the scheduling process the prices may not reflect true market conditions, since little information is available. If the number of jobs is small, then this could be a problem. It may then be hard to make appropriate decisions since the total level of information is very low. For a problem with a small number of jobs it makes sense for the JAs and MAs to have more information already at the outset.

An intermediate level of information is a situation in which the JA and MA have some general statistics with regard to the problem, i.e., the average amount of processing, the distribution of the processing times, the distribution of the weights of the jobs, the due date tightness factors, and so on. If the scheduling problem has a very large number of jobs and a learning process takes place, then both the JA and the MA will have, after a while, some forms of estimation procedures and statistics.

The Bidding and Pricing Rules: If the time frame requested in a call for bids is such that the MA believes that the machine will end up with idle periods before and after the processing of this operation (because the idle periods may not be large enough to accommodate any other operation), then the MA may

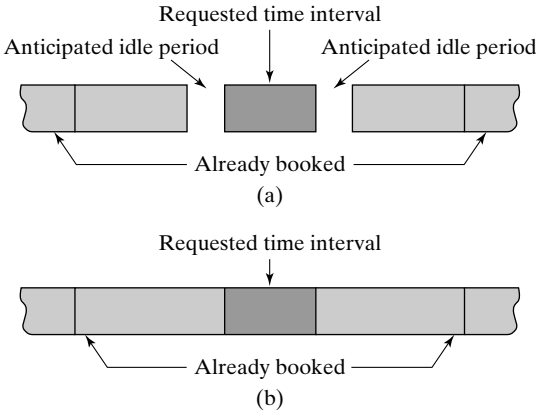


Fig. 15.5 Pricing based on goodness of fit

price its bid somewhat higher just to receive some compensation for the anticipated idle periods (see Figure 15.5.a). Or, if the MA expects more calls for bids that may result in better fitting jobs, he may not bid at all (when the MA has complete information he may have a pretty good idea about future calls for bids).

If an MA expects many calls for bids in the immediate future, he may price his bids on the higher side, because he is not afraid of losing out. In the same vein, towards the end of the process, when the number of calls for bids dwindles, prices may come down.

On the other hand, if the requested time period in the call for bids is a time period that happens to be exactly equal to an idle period in the machine's current schedule (see Figure 15.5.b), the MA may submit a bid on the low side because a perfect fit is advantageous for the machine in question and the MA would like to be sure that no other MA will come up with a lower bid. So the pricing in a bid of an MA is determined by a number of factors. The three most important ones are:

(i) The goodness of fit of the operation in the current machine schedule.
(ii) The anticipated supply of machine capacity and the level of competition.
(iii) The anticipated demand for machine capacity.

Expressions of goodness of fit can be developed for most information infrastructure settings. Even in a framework with a very low level of information a reasonable accurate estimate for the goodness of fit can be made, since the MA should have, even in such an environment, all the necessary information. On the other hand, estimates for the anticipated supply of machine capacity and demand for machine capacity may depend on the information infrastructure. The accuracy of such an assessment is, of course, higher when more information is available.

The selection heuristic a JA uses when receiving multiple bids depends, of course, on the timing and the cost. Certain simple dominance rules are evident: a bid at a lower cost at an earlier time dominates a bid at a higher cost at a later time. The JA makes its selection based on remaining slack and on the expectation of future delays at subsequent workcenters. When more exact information is available, the JA may be more willing to postpone the processing.

To determine the optimal decision rules for both the JA's and the MA's is not easy. The decision rules of the MAs depend heavily on the level of information available to the JAs and MAs. In general, it tends to be easier to determine a fair price the more information is available.

Example 15.4.1 (Bidding and Pricing in a Flexible Flow Shop)

Consider the last stage in a flexible flow shop. This last stage has two identical machines. All jobs, after having been processed at that stage, leave the system. Suppose the current time t is 400. Machine 1 is idle and does not have any commitments for the future and machine 2 is busy until time 450 (machine 2 is processing a job that started at time 380 and that has a duration of 70). At time 400 a JA sends out a call for bids. The processing time of the last operation of this job is 10 and the due date of the job is 390 (i.e., it is already overdue) and the job's weight is 5. So it is incurring for each additional time unit in the system a penalty of 5. The fixed costs of operating each machine is 4 dollars per unit time, i.e., a machine has to charge at least \$4 per unit time that it is processing a job. Which price should machine 1 quote the JA for processing the job starting immediately at time 400, assuming machine 1 has complete information? The only competitor of machine 1 is machine 2 which only can start processing this operation at time 450. Machine 2 has to charge at least $4 \times 10 = 40$ in order to make a profit. The job would be completed at 460; the tardiness of the job is 70 and the total penalty of this job is $70 \times 5 = 350$. If machine 1 would process the job, then the job would be completed at time 410, its penalty being 20×5 is 100. So processing on machine 1 reduces the penalty by 250. In order to get the award, the MA of machine 1 has to quote a price that is less than 350 + 40 - 100 = 290.

What happens now if the information is incomplete? The MA of machine 1 has to charge at least 40 to cover its cost. If machine 1 had received at time 380 a call for bids on the processing of the operation that went at that point to machine 2, then machine 1 knows that machine 2 has to be busy till time 450 and the case reduces to the case considered before. However, if machine 1 did not receive that call for bids at time 380, then it does not have any idea about what is being processed on machine 2. The price that machine 1 will charge may depend on completely different factors, for example, on the prices at which it has been awarded contracts in the past. ||

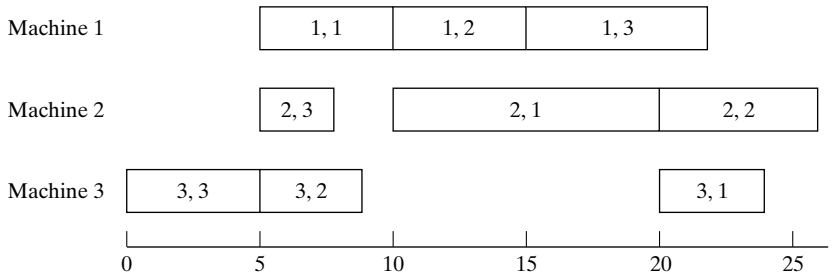


Fig. 15.6 Final schedule in Example 15.4.2

Example 15.4.2 (Application of a Market-Based Procedure to a Job Shop)
Consider the following instance of $Jm \mid r_j \mid \sum w_j T_j$.

job	w_j	r_j	d_j	machine sequence	processing times
1	1	5	24	1,2,3	$p_{11} = 5$, $p_{21} = 10$, $p_{31} = 4$
2	2	0	18	3,1,2	$p_{32} = 4$, $p_{12} = 5$, $p_{22} = 6$
3	2	0	16	3,2,1	$p_{33} = 5$, $p_{23} = 3$, $p_{13} = 7$

At time 0 both jobs 2 and 3 are interested in machine 3. Both jobs have the same weight and the same total amount of processing. However, job 3 has a tighter due date than job 2. So it is likely that $B_3 > B_2$ and that job 3 would put in a higher bid and win the bidding contest. The next decision moment occurs at time 5. Job 3 would like to go on machine 2, job 1 would like to go on machine 1, and job 2 still wants to go on machine 3. None of the three jobs faces any competition. So the JAs and the MAs should be able to come to an agreement. The next decision moment occurs at time 10. Both jobs 3 and 2 want to go on machine 1; job 1 wants to go on machine 2. So jobs 3 and 2 again have to compete with one another, now for machine 1. Since the remaining slack of job 2 is $(18 - 10 - 11) = -3$ and the remaining slack of job 3 is $(16 - 10 - 7) = -1$, it appears that job 2 will win the contest. So job 2 goes on machine 1 and job 1 goes on machine 2. At subsequent decision moments there are no contests between jobs for machines. So the resulting schedule is the one depicted in Figure 15.6. The total weighted tardiness of this schedule is 28.

Note that this schedule is the same as the one generated by the shifting bottleneck procedure described in Chapter 7. However, this schedule is not optimal. The optimal schedule has a total weighted tardiness of 18. ||

Example 15.4.3 (Application of a Bidding Procedure in Practice)

A simple bidding procedure has been implemented in the paint shop of an automobile manufacturing facility and the system has been working quite effectively. There is a dispatcher, which functions as JA, and there are two paint booths. Each paint booth has its own space where a fixed number of trucks can queue up. Each one of the paint booths has an MA. The dispatcher sends out an announcement of a new truck. The dispatcher functions as a JA and his announcement is equivalent to a Call for Bids. A paint booth has three bidding parameters, namely, the color of the last truck that has gone through, whether or not the queue is empty, and (in case the queue is not empty) whether there is space in the queue. Each MA sends the values of the three parameters to the dispatcher. The dispatcher uses the following decision rules when he makes an award: If the color of the last truck at a paint booth is the same as the color of the truck to be dispatched, then the dispatcher makes the award. If the last colors at both paint booths are different, and one of the two queues is empty, the dispatcher makes the award to the booth with the empty queue. If both queues have trucks waiting, then the dispatcher makes the award to a booth that has space in the queue. Even though this bidding process is very simple, it performs well. ||

A careful study of Examples 15.4.2 and 15.4.3 may lead to a conclusion that the bidding and pricing mechanisms described in this section may behave somewhat myopically. When a JA sends out a call for bids only for the next operation to be processed, then the results of the bidding and pricing rules may be comparable to dispatching rules. Of course, if a bidding and pricing rule is made more elaborate, then the mechanism performs less myopically. It is easy to construct a more sophisticated framework. For example, suppose a JA is allowed to issue simultaneously several calls for bids to various workcenters for a number of operations that have to be processed one after another. If the JAs issue calls for bids for a number of their operations the moment they enter the shop, then the scheduling process of each one of the MAs becomes considerably more complicated. The bidding and pricing rules will be, of course, significantly more involved. In such a framework it may be more likely that a JA has to go back and forth negotiating with several machines at the same time just to make sure that the time commitments with the various machines do not overlap. In such frameworks it may also be possible that a job may not be ready at the time of a commitment.

The mechanisms described in this section can be generalized easily to more complicated machine environments. Consider a flexible job shop in which each workcenter does not consist of a bank of *identical* machines in parallel, but rather a bank of *unrelated* machines in parallel. In this case, if a machine makes a bid for a certain operation, then its pricing may depend on how fast it can do the operation relative to the other machines in the same workcenter. For example, if a machine knows it can do an operation much faster than any other machine in the workcenter, and it can do this operation also much faster than

other operations that will come up for bidding subsequently, then it may take this into account in its pricing.

Bidding and pricing mechanisms, when applied to a completely deterministic problem, cannot perform as well as an optimization technique (e.g., a shifting bottleneck procedure) that is designed to search for a global optimum. However, distributed scheduling may play an important role in practice in more complicated machine environments with a long horizon that are subject to various forms of randomness (in settings for which it would be hard to develop optimization algorithms). In settings with long time horizons one more concept starts playing a role, namely the concept of learning. For both the JA and the MA the historical information starts playing a more important role in their decision-making.

15.5 Procedures for Scheduling Problems with Multiple Objectives

Chapter 4 presents solutions to a few single machine scheduling problems with two objectives. However, the problems considered in Chapter 4 are relatively simple. In practice the machine environment is usually more complex and the scheduler often has to deal with a weighted combination of many objectives. The weights may be time or situation dependent. Typically, a scheduler may not know the exact weights and may want to perform a parametric analysis to get a feeling for the trade-offs. When a scheduler creates a schedule that is better with respect to one objective, he may want to know how other objectives are affected.

When there are multiple objectives the concept of Pareto-optimality plays a role. A schedule is Pareto-optimal if it is impossible to improve on one of the objectives without making at least one other objective worse (see Chapter 4). Usually, only Pareto-optimal schedules are of interest. (However, in practice a schedule may be Pareto-optimal only with respect to the set of schedules generated by the heuristic but not with respect to the set of *all* possible schedules.) When there are multiple objectives, the scheduler may want to view a set of Pareto-optimal schedules before deciding which schedule to select. So a system must retain at all times multiple schedules in memory.

There are major differences between multi-objective problems that allow preemptions and those that do not allow preemptions. Problems that allow preemptions are often mathematically easier than those that do not allow preemptions. In a preemptive environment there are typically an infinite number of feasible schedules, whereas in a nonpreemptive environment there are usually only a finite number of feasible schedules. The trade-off curve in a preemptive environment is often a continuous function that is piece-wise linear and convex. The trade-off curve in a nonpreemptive environment consists of a set of points; the envelope of such a set of points is always decreasing.

Efficient (polynomial time) algorithms exist only for the simplest multi-objective scheduling problems. Most multi-objective scheduling problems are NP-hard. However, in practice it is still necessary to find good schedules in a fast and effective manner. Various adaptations of conventional heuristic procedures can be applied to multi-objective problems. The conventional approaches that can be used include:

(i) procedures based on dispatching rules,

(ii) branch-and-bound and filtered beam search, and

(iii) local search techniques.

One approach that is particularly useful for a parametric analysis of a multi-objective problem is

(iv) perturbation analysis.

The remaining part of this section discusses these four approaches in more detail.

Procedures based on dispatching rules. It is often the case that a dispatching rule that is effective with regard to one objective does not perform particularly well with regard to a second objective, while a different dispatching rule that is effective with regard to the second objective does not perform particularly well with regard to the first one. There are many ways of combining different dispatching rules within a single framework.

One way is described in Chapter 14 and is exemplified by the composite ATC and ATCS dispatching rules. A composite dispatching rule combines two (or more) dispatching rules within a single framework. Job j has a single index function $I_j(t)$ to which each rule contributes in a certain way. Every time a machine is freed a job is selected based on the values of the indices of the waiting jobs. Adjustments of the scaling parameters may result in different Pareto-optimal schedules. This is to be expected, since a particular combination of scaling parameters will emphasize a certain subset of the basic rules within the composite rule and each one of the basic rules within the composite rule may favor a different objective. So by adjusting the scaling parameters it may be possible to generate schedules for different parts of the trade-off curve.

Another approach for building a procedure that is based on dispatching rules is by analyzing the partial schedule and the status of the scheduling process. Through such an analysis it can be determined at any point in time which dispatching rule is the most appropriate one to use. The following example illustrates how two dispatching rules can be combined in a preemptive environment with multiple objectives.

Example 15.5.1 (A Trade-Off Curve in a Preemptive Environment)
Consider the following instance of $P3 \mid prmp \mid \theta_1 C_{\max} + \theta_2 \sum C_j$ with three identical machines in parallel and five jobs. The processing times of the jobs are 5, 6, 8, 12, and 14. Preemptions are allowed and there are two objectives: the makespan and the total completion time.

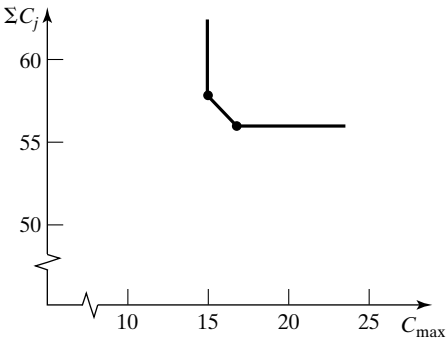


Fig. 15.7 Trade-off curve in a preemptive environment (Example 15.5.1)

The trade-off curve is piece-wise linear decreasing convex, see Figure 15.7. The minimum total completion time is 56. The minimum makespan when the total completion time is 56 is 17 (see Exercise 5.22). The overall minimum makespan is 15. The minimum total completion time when the makespan is 15 is 58 (see Exercise 5.19). The coordinates $(56, 17)$ and $(58, 15)$ are breakpoints in the trade-off curve. One endpoint of the trade-off curve is achieved with a nonpreemptive schedule while the other endpoint is achieved with a preemptive schedule. All the points on the trade-off curve can be obtained by switching, in an appropriate manner, back and forth between the SPT and LRPT rules. Actually, the algorithm for generating a schedule that corresponds to any given point on the trade-off curve is polynomial (see Exercise 5.19).

From Figure 15.7 it is clear that if $\theta_1 = \theta_2 = 0.5$, both breakpoints (and all the points in between) are optimal.

From a mathematical point of view, a scheduler can limit himself to those schedules that correspond to the breakpoints in the trade-off curve. However, it may be the case that other considerations also play a role. For example, the scheduler may prefer a nonpreemptive schedule over a preemptive one. ||

As stated before, nonpreemptive problems are usually harder than their preemptive counterparts. The following example illustrates the application of a procedure that is based on dispatching rules in a nonpreemptive environment.

Example 15.5.2 (Two Objectives in a Nonpreemptive Setting)

Consider an instance of $Pm \mid\mid \theta_1 \sum w_j T_j + \theta_2 C_{\max}$. There are m machines in parallel and n jobs. Job j has a processing time p_j, a due date d_j, and a weight w_j. The objectives are the total weighted tardiness $\sum w_j T_j$ and the makespan $C_{\max}$. The composite objective is $\theta_1 \sum w_j T_j + \theta_2 C_{\max}$, where θ_1 and θ_2 are the weights of the two objectives. An appropriate rule for the first

objective is the Apparent Tardiness Cost first (ATC) rule, which, when a machine is freed at time t, selects the job with the highest index

$$I_j(t) = \frac{w_j}{p_j} \exp\Big(- \frac{\max(d_j - p_j - t, 0)}{K\bar{p}} \Big),$$

where $\bar{p}$ is the average of the remaining processing times and K is a scaling parameter (see Chapter 14). On the other hand, a rule suitable for minimizing the makespan is the Longest Processing Time first (LPT) rule. Clearly, it is better to balance the m machines at the end of the process with shorter jobs than with longer jobs.

Combining the two rules for the joint objective can be done as follows. Since the makespan is determined mainly by the assignment of the jobs to the machines at the end of the process, it makes sense to schedule the jobs early in the schedule according to the ATC rule and switch at some point in time (or after a certain number of jobs) to the LPT rule. The timing of the switch-over depends on several factors, including the relative weights of the two objectives (θ_1 and θ_2), the number of machines and the number of jobs. So a rule can be devised that schedules the jobs in the beginning according to ATC and then switches over to LPT. ||

Examples 15.5.1 and 15.5.2 are, of course, very stylized illustrations of how different dispatching rules can be combined within a single framework. A procedure that is based on dispatching rules is usually more complicated than those described above.

Branch-and-bound and filtered beam search. These methods are applicable to problems with multiple objectives in the same way as they are applicable to problems with a single objective. The manner in which the branching tree is constructed is similar to the way it is constructed for problems with a single objective. However, the bounding techniques are often different. In filtered beam search the subroutines at each one of the nodes in the branching tree may use either composite dispatching rules or local search techniques.

Local search techniques. The four most popular local search techniques, i.e., simulated annealing, tabu-search, genetic algorithms, and Ant Colony Optimization are discussed in Chapter 14. These techniques are just as suitable for multi-objective problems as they are for single objective problems. However, there are differences. With a single objective a local search procedure must retain in memory only the very best schedule found so far. With multiple objectives a procedure must keep in memory all schedules that are Pareto-optimal among the schedules generated so far. The criteria according to which a new schedule is accepted depends on whether the new schedule is Pareto-optimal among the set of retained schedules. If a new schedule is found that is Pareto-optimal with respect to the set of schedules that are currently in memory, then all current schedules have to be checked whether they remain Pareto-optimal after the new schedule has been included. Genetic algorithms are particularly well suited for multi-objective problems, since a genetic algorithm is already

designed to carry a population of solutions from one iteration to the next. The population could include Pareto-optimal schedules as well as other schedules. The way the children of a population of schedules are generated may depend on the weights of the various individual objectives within the overall objective. The procedure may focus on those schedules that are the best with respect to the objectives with the greatest weight and use those schedules for cross-overs and other manipulations.

Perturbation analysis. Perturbation analysis is important when a parametric analysis has to be done. The weights of the various objectives may not be fixed and may be allowed to vary. In practice this may happen when a scheduler simply does not know the exact weights of the objectives, and would like to see optimal schedules that correspond to various sets of weights. Suppose that a scheduler has a schedule that is very good with respect to one set of weights but would like to increase the weight of one objective and decrease the weight of another. He may want to consider the contribution that each job makes to the objective that now has been assigned a larger weight and select those jobs that contribute the most to this objective. The positions of these jobs can now be changed via a local search heuristic in such a way that their contribution to the objective that has become more important decreases, while attempting to ensure that their contributions to the objectives that have become less important do not increase substantially.

The next example illustrates the effectiveness of combining dispatching rules with local search or perturbation analysis in a nonpreemptive environment.

Example 15.5.3 (Three Objectives in a Nonpreemptive Setting)
Consider the nonpreemptive scheduling problem described in Example 14.2.1. However, consider now three objectives: the total weighted tardiness, the makespan, and the maximum lateness, i.e.,

$$1 \mid s_{jk} \mid \theta_1 \sum w_j T_j + \theta_2 C_{\max} + \theta_3 L_{\max}.$$

One can plot the Pareto-optimal points for any two of the three objectives. The trade-offs between the total weighted tardiness and the makespan objective are particularly simple. Sequence $2, 4, 3, 1$ minimizes both the makespan and the total weighted tardiness; there are no trade-offs. The trade-offs between the total weighted tardiness and the maximum lateness are somewhat more complicated. If these two objectives are considered, there are five Pareto-optimal schedules, namely

$$(2, 4, 3, 1),\ (2, 4, 1, 3),\ (1, 3, 4, 2),\ (1, 4, 3, 2),\ (1, 4, 2, 3)$$

(see Figure 15.8). However, note that even though schedule $2, 4, 1, 3$ is Pareto-optimal, it can never be optimal when the objective is $\theta_1 L_{\max} + \theta_2 \sum w_j T_j$. For any set of weights θ_1 and θ_2, either schedule $1, 4, 3, 2$ or schedule $1, 4, 2, 3$ dominates schedule $2, 4, 1, 3$.

sequence	*rule*	$\sum w_j T_j$	$C_{\max}$	$L_{\max}$	Pareto-optimal
1,2,3,4		204	54	32	no
1,2,4,3		139	51	30	no
1,3,2,4	(SST)	162	49	27	yes
1,3,4,2	(MS, EDD)	177	53	19	yes
1,4,2,3		101	53	32	yes
1,4,3,2		129	52	20	yes
2,1,3,4	(SST)	194	50	28	no
2,1,4,3		150	50	29	yes
2,3,1,4		203	51	29	no
2,3,4,1		165	54	42	no
2,4,1,3	(SPT)	110	51	30	yes
2,4,3,1	(SPT, SST, ATCS)	98	48	36	yes
3,1,2,4		215	53	31	no
3,1,4,2		213	55	21	no
3,2,1,4		211	53	31	no
3,2,4,1		159	54	42	no
3,4,1,2		191	59	34	no
3,4,2,1		135	54	42	no
4,1,2,3		140	58	37	no
4,1,3,2		162	56	24	no
4,2,1,3	(WSPT)	118	53	32	no
4,2,3,1	(WSPT)	122	54	42	no
4,3,1,2		146	55	30	no
4,3,2,1		102	52	40	no

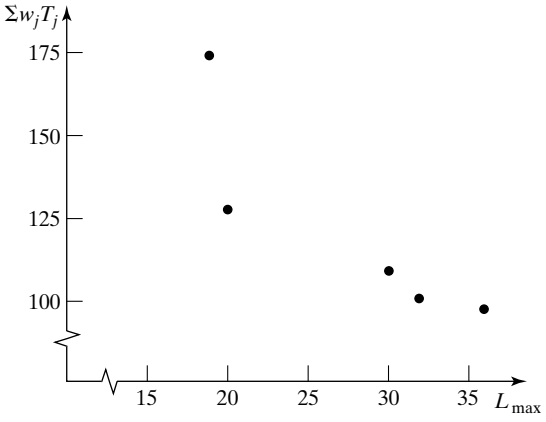


Fig. 15.8 Trade-off curve in a nonpreemptive environment (Example 15.5.3)

If the objectives $C_{\max}$ and $L_{\max}$ are considered, then there are four Pareto-optimal schedules, namely

$$(2,4,3,1),\ (1,3,4,2),\ (1,4,3,2),\ (1,3,2,4).$$

From the two sets of Pareto-optimal schedules it is clear how useful local search techniques and perturbation analysis are in conjunction with dispatching rules. Each set of Pareto-optimal schedules contains schedules that can be obtained via dispatching rules. All other schedules in these two sets are either one or at most two adjacent pairwise interchanges away from one of the schedules obtained via a dispatching rule. ||

In practice, the best approach for handling multi-objective problems may be a combination of several of the four approaches described above. For example, consider a two phase procedure where in the first phase several (composite) dispatching rules are used to develop a number of reasonably good initial feasible schedules that are all different. One schedule may have a very low value of one objective while a second may have a very low value of another objective. These schedules are then fed into a genetic algorithm that attempts to generate a population of even better schedules. Any schedule obtained in this manner can also be used in either a branch-and-bound approach or in a beam search approach, where a good initial feasible schedule reduces the number of nodes to be evaluated.

15.6 Discussion

The heuristic approaches described in this chapter are all different. They tend to work well only on certain problems and then often only on instances with certain characteristics. However, their usefulness lies often in their contribution to the overall performance when put in a framework in which they have to operate with one or more other heuristic approaches. For example, in the shifting bottleneck procedure many different approaches have been used in the optimization of the single machine subproblem, namely

(i) composite dispatching rules,
(ii) dynamic programming, and
(iii) local search.

There is another class of techniques that is based on *machine pricing* and *time pricing* principles; this class is somewhat similar to the market-based and agent-based procedures. Complicated job shop scheduling problems can be formulated as mathematical programs. These mathematical programs usually have several sets of constraints. For example, one set of constraints may enforce the fact that two jobs cannot be assigned to the same machine at the same point in time. Such a set may be regarded as machine capacity constraints. Another set of constraints may have to ensure that certain precedence constraints are

enforced. Disregarding or relaxing one or more sets of constraints may make the solution of the scheduling problem significantly easier. It is possible to incorporate such a set of constraints in the objective function by multiplying it with a so-called *Lagrangean* multiplier. This Lagrangean multiplier is in effect the penalty one pays for violating the constraints. If the solution of the modified mathematical program violates any one of the relaxed constraints, the value of the Lagrangean multiplier has to be changed at the next iteration of the scheduling process in order to increase the penalty of violation and encourage the search for a solution that does not violate the relaxed constraints.

The third section of Chapter 18 focuses on the design of scheduling engines and algorithm libraries. The scheduling engine has to be designed in such a way that the scheduler can easily build a framework by combining a number of different techniques in the way that is the most suitable for the particular problem (or instance) at hand.

Exercises (Computational)

15.1. Consider the following instance of $1 \mid r_j \mid \sum w_j C_j$.

jobs	1	2	3	4	5	6	7	8	9
r_j	7	7	9	17	33	35	39	40	42
w_j	1	5	1	5	5	1	5	1	5
p_j	9	10	6	3	10	8	6	2	2

Design a job-based decomposition procedure for this problem and apply the procedure to the given instance. Compare your result with the result in Example 15.2.1 when all weights are equal.

15.2. Consider $1 \mid r_j, prmp \mid \sum w_j T_j$.

jobs	1	2	3	4	5	6	7	8	9
p_j	7	6	7	6	5	5	8	9	4
w_j	6	7	7	6	5	6	8	9	6
r_j	2	8	11	11	27	29	31	33	45
d_j	22	25	24	24	42	51	53	52	55

Design a job-based decomposition technique when the jobs are subject to release dates and preemptions are allowed. Apply your procedure to the instance above.

15.3. Apply the constraint guided heuristic search technique to the following instance of $Jm \mid\mid L_{\max}$

jobs	*machine sequence*	d_j	*processing times*
1	1, 2, 3	28	$p_{11} = 10, \quad p_{21} = 8, \quad p_{31} = 4$
2	2, 1, 4, 3	22	$p_{22} = 8, \quad p_{12} = 3, \quad p_{42} = 5, \quad p_{32} = 6$
3	1, 2, 4	21	$p_{13} = 4, \quad p_{23} = 7, \quad p_{43} = 3$

15.4. Consider the instance of $Jm \mid r_j \mid \sum w_j T_j$ described in Example 15.4.2.

job	w_j	r_j	d_j	*machine sequence*	*processing times*
1	1	5	24	1,2,3	$p_{11} = 5$, $p_{21} = 10$, $p_{31} = 4$
2	2	0	18	3,1,2	$p_{32} = 4$, $p_{12} = 5$, $p_{22} = 6$
3	2	0	16	3,2,1	$p_{33} = 5$, $p_{23} = 3$, $p_{13} = 7$

Apply the constraint guided heuristic search technique to this instance. (*Hint:* In contrast to the $L_{\max}$ objective, the different jobs now may have completely different tardinesses. The following heuristic may be used. Assume that all the jobs end up with approximately the same amount of tardiness penalty $w_j T_j$. Parametrize on this tardiness penalty: Assume that each job has the same weighted tardiness penalty $w_j T_j = z$. The z together with the weight w_j and the due date d_j translates into a deadline, i.e., $\bar{d}_j = d_j + z/w_j$.)

15.5. Consider a flexible flow shop with a number of stages. The last stage is a bank of two unrelated machines in parallel. At time 400 only one more job has to be processed at the last stage. This job has a due date of 390 and a weight of 5. Both machines are idle at time 400 and if machine 1 would process the job its processing time is 10 and if machine 2 would process the job its processing time is 40. The cost of operating machine 1 is \$4 per unit time and the cost of operating machine 2 is \$1 per unit time. Each MA has all information.
What prices should the two machines submit and which machine ends up processing the job?

15.6. Consider the following instance of $1 \mid\mid \theta_1 \sum w_j C_j + \theta_2 L_{\max}$ with 6 jobs. The objectives are $\sum w_j C_j$ and $L_{\max}$ and the weights are θ_1 and θ_2, respectively.

jobs	1	2	3	4	5	6
d_j	7	7	9	17	33	35
w_j	9	15	12	3	20	24
p_j	9	10	6	3	10	8

(a) Show that the optimal preemptive schedule is nonpreemptive.

(b) Describe a heuristic that gives a good solution with any given combination of weights θ_1 and θ_2.

(c) Describe a branch-and-bound approach for this problem.

15.7. Apply the constraint guided heuristic search procedure to the following instance of $Jm \mid\mid C_{\max}$.

jobs	*machine sequence*	*processing times*			
1	1,2,3,4	$p_{11} = 9$,	$p_{21} = 8$,	$p_{31} = 4$,	$p_{41} = 4$
2	1,2,4,3	$p_{12} = 5$,	$p_{22} = 6$,	$p_{42} = 3$,	$p_{32} = 6$
3	3,1,2,4	$p_{33} = 10$,	$p_{13} = 4$,	$p_{23} = 9$,	$p_{43} = 2$

Compare your result with the result of Exercise 7.3.

15.8. Consider again the instance of $Jm \mid r_j \mid \sum w_j T_j$ described in Example 15.4.2 and in Exercise 15.4. Apply a bidding and pricing heuristic, in which each JA has to negotiate (whenever an operation has been completed) for the operation that comes after the next one (not for the next one as in Example 15.4.2). Note that while the scheduling process goes on, the JA has, when an operation has been completed, already a contract in place for the next operation; the JA then only has to negotiate for the operation after the next one (however, when a job enters the shop for the first time the JA has to negotiate for both the first and the second operation on the job's route).
Compare your results with the results in Example 15.4.2.

15.9. Consider the instance in Example 15.5.3.

(a) Consider the objective $\theta_1 L_{\max} + \theta_2 \sum w_j T_j$ (assume $\theta_1 + \theta_2 = 1$). Find the ranges of θ_1 and θ_2 for which any one of the schedules is optimal.

(b) Consider the objective $\theta_1 L_{\max} + \theta_3 C_{\max}$ (assume $\theta_1 + \theta_3 = 1$). Find the ranges of θ_1 and θ_3 for which any one of the schedules is optimal.

15.10. Consider again the instance in Example 15.5.3. Consider now the objective $\theta_1 L_{\max} + \theta_2 \sum w_j T_j + \theta_3 C_{\max}$ (assume $\theta_1 + \theta_2 + \theta_3 = 1$). Find the ranges of θ_1, θ_2 and θ_3 for which any one of the schedules is optimal.

Exercises (Theory)

15.11. Consider Example 15.2.1 and Exercise 15.1. Design a hybrid decomposition scheme for $1 \mid r_j \mid \sum w_j C_j$, that takes the function $V(t)$ into account as well as the different weights of the jobs.

15.12. Consider Example 15.2.3 and Exercise 15.2. Design a hybrid decomposition scheme for $1 \mid r_j, prmp \mid \sum w_j T_j$ that takes into account $V(t)$, due date clusters and the weights of the jobs.

15.13. Consider the problem $1 \mid r_j \mid \sum T_j$. Design a time-based decomposition method that is based on both the release dates and the due dates of the jobs.

15.14. Consider the market-based procedure described in Section 15.4. In the heuristic that the MA uses, he may wish to have an estimate for the future supply and demand for machine capacity. Describe methods to estimate future supply and demand for the two information infrastructures described in Section 15.4.

15.15. Consider the market-based procedure described in Section 15.4. Describe a goodness of fit measure assuming the MA knows the frequency of the calls for bids that are about to come in as well as the distribution of the corresponding processing times (i.e., the mean and the variance).

15.16. Consider the market-based procedure described in Section 15.4. Suppose a JA receives multiple bids after sending out a call for bids. Design a heuristic for the JA to select a bid.

15.17. Consider a machine environment with two uniform machines in parallel with different speeds. Preemptions are allowed. There are two objectives: the makespan and the total completion time. Prove or disprove that the trade-off curve is decreasing convex.

Comments and References

Beam search and filtered beam search were first applied to scheduling by Ow and Morton (1988). For more details on this method, see the text by Morton and Pentico (1993).

An excellent treatise on decomposition methods is given by Ovacik and Uzsoy (1997). For some more recent results, see the papers by Chand, Traub and Uzsoy (1996, 1997), Szwarc (1998), and Elkamel and Mohindra (1999).

Constraint guided heuristic search (constraint-based programming) is a development that originated among computer scientists and artificial intelligence experts; see Fox and Smith (1984), and Fox (1987). The example of constraint guided heuristic search presented in Section 15.3 is an adaptation of Chapter 8 in the book by Baptiste, Le Pape and Nuijten (2001). For more applications of constraint-based programming to scheduling, see Nuijten (1994), Baptiste, Le Pape, and Nuijten (1995), Nuijten and Aarts (1996) and Cheng and Smith (1997).

Market-Based and Agent-Based procedures have been a topic of significant research interest in the scheduling community as well as in other communities. Some of the early papers in the scheduling field are by Shaw (1987, 1988a, 1989), Ow, Smith, and Howie (1988) and Roundy, Maxwell, Herer, Tayur and Getzler (1991). For examples of auction and bidding protocols, see the paper by Sandholm (1993), Kutanoglu and Wu (1999) and Wellman, Walsh, Wurman and MacKie-Mason (2000). Sabuncuoglu and Toptal (1999a, 1999b) present a clear overview of the concepts in distributed scheduling and bidding algorithms.

A significant amount of research has been done on multi-objective scheduling. Most of it focuses on single machine scheduling, see Chapter 4 and its references. Less research has been done on parallel machine problems with multiple objectives. Eck and Pinedo (1993) consider a nonpreemptive parallel machine scheduling problem with makespan and total completion time as objectives. McCormick and Pinedo (1995) consider a preemptive parallel machine scheduling problem with makespan and total completion time as objectives and obtain a polynomial time algorithm. The book by Deb (2001) focuses on the application of genetic algorithms to multi-objective optimization problems.

The pricing procedure described in the discussion section is due to Luh, Hoitomt, Max and Pattipati (1990), Hoitomt, Luh and Pattipati (1993) and Luh and Hoitomt (1993).

Chapter 16

Modeling and Solving Scheduling Problems in Practice

In Parts I and II a number of stylized and (supposedly) elegant mathematical models are discussed in detail. The deterministic models have led to a number of simple priority rules as well as to many algorithmic techniques and heuristic procedures. The stochastic models have provided some insight into the robustness of the priority rules. The results for the stochastic models have led to the conclusion that the more randomness there is in a system, the less advisable it is to use very sophisticated optimization techniques. Or, equivalently, the more randomness the system is subject to, the simpler the scheduling rules ought to be.

It is not clear how all this knowledge can be applied to scheduling problems in the real world. Such problems tend to differ considerably from the stylized models studied by academic researchers. The first section of this chapter focuses on the differences between the real world problems and the theoretical models. The subsequent four sections deal with examples of scheduling problems that have appeared in industry and for which algorithmic procedures have been developed. The second section illustrates the use of the Profile Fitting heuristic (described in Section 6.2). The third section discusses an application of the LPT heuristic (described in Section 5.1) within an algorithmic framework for flexible flow shops with bypass. The next section illustrates an application of the ATCS

M.L. Pinedo, *Scheduling*, DOI: 10.1007/978-0-387-78935-4_16,

heuristic (described in Section 14.2) within a framework for flexible flow shops without bypass. The fifth section contains an application of constraint guided heuristic search (described in Section 15.3). In the last section a number of modelling issues are discussed.

The applications described in this chapter illustrate the fact that the rules and the techniques introduced and analyzed in Parts I and II can be very useful. However, these rules and techniques usually have to be embedded in a more elaborate framework that deals with all aspects of the problem.

16.1 Scheduling Problems in Practice

Real world scheduling problems are usually very different from the mathematical models studied by researchers in academia. It is not easy to list all the differences between the real world problems and the theoretical models, as every real world problem has its own particular idiosyncrasies. Nevertheless, a number of differences are common and therefore worth mentioning.

(i) Theoretical models usually assume that there are n jobs to be scheduled and that after scheduling these n jobs the problem is solved. In the real world there may be at any point in time n jobs in the system, but new jobs are added continuously. Scheduling the current n jobs has to be done without a perfect knowledge of the near future. Hence, some provisions have to be made in order to be prepared for the unexpected. The dynamic nature may require, for example, that slack times are built into the schedule to accommodate unexpected rush jobs or machine breakdowns.

(ii) Theoretical models usually do not emphasize the *resequencing* problem. In practice the following problem often occurs: there exists a schedule, which was determined earlier based on certain assumptions, and an (unexpected) random event occurs that requires either major or minor modifications in the existing schedule. The rescheduling process, which is sometimes referred to as *reactive* scheduling, may have to satisfy certain constraints. For example, one may wish to keep the changes in the existing schedule at a minimum, even if an optimal schedule cannot be achieved this way. This implies that it is advantageous to construct schedules that are in a sense "robust". That is, resequencing brings about only minor changes in the schedule. The opposite of robust is often referred to as "brittle".

(iii) Machine environments in the real world are often more complicated than the machine environments considered in previous chapters. Processing restrictions and constraints may also be more involved. They may be either machine dependent, job dependent or time dependent.

(iv) In the mathematical models the weights (priorities) of the jobs are assumed to be fixed, i.e., they do not change over time. In practice, the weight of a job often fluctuates over time and it may do so as a random function. A low priority job may become suddenly a high priority job.

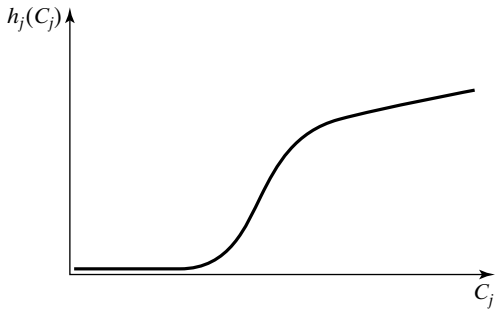


Fig. 16.1 A Penalty Function in Practice

(v) Mathematical models often do not take *preferences* into account. In a model a job either can or cannot be processed on a given machine. That is, whether or not the job can be scheduled on a machine is a $0-1$ proposition. In reality, it often occurs that a job *can* be scheduled on a given machine, but for some reason there is a preference to process it on another one. Scheduling it on the first machine would only be done in case of an emergency and may involve additional costs.

(vi) Most theoretical models do not take machine availability constraints into account; usually it is assumed that machines are available at all times. In practice machines are usually *not* continuously available. There are many reasons why machines may not be in operation. Some of these reasons are based on a deterministic process, others on a random process. The shift pattern of the facility may be such that the facility is not in operation throughout. At times preventive maintenance may be scheduled. The machines may be also subject to a random breakdown and repair process.

(vii) Most penalty functions considered in research are piecewise linear, e.g., the tardiness of a job, the unit penalty, and so on. In practice there usually does exist a committed shipping date or due date. However, the penalty function is usually not piecewise linear. In practice, the penalty function may take, for example, the shape of an "S" (see Figure 16.1). Such a penalty function may be regarded as a function that lies somewhere in between the tardiness function and the unit penalty function.

(viii) Most theoretical research has focused on models with a single objective. In the real world there are usually a number of objectives. Not only are there several objectives, their respective weights may vary over time and may even depend on the particular scheduler in charge. One particular combination of objectives appears to occur very often, especially in the process industry, namely the minimization of the total weighted tardiness and the minimization of the sum of the sequence dependent setup times (especially on bottleneck machines). The minimization of the total weighted tardiness is important since maintaining quality of service is usually an objective that carries weight. The

minimization of the sum of the sequence dependent setup times is important as, to a certain extent, it increases the throughput. When such a combination is the overall objective, the weights given to the two objectives may not be fixed. The weights may depend on the time as well as on the current status of the production environment. If the workload is relatively heavy, then minimization of the sequence dependent setup times is more important; if the workload is relatively light, minimization of the total weighted tardiness is more important.

(ix) The scheduling process is in practice often strongly connected with the assignment of shifts and the scheduling of overtime. Whenever the workload appears to be excessive and due dates appear to be too tight, the decision-maker may have the option to schedule overtime or put in extra shifts in order to meet the committed shipping dates.

(x) The stochastic models studied in the literature usually assume very special processing time distributions. The exponential distribution, for example, is a distribution that has been studied in depth. In reality, processing times are usually not exponentially distributed. Some measurements have shown that processing times may have a density function like the one depicted in Figure 16.2.a. One can think of this density function as the convolution of a deterministic (fixed value) and an Erlang(k, λ). The number of phases of the Erlang(k, λ) tends to be low, say 2 or 3. This density function tends to occur in the case of a manual performance of a given task. That processing times may have such a density function is plausible. One can imagine that there is a certain minimum time that is needed to perform the task to be done. Even the best worker cannot get below this minimum (which is equal to the fixed value). However, there is a certain amount of variability in the processing times that may depend on the person performing the task. The density function may have a tail at the right which represents those processing times during which something went wrong. One can easily show that this density function has an Increasing Completion Rate. Another type of density function that does occur in practice is the one depicted in Figure 16.2.b. The processing time is a fixed value with a very high probability, say .98, and with a very low probability, say .02, it is an additional random time that is exponentially distributed with a very large mean. This type of density function occurs often in automated manufacturing or assembly. If a robot performs a task, the processing time is always fixed (deterministic); however, if by accident something goes wrong the processing time becomes immediately significantly larger.

(xi) Another important aspect of random processing times is correlation. Successive processing times on the same machine tend to be highly positively correlated in practice. In the stochastic models studied in the literature usually all processing times are assumed to be independent draws from (a) given distribution(s). One of the effects of a positive correlation is an increase the variance of the performance measures.

(xii) Processing time distributions may be subject to change due to *learning* or *deterioration*. When the distribution corresponds to a manual operation, then the possibility of learning exists. The human performing the operation may be

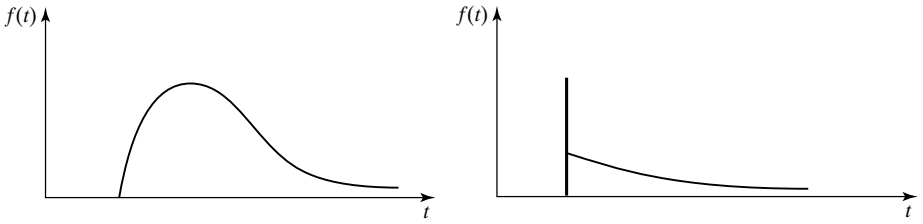


Fig. 16.2 Density functions in practice

able to reduce the average time he needs for performing the operation. If the distribution corresponds to an operation in which a machine is involved, then the aging of the machine may have as effect that the average processing time increases.

In spite of the many differences between the real world and the mathematical models discussed in the previous sections, the general consensus is that the theoretical research done in past years has not been a complete waste of time. It has provided valuable insights into many scheduling problems and these insights have proven to be useful in the development of the algorithmic framework for a large number of real world scheduling systems.

The remaining part of this chapter deals with examples of real world problems for which scheduling theory has turned out to be useful. In practice scheduling problems are often tackled with seemingly crude heuristics. The reason for not applying more sophisticated procedures is usually based on the relatively high frequency of random events. Such events often cause schedule changes during the execution of an existing schedule.

16.2 Cyclic Scheduling of a Flow Line

Consider a number of machines in series with a limited buffer between each two successive machines. When a buffer is full, the machine that feeds that buffer cannot release any more jobs; this machine is then blocked. Serial processing is common in assembly lines for physically large items, such as television sets, copiers and automobiles, whose size makes it difficult to keep many items waiting in front of a machine. Also, the material handling system does not permit a job to bypass another so that each machine serves the jobs according to the First In First Out (FIFO) discipline. These flow lines with blocking are often used in *Flexible Assembly Systems*.

Since different jobs may correspond to different product types, the processing requirements of one job may be different from those of another. Even if jobs are from the same product family, they may have different option packages. The timing of job releases from a machine may be a function of the queue at the machine immediately downstream as well as of the queue at the machine itself.

This machine environment with limited buffers is mathematically equivalent to a system of machines in series with *no* buffers between any machines. Within such a system a buffer space can be modeled as a machine at which all processing times are zero. So any number of machines with limited buffers can be transformed mathematically into a system with a (larger) number of machines in series and *zero* buffers in between the machines. Such a model, with all buffers equal to zero, is mathematically easier to formulate because of the fact that there are no differences in the buffer sizes.

Sequences and schedules used in flow lines with blocking are often periodic or cyclic. That is, a number of different jobs are scheduled in a certain way and this schedule is repeated over and over again. It is not necessarily true that a cyclic schedule has the maximum throughput. Often, an acyclic schedule has the maximum throughput. However, cyclic schedules have a natural advantage because of their simplicity; they are easy to keep track of and impose a certain discipline. In practice, there is often an underlying basic cyclic schedule from which minor deviations are allowed, depending upon current orders.

In order to discuss this class of schedules further, it is useful to define the *Minimum Part Set (MPS)*. Let $\mathcal{N}_k$ denote the number of jobs that correspond to product type k in the overall production target. Suppose there are l different product types. If q is the greatest common divisor of the integers $\mathcal{N}_1, \ldots, \mathcal{N}_l$, then the vector

$$\bar{\mathcal{N}} = \left(\frac{\mathcal{N}_1}{q}, \ldots, \frac{\mathcal{N}_l}{q}\right)$$

represents the smallest set having the same proportions of the different product types as the production target. This set is usually referred to as the Minimum Part Set (MPS). Given the vector $\bar{\mathcal{N}}$, the items in an MPS may, without loss of generality, be regarded as n jobs, where

$$n = \frac{1}{q} \sum_{k=1}^{l} \mathcal{N}_k.$$

The processing time of job j, $j = 1, \ldots, n$, on machine i is p_{ij}. Cyclic schedules are specified by the sequence of the n jobs in the MPS. The fact that some jobs within an MPS may correspond to the same product type and have identical processing requirements does not affect the approach described below.

The *Profile Fitting (PF)* heuristic described in Chapter 6 for $Fm \mid block \mid C_{\max}$ is well suited for the machine environment described above and appears to be a good heuristic for minimizing the cycle time in steady state. The cycle time is the time between the first jobs of two consecutive MPS's entering the system. Minimizing the cycle time is basically equivalent to maximizing the throughput. The following example illustrates the cycle time concept.

Example 16.2.1 (MPS Cycle Time)

Consider an assembly line with four machines and no buffers between machines. There are three different product types and they have to be produced

in equal amounts, i.e.,

$$\bar{\mathcal{N}} = (1, 1, 1).$$

The processing times of the three jobs in the MPS are:

jobs	1	2	3
p_{1j}	0	1	0
p_{2j}	0	0	0
p_{3j}	1	0	1
p_{4j}	1	1	0

The second "machine" (that is, the second row of processing times), with zero processing times for all three jobs, functions as a buffer between the first and third machines. The Gantt charts for this example under the two different sequences are shown in Figure 16.3. Under both sequences steady state is reached during the second cycle. Under sequence $1, 2, 3$ the cycle time is three, whereas under sequence $1, 3, 2$ the cycle time is two. ||

Heuristics that attempt to minimize the makespan in a flow shop with blocking and a finite number of jobs ($Fm \mid block \mid C_{\max}$), also tend to minimize the cycle time of a cyclic schedule for a flow environment with blocking. A variation of the Profile Fitting (PF) heuristic suitable for the problem described above works as follows: one job is selected to go first. The selection of the first job in the MPS sequence may be done arbitrarily or according to some scheme. This first job generates a *profile*. For the time being, it is assumed that the job does not encounter any blocking and proceeds smoothly from one machine to the next (in steady state the first job in an MPS may be blocked by the last job from the previous MPS). The profile is determined by the departure time of this first job, job j_1, from machine i.

$$D_{i,j_1} = \sum_{h=1}^{i} p_{h,j_1}$$

In order to determine which is the most appropriate job to go second, every remaining job in the MPS is tried out. For each candidate job a computation has to be made to determine the times that the machines are idle and the times that the job is blocked at the various machines. The departure times of a candidate job for the second position, say job c, can be computed recursively as follows:

$$\begin{aligned} D_{1,j_2} &= \max(D_{1,j_1} + p_{1c}, D_{2,j_1}) \\ D_{i,j_2} &= \max(D_{i-1,j_2} + p_{ic}, D_{i+1,j_1}), \qquad i = 2, \ldots, m-1 \\ D_{m,j_2} &= D_{m-1,j_2} + p_{mc} \end{aligned}$$

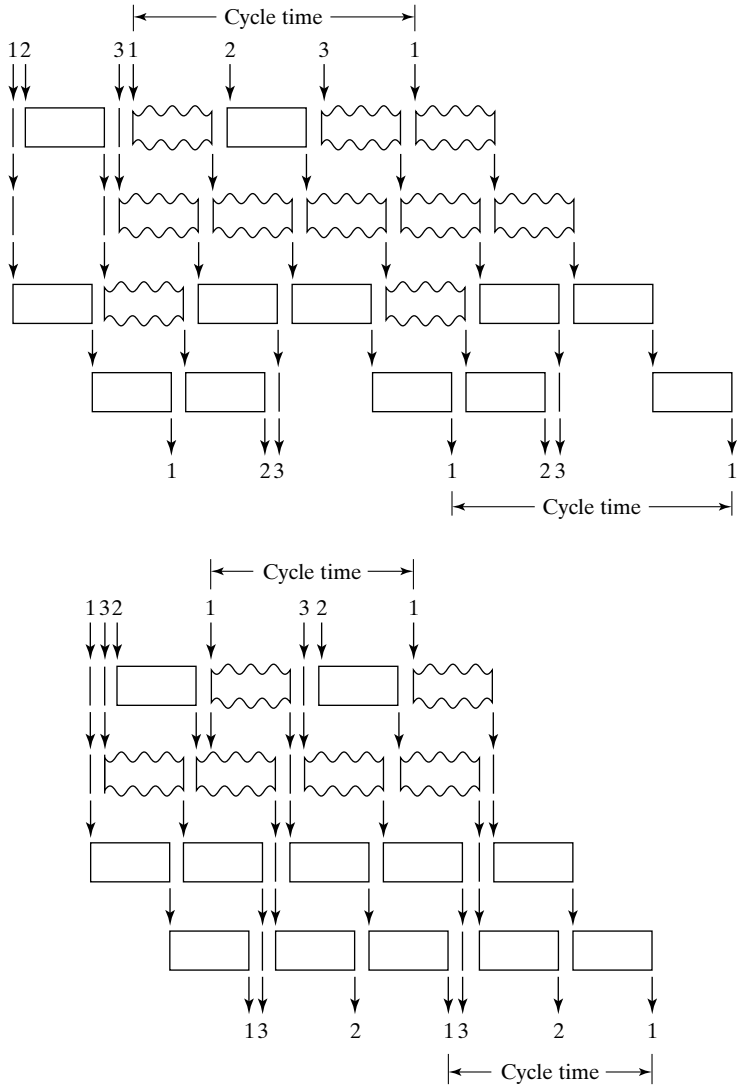


Fig. 16.3 Gantt charts for Example 16.2.1

If candidate c is put into the second position, then the time wasted at machine i, because of either idleness or blocking, is $D_{i,j_2} - D_{i,j_1} - p_{ic}$. The sum of these idle and blocked times over all m machines is computed for candidate c. This procedure is repeated for all remaining jobs in the MPS. The candidate with the smallest total wasted time is then selected for the second position.

After the best fitting job is added to the partial sequence, the new profile (the departure times of this job from all the machines) is computed and the

procedure is repeated. From the remaining jobs in the MPS again the best fitting is selected and so on.

Observe that in Example 16.2.1 after job 1, the PF heuristic would select job 3 to go next, as this would cause only one unit of blocking time (on machine 2) and no idle times. If job 2 were selected to go after job 1, one unit of idle time would be incurred at machine 2 and one unit of blocking on machine 3, resulting in two units of time wasted. So in this example the heuristic would lead to the optimal sequence.

Experiments have shown that the PF heuristic results in schedules that are close to optimal. However, the heuristic can be refined and made to perform even better. In the description presented above the goodness of fit of a particular job was measured by summing all the times wasted on the m machines. Each machine was considered equally important. Suppose one machine is a bottleneck machine, at which more processing has to be done than on any one of the other machines. It is intuitive that lost time on a bottleneck machine is more damaging than lost time on a machine that on the average does not have much processing to do. When measuring the total amount of lost time, it may be appropriate to weight each inactive time period by a factor proportional to the degree of congestion at the particular machine. The higher the degree of congestion at a machine, the larger the weight. One measure for the degree of congestion of a machine is easy to calculate; simply determine the total amount of processing to be done on all jobs in an MPS at the machine in question. In the numerical example presented above the third and the fourth machines are more heavily used than the first and second machines (the second machine was not used at all and basically functioned as a buffer). Time wasted on the third and fourth machines is therefore less desirable than time wasted on the first and second machines. Experiments have shown that such a weighted version of the PF heuristic works exceptionally well.

Example 16.2.2 (Application of Weighted Profile Fitting)

Consider three machines and an MPS of four jobs. There are no buffers between machines 1 and 2 and between machines 2 and 3. The processing times of the four jobs on the three machines are in the following table.

jobs	1	2	3	4
p_{1j}	2	4	2	3
p_{2j}	4	4	0	2
p_{3j}	2	0	2	0

All three machines are actual machines and none are buffers. The workloads on the three machines are not entirely balanced. The workload on machine 1 is 11, on machine 2 it is 10 and on machine 3 it is 4. So, time lost on machine 3 is less detrimental than time lost on the other two machines. If a weighted version of the Profile Fitting heuristic is used, then the weight applied to

the time wasted on machine 3 must be smaller than the weights applied to the time wasted on the other two machines. In this example the weights for machines 1 and 2 are chosen to be 1 while the weight for machine 3 is chosen to be 0.

Assume job 1 is the first job in the MPS. The profile can be determined easily. If job 2 is selected to go second, then there will be no idle times or times blocked on machines 1 and 2; however machine 3 will be idle for 2 time units. If job 3 is selected to go second, machines 1 and 2 are both blocked for two units of time. If job 4 is selected to go second, machine 1 will be blocked for one unit of time. As the weight of machine 1 is 1 and of machine 3 is 0, the weighted PF selects job 2 to go second. It can easily be verified that the weighted PF results in the cycle $1, 2, 4, 3$, with a cycle time of 12. If the unweighted version of the PF heuristic were applied and job 1 again was selected to go first, then job 4 would be selected to go second. The unweighted PF heuristic would result in the sequence $1, 4, 3, 2$ with a cycle time of 14. ||

16.3 Scheduling of a Flexible Flow Line with Limited Buffers and Bypass

Consider a number of stages in series with a number of machines in parallel at each stage. A job, which in this environment often is equivalent to a batch of relatively small identical items such as Printed Circuit Boards (PCB's), needs to be processed at every stage on only one machine. Usually, any of the machines will do, but it may be that not all the machines at a given stage are identical and that a given job has to be processed on a specific machine. A job may not need to be processed at a stage at all; in this case the material handling system that moves jobs from one stage to the next will usually allow a job to bypass a stage and all the jobs residing at that stage (see Figure 16.4). The buffers at each stage may have limited capacity. If this is the case and the buffer is full, then either the material handling system must come to a standstill, or, if there is the option to recirculate, the job bypasses that stage and recirculates. The manufacturing process is repetitive and it is of interest to find a cyclic schedule (similar to the one in the previous section).

The *Flexible Flow Line Loading (FFLL)* algorithm was designed at IBM for the machine environment described above. The algorithm was originally conceived for an assembly system used for the insertion of components in PCB's. The two main objectives of the algorithm are

(i) the maximization of throughput and

(ii) the minimization of WIP.

With the goal of maximizing the throughput an attempt is made to minimize the makespan of a whole day's mix. The FFLL algorithm actually attempts to minimize the cycle time of a Minimum Part Set (MPS). As the amount of

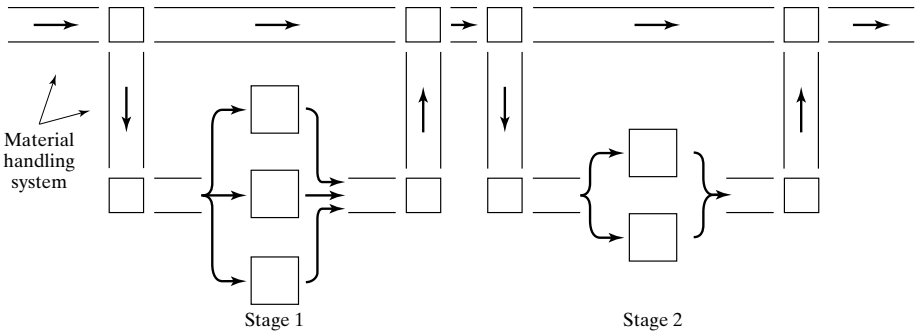


Fig. 16.4 Flexible flow shop with bypass

buffer space is limited, it is recommended to minimize the WIP in order to reduce blocking probabilities. The FFLL algorithm consists of three phases:

Phase 1: machine allocation,

Phase 2: sequencing,

Phase 3: timing of releases.

The machine allocation phase assigns each job to a particular machine in each bank of machines. Machine allocation is done before sequencing and timing because in order to perform the last two phases the workload assigned to each machine must be known. The lowest conceivable maximum workload at a bank would be obtained if all the machines in that bank were given the same workload. In order to obtain balanced, or nearly balanced, workloads over the machines in a bank, the Longest Processing Time first (LPT) heuristic, described in Section 5.1, is used. According to this heuristic all the jobs are, for the time being, assumed to be available at the same time and are allocated to a bank one at a time on the next available machine in decreasing order of their processing time. After the allocation is determined in this way, the items assigned to a machine may be resequenced, which does not alter the workload balance over the machines in a given bank. The output of this phase is merely the allocation of jobs and not the sequencing of the jobs or the timing of the processing.

The sequencing phase determines the order in which the jobs of the MPS are released into the system, which has a strong effect on the cycle time. The FFLL algorithm uses a *Dynamic Balancing* heuristic for sequencing an MPS. This heuristic is based on the intuitive observation that jobs tend to queue up in the buffer of a machine if a large workload is sent to that machine within a short time interval. This occurs when there is an interval in the loading sequence that contains many jobs with large processing times going to the same machine. Let n be the number of jobs in an MPS and m the number of machines in the entire system. Let p_{ij} denote the processing time of job j on machine i. Note that

$p_{ij} = 0$ for all but one machine in a bank. Let

$$W_i = \sum_{j=1}^{n} p_{ij},$$

that is, the workload in an MPS destined for machine i. Let

$$W = \sum_{i=1}^{m} W_i,$$

represent the entire workload of an MPS. For a given sequence, let S_j denote the set of jobs loaded into the system up to and including job j. Let

$$\alpha_{ij} = \sum_{k \in S_j} \frac{p_{ik}}{W_i}.$$

The α_{ij} represent the fraction of the total workload of machine i that has entered the system by the time job j is loaded. Clearly, $0 \leq \alpha_{ij} \leq 1$. The Dynamic Balancing heuristic attempts to keep the $\alpha_{1j}, \alpha_{2j}, \ldots, \alpha_{mj}$ as close to one another as possible, i.e., as close to an ideal target α_j^*, which is defined as follows:

$$\begin{aligned} \alpha_j^* &= \sum_{k \in S_j} \sum_{i=1}^{m} p_{ik} / \sum_{k=1}^{n} \sum_{i=1}^{m} p_{ik} \\ &= \sum_{k \in S_j} p_k / W, \end{aligned}$$

where

$$p_k = \sum_{i=1}^{m} p_{ik},$$

which represents the workload on the entire system due to job k. So α_j^* is the fraction of the total system workload that is released into the system by the time job j is loaded. The cumulative workload on machine i, $\sum_{k \in S_j} p_{ik}$, should be as close as possible to the target $\alpha_j^* W_i$. Now let o_{ij} denote a measure of *overload* at machine i due to job j entering the system

$$o_{ij} = p_{ij} - p_j W_i / W.$$

Clearly, o_{ij} may be negative (in which case there is actually an underload). Now, let

$$O_{ij} = \sum_{k \in S_j} o_{ik} = \Big(\sum_{k \in S_j} p_{ik} \Big) - \alpha_j^* W_i.$$

That is, O_{ij} denotes the cumulative overload (or underload) on machine i due to the jobs in the sequence up to and including job j. To be exactly on target means that machine i is neither overloaded nor underloaded when job j enters the system, i.e., $O_{ij} = 0$. The Dynamic Balancing heuristic now attempts to minimize

$$\sum_{i=1}^{m} \sum_{j=1}^{n} \max(O_{ij}, 0).$$

The procedure is basically a greedy heuristic, that selects from among the remaining items in the MPS the one that minimizes the objective at that point in the sequence.

The timing of releases phase works as follows. From the allocation phase the MPS workloads at each machine are known. The machine with the greatest MPS workload is the bottleneck since the cycle time of the schedule cannot be smaller than the MPS workload at the bottleneck machine. It is easy to determine a timing mechanism that results in a minimum cycle time schedule. First, let all jobs in the MPS enter the system as rapidly as possible. Consider the machines one at a time. At each machine the jobs are processed in the order in which they arrive and processing starts as soon as the job is available. The release times are now modified as follows. Assume that the starting times and the completion times on the bottleneck machine are fixed. First, consider the machines that are positioned before the bottleneck machine (that is, upstream of the bottleneck machine) and delay the processing of all jobs on each one of these machines as much as possible without altering the job sequences. The delays are thus determined by the starting times on the bottleneck machine. Second, consider the machines that are positioned after the bottleneck machine. Process all jobs on these machines as early as possible, again without altering job sequences. These modifications in release times tend to reduce the number of jobs waiting for processing, thus reducing required buffer space.

This three-phase procedure attempts to find the cyclic schedule with minimum cycle time in steady state. If the system starts out empty at some point in time, it may take a few MPS's to reach steady state. Usually, this transient period is very short. The algorithm tends to achieve short cycle times during the transient period as well.

Extensive experiments with the FFLL algorithm indicates that the method is a valuable tool for scheduling flexible flow lines.

Example 16.3.1 (Application of the FFLL Algorithm)

Consider a flexible flow shop with three stages. At stages 1 and 3 there are two machines in parallel. At stage 2, there is a single machine. There are five jobs in an MPS. If p'_{kj} denotes the processing time of job j at stage k, $k = 1, 2, 3$, then

jobs	1	2	3	4	5
p'_{1j}	6	3	1	3	5
p'_{2j}	3	2	1	3	2
p'_{3j}	4	5	6	3	4

The first phase of the FFLL algorithm performs an allocation procedure for stages 1 and 3. Applying the LPT heuristic to the five jobs on the two machines in stage 1 results in an allocation of jobs 1 and 4 to one machine and jobs 5, 2 and 3 to the other machine. Both machines have to perform a total of 9 time units of processing. Applying the LPT heuristic to the five jobs on the two machines in stage 3 results in an allocation of jobs 3 and 5 to one machine and jobs 2, 1 and 4 to the other machine (the LPT heuristic actually does *not* result in an optimal balance in this case). Note that machines 1 and 2 are at stage 1, machine 3 is at stage 2 and machines 4 and 5 are at stage 3. If p_{ij} denotes the processing time of job j on machine i, then

jobs	1	2	3	4	5
p_{1j}	6	0	0	3	0
p_{2j}	0	3	1	0	5
p_{3j}	3	2	1	3	2
p_{4j}	4	5	0	3	0
p_{5j}	0	0	6	0	4

The workload of machine i due to a single MPS, W_i, can now be computed easily. The workload vector W_i is $(9, 9, 11, 12, 10)$ and the entire workload W is 51. The workload imposed on the entire system due to job k, p_k, can also be computed easily. The p_k vector is $(13, 10, 8, 9, 11)$. Based on these numbers all values of the o_{ij} can be computed, for example,

$$o_{11} = 6 - 13 \times 9/51 = +3.71$$
$$o_{21} = 0 - 13 \times 9/51 = -2.29$$
$$o_{31} = 3 - 13 \times 11/51 = +0.20$$
$$o_{41} = 4 - 13 \times 12/51 = +0.94$$
$$o_{51} = 0 - 13 \times 10/51 = -2.55$$

and computing the entire o_{ij} matrix yields

$$\begin{Vmatrix} +3.71 & -1.76 & -1.41 & +1.41 & -1.94 \\ -2.29 & +1.23 & -0.41 & -1.59 & +3.06 \\ +0.20 & -0.16 & -0.73 & +1.06 & -0.37 \\ +0.94 & +2.64 & -1.88 & +0.88 & -2.59 \\ -2.55 & -1.96 & +4.43 & -1.76 & +1.84 \end{Vmatrix}$$

Of course, the solution also depends on the initial job in the MPS. If the initial job is chosen according to the Dynamic Balancing heuristic, then job 4 is the job that goes first. The O_{i4} vector is then $(+1.41, -1.59, +1.06, +0.88, -1.76)$. There are four jobs that then qualify to go second, namely jobs 1, 2, 3 and 5. If job j goes second, then the respective O_{ij} vectors are:

$$O_{i1} = (+5.11, -3.88, +1.26, +1.82, -4.31)$$
$$O_{i2} = (-0.35, -0.36, +0.90, +3.52, -3.72)$$
$$O_{i3} = (+0.00, -2.00, +0.33, -1.00, +2.67)$$
$$O_{i5} = (-0.53, +1.47, +0.69, -1.71, +0.08)$$

It is clear that the Dynamic Balancing heuristic then selects job 5 to go second. Proceeding in the same manner the Dynamic Balancing heuristic selects job 1 to go third and

$$O_{i1} = (+3.18, -0.82, +0.89, -0.77, -2.47).$$

In the same manner it is determined that job 3 goes fourth. Then

$$O_{i3} = (+1.76, -1.23, +0.16, -2.64, +1.96).$$

The final cycle is thus $4, 5, 1, 3, 2$.

Applying the release timing phase on this cycle results initially in the schedule depicted in Figure 16.5. The cycle time of 12 is actually determined by machine 4 (the bottleneck machine) and there is therefore no idle time allowed between the processing of jobs on this machine. It is clear that the processing of jobs 3 and 2 on machine 5 can be postponed by three time units. ||

16.4 Scheduling of a Flexible Flow Line with Unlimited Buffers and Setups

In this section the environment of Example 1.1.1 in Chapter 1 is considered (the paper bag factory). As in the previous section, there are a number of stages in series with a number of machines in parallel at each stage. The machines at a particular stage may be different for various reasons. For example, the more modern machines can accommodate a greater variety of jobs and can operate at a higher speed than the older machines. One stage (sometimes two) constitutes

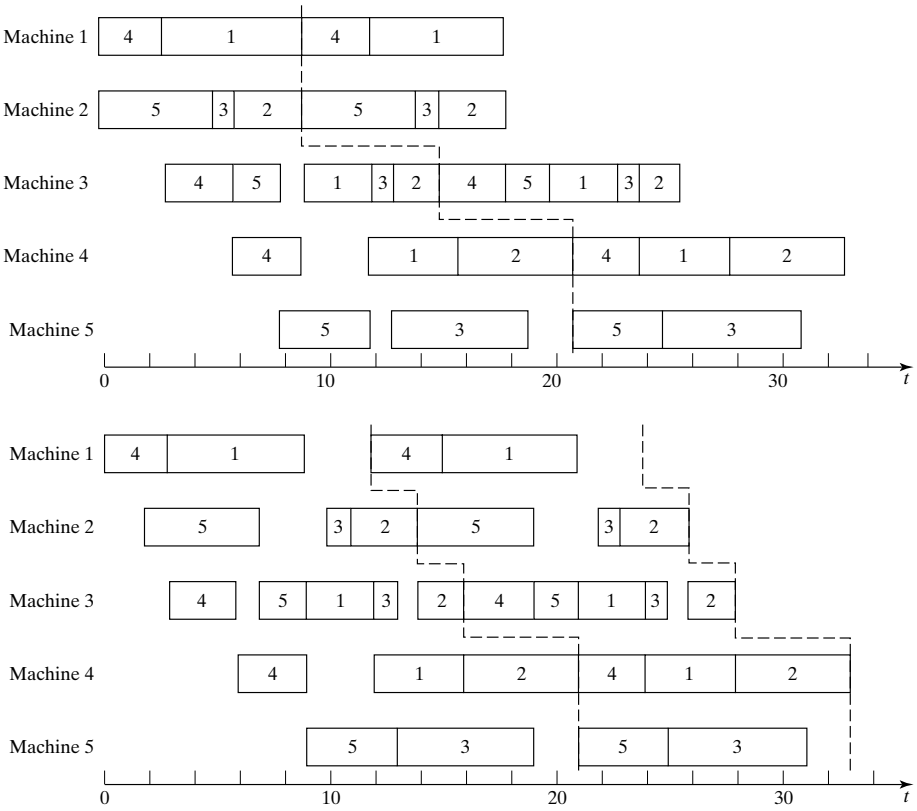


Fig. 16.5 Gantt charts for the FFLL algorithm

a bottleneck. The scheduler is usually aware which stage this is. However, unlike the environments considered in Sections 16.2 and 16.3, this is not a repetitive manufacturing process. The set of jobs to be scheduled at one point in time is usually different from the set of jobs to be scheduled at any other point in time.

The jobs that have to be processed are characterized by their processing times, their due dates (committed shipping dates) as well as by their physical characteristics. The physical characteristics of a job are usually determined by one or more parameters. In the paper bag factory described in Example 1.1.1 the parameters of a job are the size and shape of the bags as well as the colors that are used in the printing process. When one job is completed on a given machine and another has to start, a setup is required. The duration of the setup depends on both jobs; in particular, on the similarities (or differences) between the physical characteristics of the two jobs.

The machine configuration is a flexible flow shop (see Figure 16.6). The scheduler has to keep three basic objectives in mind. The primary objective is to meet the due dates (committed shipping dates). This objective is more

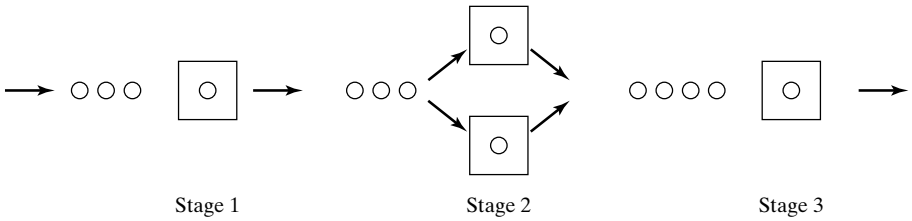


Fig. 16.6 Flexible flow shop with unlimited buffers

or less equivalent to minimizing the total weighted tardiness $\sum w_j T_j$. Another important objective is the maximization of throughput. This objective is somewhat equivalent to the minimization of the sum of setup times, especially with regard to the machines at the bottleneck stage. A third objective is the minimization of the Work-In-Process inventory. The decision making process is not that easy. For example, suppose the decision-maker can schedule a job on a particular machine at a particular time in such a way that the setup time is zero. However, the job's committed shipping date is six weeks from now. His decision depends on the machine utilization, the available storage space and other factors. Clearly, the final schedule is the result of compromises between the three objectives.

The setup time between two consecutive jobs on a machine at any one of the stages is a function of the physical characteristics of the two jobs. As said before, the physical characteristics of the jobs are determined by one or more parameters. For simplicity it is assumed here that only a single parameter is associated with the setup time of job j on machine i, say a_{ij}. If job k follows job j on machine i, then the setup time in between jobs j and k is

$$s_{ijk} = h_i(a_{ij}, a_{ik}).$$

The function h_i may be machine dependent.

The algorithmic framework consists of five phases:

Phase 1: Bottleneck Identification,
Phase 2: Computation of Time Windows at Bottleneck Stage,
Phase 3: Computation of Machine Capacities at Bottleneck Stage,
Phase 4: Scheduling of Bottleneck Stage,
Phase 5: Scheduling of Non-Bottleneck Stages.

The first phase constitutes the bottleneck identification, which works as follows. At least one of the stages has to be the bottleneck. The scheduler usually knows in advance which stage is the bottleneck and schedules this stage first. If two stages are bottlenecks, then the scheduler starts with the bottleneck that is the most downstream. If it is not clear which stage is the bottleneck, then it can be determined from the loading, the number of shifts assigned and the

estimates of the amounts of time that machines are down due to setups. The phenomenon of a moving bottleneck is not taken into account in this phase, as the planning horizon is assumed to be short; the bottleneck therefore does not have sufficient time to move.

The second phase computes the time windows for the jobs at the bottleneck stage. For each job a time window is computed, i.e., a release date and a due date, during which the job should be processed at the bottleneck stage. The due date of the job is computed as follows. The shipping date is the due date at the last stage. We assume that no job, after leaving the bottleneck stage, has a long wait at any one of the subsequent stages. That is, the length of their stay at any one of these stages equals their processing time multiplied by some safety factor. Under this assumption a (local) due date for a job at the bottleneck stage can be obtained. The release date for a job at the bottleneck stage is computed as follows. For each job the status is known. The status of job j, σ_j, may be 0 (the raw material for this job has not yet arrived), 1 (the raw material has arrived but no processing has yet taken place), 2 (the job has gone through the first stage but not yet through the second), or 3 (the job has gone through the second but not yet through the third), and so on. If

$$\sigma_j = l, \qquad l = 0, 1, 2, \ldots, s-1,$$

then the release date of job j at the bottleneck stage b is $r_{bj} = f(l)$, where the function $f(l)$ is decreasing in the status l of job j. A high value of σ_j implies that job j already has received some processing and is expected to have an early release at the bottleneck stage b. The function f has to be determined empirically.

The third phase computes the machine capacities at the bottleneck stage. The capacity of each machine over the planning horizon is computed based on its speed, the number of shifts assigned to it and an estimate of the amount of time spent on setups. If the machines at the bottleneck stage are not identical, then the jobs that go through this stage are partitioned into buckets. For each type of machine it has to be determined which jobs *must* be processed on that type and which jobs *may* be processed on that type. These statistics are gathered for different time frames, e.g., one week ahead, two weeks ahead, etc. Based on these statistics, it can be determined which machine(s) at this stage have the largest loads and are the most critical.

The fourth phase does the scheduling of the jobs at the bottleneck stage. The jobs are scheduled one at a time. Every time a machine is freed, a job is selected to go next. This job selection is based on various factors, namely the setup time (which depends on the job just completed), the due date, and the machine capacity (in case the machines at the bottleneck stage are not identical), and so on. The rule used may be, in its simplest form, equivalent to the ATCS rule described in Section 14.2.

The fifth and last phase does the scheduling of the jobs at the non-bottleneck stages. The sequence in which the jobs go through the bottleneck stage more

or less determines the sequence in which the jobs go through the other stages. However, some minor swaps may still be made in the sequences at the other stages. A swap may be able to reduce the setup times on a machine.

The following example illustrates the algorithm.

Example 16.4.1 (A Flexible Flow Shop with Setups)

Consider a flexible flow shop with three stages. Stages 1 and 3 consist of a single machine, while stage 2 consists of two machines in parallel. The speed of the machine at stage 1, say machine 1, is 4, i.e., $v_1 = 4$. The two machines at stage 2, say machines 2 and 3, both have speed 1, i.e., $v_2 = v_3 = 1$. The speed of the machine at stage 3 is 4, i.e., $v_4 = 4$. There are 6 jobs to be processed. Job j is characterized by a processing requirement p_j and the time it spends on machine i is p_j/v_i. All the relevant data regarding job j are presented in the table below.

jobs	1	2	3	4	5	6
p_j	16	24	20	32	28	22
r_j	1	5	9	7	15	6
d_j	30	35	60	71	27	63
w_j	2	2	1	2	2	1

There are sequence dependent setup times on each one of the four machines. These sequence dependent setup times are determined by *machine settings* a_{ij} which have to be in place for job j when it is processed on machine i. If job k follows job j on machine i, then the setup time

$$s_{ijk} = \mid a_{ik} - a_{ij} \mid .$$

(This setup time structure is a special case of the setup time structure discussed in Section 4.4). The initial setup times on each one of the machines is zero. The machine settings are presented in the table below.

jobs	1	2	3	4	5	6
a_{1j}	4	2	3	1	4	2
a_{2j}	3	4	1	3	1	3
a_{3j}	3	4	1	3	1	3
a_{4j}	2	2	4	1	2	3

Applying the five-phase procedure to minimize the total weighted tardiness results in the following steps:

The bottleneck identification process is easy. It is clear that the second stage is the bottleneck as its throughput rate is only half the throughput rate of the first stage and the third stage.

The second phase requires a computation of time windows at the bottleneck stage. In order to compute these time windows local release times and local due times for stage 2 have to be computed. In order to compute these times, suppose the total time a job spends at either stage 1 or stage 3 is equal to twice its processing time, i.e., the time waiting for a machine at stage 1 or 3 is at most equal to its processing time. These local release dates and due dates are presented in the table below.

job	1	2	3	4	5	6
r_{2j}	9	17	19	23	29	18
d_{2j}	22	23	50	55	13	52

The third phase of the procedure is not applicable here as both machines are assumed to be identical and all jobs can be processed on either one of the two machines.

The fourth phase requires the scheduling of the jobs at the bottleneck stage by the ATCS rule described in Section 14.2. Applying the ATCS rule results in the following sequence: as jobs 1 and 2 are supposedly the first two jobs to be released at stage 2, they start their processing on machines 2 and 3 at their estimated (local) release times 9 and 17, respectively. The estimated completion time of job 1 on machine 2 is $9 + 16 = 25$ and of job 2 on machine 3 is $17 + 24 = 41$. The ATCS rule has to be applied at time 25. There are three jobs to be considered then, namely jobs 3, 4 and 6. The ATCS indices have to be computed for these three jobs. The average processing time of the remaining jobs is approximately 25 and the setup times are between 0 and 3. Assume a K_1 value of 2 and a K_2 value of 0.7. Computing the ATCS indices results in job 4 being started on machine 2 at time 25. The setup time in between the processing of jobs 1 and 4 on machine 2 is zero. So the estimated completion time of job 4 on machine 2 is $25 + 32 = 57$. The next time a machine is freed occurs at the completion of job 2 on machine 3 and the estimated time is 41. Three jobs have to be considered as candidates at time 41, namely jobs 3, 5 and 6. Computing the indices results in the selection of job 5. The setup time in between the processing of jobs 2 and 5 on machine 3 is equal to 3. So the estimated completion time of job 5 on machine 3 is $41 + 3 + 28 = 72$. Machine 2 is the next one to be freed at the estimated time 57. Jobs 3 and 6 remain. The ATCS routine selects job 6, with the larger processing time and the smaller setup time, for processing on machine 2. So job 3 goes on machine 3, starting its processing at 72 and completing it at 92 (see Figure 16.7).

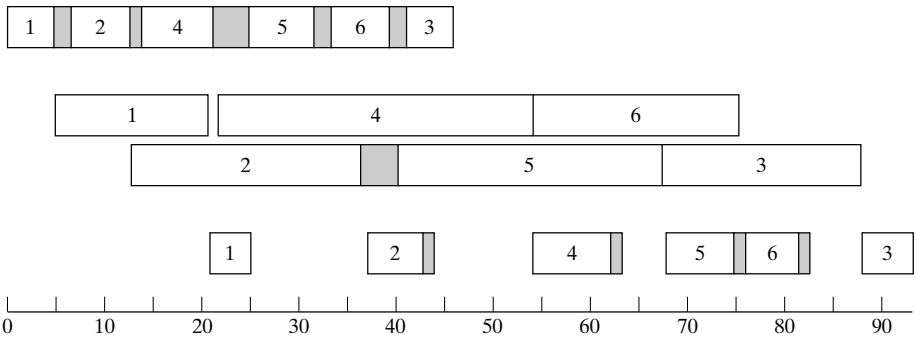


Fig. 16.7 Gantt chart for flexible flow shop with six jobs (Example 16.4.1)

In the fifth and last phase the schedules on machines 1 and 4 are determined. The sequence in which the jobs start their processing at the first stage is basically identical to the sequence in which they start their processing at the second stage. So the jobs are processed on machine 1 in the sequence $1, 2, 4, 5, 6, 3$. After the completion times of the jobs on machine 1 have been determined, the starting times and completion times on machines 2 and 3 have to be recomputed. The sequence in which the jobs start their processing at the third stage is basically identical to the sequence in which the jobs complete their processing at the second stage. After computing the completion times at stage 2, the starting times and completion times on machine 4 are computed. The completion times on machine 4 and the tardinesses are presented in the table below.

jobs	1	2	3	4	5	6
C_j	25	43	93	62	75	81.5
$w_j T_j$	0	16	33	0	96	18.5

The total weighted tardiness is 163.5. The final schedule on the four machines is depicted in Figure 16.7. ||

In more elaborate models the third and fourth phases of the algorithm can be quite complicated. Suppose that at the bottleneck stage there are a number of nonidentical machines in parallel. The jobs may then be partitioned according to their M_j sets. Jobs which have the same M_j set are put into the same *bucket*. If there are two jobs in the same bucket with exactly the same combination of parameters (which implies a zero setup time), the two jobs may be combined to form a single macro job (provided the due dates are not too far apart).

Whenever a machine at the bottleneck stage is freed, various machine statistics may be considered and possibly updated, in order to determine which job goes next. First, it has to be determined what the remaining capacity of the machine is over the planning horizon. Second, it has to be determined how much *must* be produced on this machine over the planning horizon. If this number is less than the remaining capacity, other jobs (which can be processed on this machine as well as on others) may be considered as well, i.e., the pool of jobs from which a job can be selected may be enlarged. The job selection process may be determined by rules that are more elaborate than the ATCS rule. The rules may be based on the following criteria:

(i) the setup time of the job;
(ii) the due date of the job;
(iii) the flexibility of the job (the number of machines it can go on);
(iv) the processing time of the job.

The shorter the setup of a job and the earlier its due date, the higher its priority. The more machines a job can be processed on, the lower its priority. If a machine usually operates at a very high speed (relative to other machines), there is a preference to put larger jobs on this machine. After a job is selected, the statistics with regard to the machine, as well as those with regard to the buckets, have to be updated, i.e., remaining capacity of the machine, contents of the buckets, and so on. In the case of two bottlenecks, the same procedure still applies. The procedure starts with the bottleneck that is most downstream. After scheduling the jobs at this bottleneck, it proceeds with scheduling the jobs at the second bottleneck.

16.5 Scheduling a Bank of Parallel Machines with Jobs having Release Dates and Due Dates

This section focuses on the gate assignments at an airport, as described in Example 1.1.3 in Chapter 1 and Example 2.2.3 in Chapter 2.

Consider m machines in parallel and n jobs. Job j has a processing time p_j, a release date r_j and a due date d_j. Job j may be processed on any machine that belongs to set M_j. The goal is to find a feasible assignment of jobs to machines.

At times, one can make in the problem formulation a distinction between hard constraints and soft constraints. A hard constraint has to be satisfied at all cost. A soft constraint basically refers to a preference. For example, it may be the case that it is preferable to process job j on a machine that belongs to a set M_j, but if necessary, job j can be processed on anyone of the m machines.

There may be additional constraints as well. For example, job j may be processed on machine $i \in M_j$ only if it is started after r_{ij} and completed before d_{ij}.

The entire algorithmic procedure consists of three phases.

Phase 1: Constraint guided heuristic search;

Phase 2: Constraint relaxation;

Phase 3: Assignment adjustment.

The constraint guided heuristic search phase uses a generalization of the procedure described in Section 15.3 to search for a feasible schedule or an assignment that meets all due dates. More often than not, this phase does not yield immediately a feasible schedule with an assignment of all jobs. It often occurs that after a first pass through Phase 1 only a subset of the jobs are assigned and with this partial assignment there does not exist a feasible schedule for the remaining jobs. If this is indeed the case, then the procedure has to go through the second phase.

The constraint relaxation phase attempts to find a feasible schedule by relaxing some of the constraints. Constraint relaxation can be done in various ways. First, due dates of jobs that could not be assigned in the first phase may be postponed and/or the processing times of these jobs may be shortened. Second, in the first pass a particular assignment of a job to a machine may have been labelled *impossible*, whereas in reality this assignment would have been considered only *less preferable* because the machine in question was not *perfectly* suited for the particular job (i.e., a soft constraint was being violated). However, such an assignment can be made possible in a second pass through Phase 1 by relaxing the soft constraint. Third, one of the additional constraints may be relaxed. Constraint relaxation usually yields a complete schedule, with all jobs assigned to machines.

The assignment adjustment phase attempts to improve the schedule via pairwise swaps. Even though the main goal is to obtain a feasible schedule, other objectives may still play a role. Such objectives can include the balancing of the workloads over the various machines and the maximization of the idle times between the processing of consecutive jobs on the various machines (this last objective may be important in order to be prepared for potential fluctuations in the processing times).

Note that the second phase in this framework, the constraint relaxation phase, is different from the backtracking that may occur in the execution of a constraint program. When a program backtracks, it annuls some segments of the partial schedule it had already created; however, it does not make any changes in the formulation or the statement of the scheduling problem. Constraint relaxation, on the other hand, refers to changes that are made in the actual formulation of the problem.

The approach described in this section has at times also been referred to as the *reformulative* approach. The idea is simple: the model has to be reformulated until a feasible schedule is found.

16.6 Discussion

In all four cases described the scheduling procedures are based on *heuristics* and not on procedures that aim for an *optimal* solution. There are several reasons for this. First, the model is usually only a crude representation of the actual problem; so an optimal solution for the model may not actually correspond to the best solution for the real problem. Second, almost all scheduling problems in the real world are strongly NP-hard; it would take a very long time to find an optimal solution on a PC, or even on a workstation. Third, in practice the scheduling environment is usually subject to a significant amount of randomness; it does not pay therefore to spend an enormous amount of computation time to find a supposedly optimal solution when within a couple of hours, because of some random event, either the machine environment or the job set changes.

The solution procedure in each one of the last three sections consisted of a number of phases. There are many advantages in keeping the procedure segmented or modular. The programming effort can be organized more easily and debugging is made easier. Also, if there are changes in the environment and the scheduling procedures have to be changed, a modular design facilitates the reprogramming effort considerably.

Exercises (Computational)

16.1. Consider in the model of Section 16.2 an MPS of 6 jobs.

(a) Show that when all jobs are different the number of different cyclic schedules is 5!

(b) Compute the number of different cyclic schedules when two jobs are the same (i.e., there are five different job types among the 6 jobs).

16.2. Consider the model discussed in Section 16.2 with 4 machines and an MPS of 4 jobs.

jobs	1	2	3	4
p_{1j}	3	2	3	4
p_{2j}	1	5	2	3
p_{3j}	2	4	0	1
p_{4j}	4	1	3	3

(a) Apply the unweighted PF heuristic to find a cyclic schedule. Choose job 1 as the initial job and compute the cycle time.

(b) Apply again the unweighted PF heuristic. Choose job 2 as the initial job and compute the cycle time.

(c) Find the optimal schedule.

16.3. Consider the same problem as in the previous exercise.

(a) Apply a weighted PF heuristic to find a cyclic schedule. Choose the weights associated with machines $1, 2, 3, 4$ as $2, 2, 1, 2$, respectively. Select job 1 as the initial job.

(b) Apply again a weighted PF heuristic but now with weights $3, 3, 1, 3$. Select again job 1 as the initial job.

(c) Repeat again (a) and (b) but select job 2 as the initial job.

(d) Compare the impact of the weights on the heuristic's performance with the effect of the selection of the first job.

16.4. Consider again the model discussed in Section 16.2. Assume that a system is in steady state if each machine is in steady state. That is, at each machine the departure of job j in one MPS occurs exactly the cycle time before the departure of job j in the next MPS. Construct an example with 3 machines and an MPS of a single job that takes more than 100 MPS's to reach steady state, assuming the system starts out empty.

16.5. Consider the application of the FFLL algorithm in Example 16.3.1. Instead of letting the Dynamic Balancing heuristic minimize

$$\sum_{i=1}^{m} \sum_{j=1}^{n} \max(O_{ij}, 0),$$

let it minimize

$$\sum_{i=1}^{m} \sum_{j=1}^{n} \mid O_{ij} \mid .$$

Redo Example 16.3.1 and compare the performances of the two Dynamic Balancing heuristics.

16.6. Consider the application of the FFLL algorithm to the instance in Example 16.3.1. Instead of applying LPT in the first phase of the algorithm, find the optimal allocation of jobs to machines (which leads to a perfect balance of machines 4 and 5). Proceed with the sequencing phase and release timing phase based on this new allocation.

16.7. Consider again the instance in Example 16.3.1.

(a) Compute in Example 16.3.1 the number of jobs waiting for processing at each stage as a function of time and determine the required buffer size at each stage.

(b) Consider the application of the FFLL algorithm to the instance in Example 16.3.1 with the machine allocation as prescribed in Exercise 16.6.

Compute the number of jobs waiting for processing at each stage as a function of time and determine the required buffer size.
(Note that with regard to the machines before the bottleneck, the release timing phase in a sense postpones the release of each job as much as possible and tends to reduce the number of jobs waiting for processing at each stage.)

16.8. Consider Example 16.4.1. In the second phase of the procedure the time windows at the bottleneck stage have to be computed. In the example, job j is estimated to spend twice its processing time at a non-bottleneck stage.

(a) Repeat the procedure and estimate job j's sojourn time as 1.5 times its processing time.
(b) Repeat the procedure and estimate job j's sojourn time as 3 times its processing time.
(c) Compare the results obtained.

16.9. Consider again the instance in Example 16.4.1. Instead of applying the ATCS rule in the fourth phase, select every time a machine is freed a job with the shortest setup time. Compare again the sum of the weighted tardinesses under the two rules.

16.10. Consider again the instance in Example 16.4.1. Instead of applying the ATCS rule in the fourth phase, select every time a machine is freed the job with the earliest estimated local due date. Compare the sum of the weighted tardinesses under the two rules.

Exercises (Theory)

16.11. Consider a distribution which is a convolution of a deterministic (i.e., a fixed value) D and an Erlang(k, λ) with parameters k and λ (see Figure 16.2.a). Determine its coefficient of variation as a function of D, k, and λ.

16.12. Consider a random variable X with the following distribution:

$$P(X = D) = p$$

and

$$P(X = D + Y) = 1 - p,$$

where D is a fixed value and the random variable Y is exponentially distributed with rate λ (see Figure 16.2.b). Determine the coefficient of variation as a function of p, D and λ. Show that this distribution is neither ICR nor DCR.

16.13. Consider a single machine and n jobs. The processing time of job j is a random variable from distribution F. Compare the following two scenarios. In the first scenario the n processing times are i.i.d. from distribution F and in the second scenario the processing times of the n jobs are all equal to the

same random variable X from distribution F. Show that the expected total completion time is the same in the two scenarios. Show that the variance of the total completion time is larger in the second scenario.

16.14. Show that the cyclic scheduling problem described in Section 16.2 with two machines and zero intermediate buffer is equivalent to the TSP. Describe the structure of the distance matrix. Determine whether the structure fits the TSP framework considered in Section 4.4.

16.15. Consider the model in Section 16.2. Construct an example where no cyclic schedule of a single MPS maximizes the long term average throughput rate. That is, in order to maximize the long term average throughput rate one has to find a cyclic schedule of k MPS's, $k \geq 2$.

16.16. The selection of the first job in an MPS in the model of Section 16.2 can be done by choosing the job with the largest total amount of processing. List the advantages and disadvantages of such a selection.

16.17. Consider an MPS with n jobs which includes jobs j and k where $p_{ij} = p_{ik} = 1$ for all i. Show that there exists an optimal cyclic schedule with jobs j and k adjacent.

16.18. Consider an MPS with n jobs. For any pair of jobs j and k either $p_{ij} \geq p_{ik}$ for all i or $p_{ij} \leq p_{ik}$ for all i. Describe the structure of the optimal cyclic schedule.

16.19. Consider the FFLL algorithm. Determine whether or not the longest makespan that could be obtained in Phase 2 always would end up to be equal to the cycle time of the cyclic schedule generated by the algorithm.

16.20. Describe an approach for the model in Section 16.4 with two stages being the bottleneck. List the advantages and disadvantages of scheduling the upstream bottleneck first. Do the same with regard to the downstream bottleneck.

16.21. Consider the scheduling problem discussed in Section 16.4. Design alternate ways for computing the time windows, i.e., determining the local release dates and due dates (estimates of the amount of time downstream and the amount of time upstream, and so on). Explain how to take setup times into account.

16.22. Consider the scheduling problem discussed in Section 16.4. If a nonlinear function of the congestion is used for estimating the transit time through one or more stages, should the function be increasing concave or increasing convex? How does the amount of randomness in the system affect the shape of the function?

16.23. Consider the scheduling problem considered in Section 16.5. Design a composite dispatching rule for this problem (*Hint:* Try to integrate the LFJ

or LFM rule with the ATC rule). Determine the number of scaling parameters and determine the factors or statistics necessary to characterize scheduling instances.

Comments and References

The differences between real world scheduling problems and theoretical models has been the subject of a number of papers, see, for example, Mckay, Safayeni and Buzacott (1988), Rickel (1988), Oliff (1988), Buxey (1989), Baumgartner and Wah (1991) and Van Dyke Parunak (1991).

The cyclic scheduling of the flow line with limited buffers is analyzed by Pinedo, Wolf and McCormick (1986) and McCormick, Pinedo, Shenker and Wolf (1989). Its transient analysis is discussed in McCormick, Pinedo, Shenker and Wolf (1990). For more results on cyclic scheduling, see Matsuo (1990) and Roundy (1992).

The scheduling problem of the flexible flow line with limited buffers and bypass is studied by Wittrock (1985, 1988).

The flexible flow line with unlimited buffers and setups is considered by Adler, Fraiman, Kobacker, Pinedo, Plotnicoff and Wu (1993).

The scheduling problem of a bank of parallel machines with release dates and due dates is considered by Brazile and Swigger (1988, 1991).

Of course, scheduling problems in many other types of industries have been discussed in the literature also. For scheduling problems and solutions in the micro-electronics industry, see, for example, Bitran and Tirupati (1988), Wein (1988), Uzsoy, Lee and Martin-Vega (1992a, 1992b), Lee, Uzsoy and Martin-Vega (1992), Lee, Martin-Vega, Uzsoy and Hinchman (1993). For scheduling problems and solutions in the automotive industry see, for example, Burns and Daganzo (1987), Yano and Bolat (1989), Bean, Birge, Mittenthal and Noon (1991), and Akturk and Gorgulu (1999).

Chapter 17

Design and Implementation of Scheduling Systems: Basic Concepts

Analyzing a scheduling problem and developing a procedure for dealing with it on a regular basis is, in the real world, only part of the story. The procedure has to be embedded in a system that enables the decision-maker to actually use it. The system has to be integrated into the information system of the enterprise, which can be a formidable task. This chapter focuses on system design and implementation issues.

The first section presents an overview of the infrastructure of the information systems and the architecture of the decision support systems in an enterprise. We focus on scheduling systems in particular. The second section covers database, object base, and knowledge-base issues. The third section describes the modules that generate the schedules, while the fourth section discusses issues concerning user interfaces and interactive optimization. The fifth section describes the advantages and disadvantages of generic systems and application-specific systems, while the last section discusses implementation and maintenance issues.

It is, of course, impossible to cover everything concerning the topics mentioned above. Many books have been written on each of these topics. This chapter focuses only on some of the more important issues concerning the design, development and implementation of scheduling systems.

M.L. Pinedo, *Scheduling*, DOI: 10.1007/978-0-387-78935-4_17,

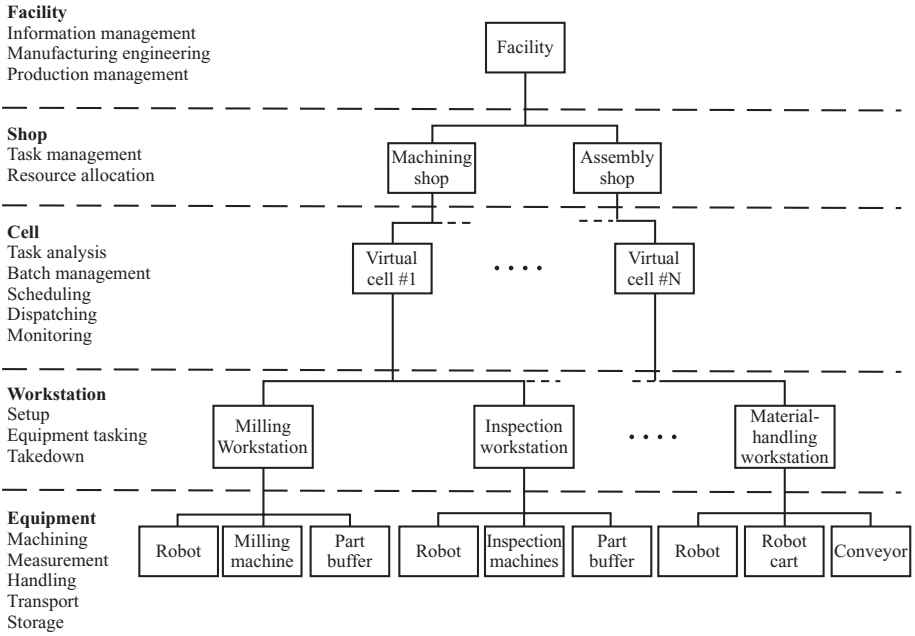


Fig. 17.1 Simplified version of reference model

17.1 Systems Architecture

To visualize the information flows throughout an enterprise one often uses a *reference* model, which depicts all the relevant information flows. An example of a simplified reference model is presented in Figure 17.1

Nowadays, there are companies that specialize in the design and development of software that can serve as a backbone for an enterprise-wide information system. Each decision support system, at any level of the company, can then be linked to such a backbone. Such a framework facilitates the connectivity of all the modules in an enterprise-wide information system. A company that is very active in this type of software development is SAP, which has its headquarters in Walldorf (Germany).

As described in Section 1.2, a scheduling system usually has to interact with a number of different systems in an organization. It may receive information from a higher level system that provides guidelines for the actions to be taken with regard to long term plans, medium term plans, short term schedules, workforce allocations, preventive maintenance, and so on. It may interact with a Material Requirements Planning (MRP) system in order to determine proper release dates for the jobs. A system may also interact with a shop floor control system that provides up-to-date information concerning availability of machines, statuses of jobs, and so on (see Figures 1.1 and 17.1).

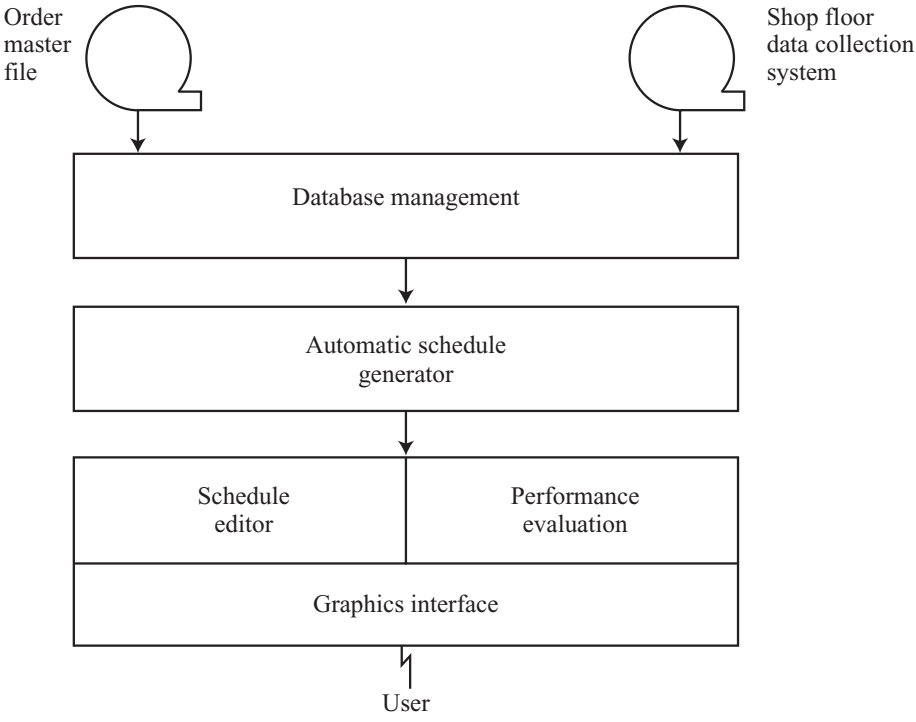


Fig. 17.2 Configuration of a scheduling system

A scheduling system typically consists of a number of different modules (see Figure 17.2). The various types of modules can be categorized as follows:

(i) database, object base, and knowledge-base modules,
(ii) modules that generate the schedules and
(iii) user interface modules.

These modules play a crucial role in the functionality of the system. Significant effort is required to make a factory's database suitable for input to a scheduling system. Making a database accurate, consistent, and complete often involves the design of a series of tests that the data must pass before it can be used. A database management module may also be able to manipulate the data, perform various forms of statistical analysis and allow the decision-maker, through some user interface, to see the data in graphical form. Some systems have a knowledge-base that is specifically designed for scheduling purposes. A knowledge-base may contain, in one format or another, a list of rules that have to be followed in specific situations and maybe also a list of objects representing orders, jobs, and resources. A knowledge-base may at times also take the form of a constraint store that contains the constraints that have to be satis-

fied by the schedules. However, few systems have a separate knowledge-base; a knowledge-base is often embedded in the module that generates the schedules.

The module that generates the schedules typically contains a suitable model with objective functions, constraints and rules, as well as heuristics and algorithms.

User interface modules are important, especially in the implementation process. Without an excellent user interface there is a good chance that, regardless of its capabilities, the system will be too unwieldy to use. User interfaces often take the form of an electronic Gantt chart with tables and graphs that enable a user to edit a schedule generated by the system and take last minute information into account (see Figure 17.2). After a user has adjusted the schedule manually, he is usually able to see the impact his changes have on the performance measures, compare different solutions, and perform "what if" analyses.

17.2 Databases, Object Bases, and Knowledge-Bases

The database management subsystem may be either a custom-made or a commercial system. Various commercial database systems that are available on the market have proven to be useful for scheduling systems. They are typically relational databases incorporating Structured Query Language (SQL). Examples of such database management systems are Oracle and Sybase.

Whether a database management subsystem is custom made or commercial, it needs a number of basic functions, that include multiple editing, sorting and searching routines. Before generating a schedule, a decision-maker may want to see certain segments of an order masterfile and collect some statistics with regard to the orders and the related jobs. Actually, at times, he may not want to feed all the jobs into the scheduling routines, but only a subset.

Within the database a distinction can be made between *static* and *dynamic* data. Static data include job data and machine or resource data that do *not* depend on the schedule. Some job data may be specified in the customer's order form, such as the ordered quantity (which is proportional to the processing times of all the operations associated with the job), the committed shipping date (the due date), the time at which all necessary material is available (the release date) and possibly some processing (precedence) constraints. The priorities (weights) of the jobs are also static data as they do not depend on the schedule. Having different weights for different jobs is usually a necessity, but determining their values is not that easy. In practice, it is seldom necessary to have more than three priority classes; the weights are then, for example, 1, 2 and 4. The three priority classes are sometimes described as "hot", "very hot" and "hottest" dependent upon the level of manager pushing the job. These weights actually have to be entered manually by the decision-maker into the database. To determine the priority level, the person who enters the weight may use his own judgment, or may use a formula that takes into account certain data from the information system (for instance, total annual sales to the

customer or some other measure of customer criticality). The weight of a job may also change from one day to another; a job that is not urgent today, may be urgent tomorrow. The decision-maker may have to go into the file and change the weight of the job before generating a new schedule. Static machine data include machine speeds, scheduled maintenance times, and so on. There may also be static data that are both job and machine dependent, e.g., the setup time between jobs j and k assuming the setup takes place on machine i.

The dynamic data consists of all the data that are dependent upon the schedule: the starting times and completion times of the jobs, the idle times of the machines, the times that a machine is undergoing setups, the sequences in which the jobs are processed on the machines, the number of jobs that are late, the tardinesses of the late jobs, and so on.

The following example illustrates some of these notions.

Example 17.2.1 (Order Master File in a Paper Bag Factory)

Consider the paper bag factory described in Example 1.1.1. The order master file may contain the following data:

ORDER	CUSTOMER	CMT	FC	GS	FBL	QTY	DDT	PRDT
DVN01410	CHEHEBAR CO		16.0	5.0	29.0	55.0	05/25	05/24
DVN01411	CHEHEBAR CO		16.0	4.0	29.0	20.0	05/25	05/25
DVN01412	CHEHEBAR CO		16.0	4.0	29.0	35.0	06/01	
DXY01712	LANSBERG LTD	PR	14.0	3.0	21.0	7.5	05/28	05/23
DXY01713	LANSBERG LTD		14.0	3.0	21.0	45.0	05/28	05/23
DXY01714	LANSBERG LTD		16.0	3.0	21.0	50.0	06/07	
EOR01310	DERMAN INC	HLD	16.0	3.0	23.0	27.5	06/15	

Each order is characterized by an 8 digit alphanumeric order number. A customer may place a number of different orders, each one representing a different type of bag. A bag is characterized by three physical parameters, the so-called face width (FC), the gusset (GS) and the finished bag length (FBL), which correspond to machine settings for a bag of that size. The quantities of bags ordered (QTY) are in multiples of a thousand, e.g., the first order represents 55,000 bags. The month and day of the committed shipping date are specified in the DDT column. The month and day of the completion date of the order are specified in the PRDT column; the days specified in this column can be either actual completion dates or planned completion dates. The comments (CMT) column is often empty. If a customer calls and puts an order an hold, then HLD is entered in this column and the scheduler knows that this order should not yet be scheduled. If an order has a high priority, then PR is entered in this column. The weights will be a function of these

entries, i.e., a job on hold has a low weight, a priority job has a high weight and the default value corresponds to an average weight. ||

Setup times may be regarded either as static or as dynamic data, depending on how they are generated. Setup times may be stored in a table so that whenever a particular setup time needs to be known, the necessary table entry is retrieved. However, this method is not very efficient if the set is very large and if relatively few table look-ups are required. The size of the matrix is n^2 and all entries of the matrix have to be computed beforehand, which may require considerable CPU time as well as memory. An alternative way to compute and retrieve setup times, that is more efficient in terms of storage space and may be more efficient in terms of computation time, is the following. A number of parameters, say

$$a_{ij}^{(1)}, \ldots, a_{ij}^{(l)},$$

may be associated with job j and machine i. These parameters are static data and may be regarded as given machine settings necessary to process job j on machine i. The setup time between jobs j and k on machine i, s_{ijk}, is a known function of the $2l$ parameters

$$a_{ij}^{(1)}, \ldots, a_{ij}^{(l)}, \quad a_{ik}^{(1)}, \ldots, a_{ik}^{(l)}.$$

The setup time usually is a function of the differences in machine settings for jobs j and k and is determined by production standards.

Example 17.2.2 (Sequence Dependent Setup Times)

Assume that, in order to start a job on machine i, three machine settings have to be fixed (for example, the face, the gusset and the finished bag length of a bag in the factory of Example 1.1.1). So the total setup time s_{ijk} depends on the time it takes to perform these three changeovers and is a function of six parameters, i.e.,

$$a_{ij}^{(1)}, a_{ij}^{(2)}, a_{ij}^{(3)}, \quad a_{ik}^{(1)}, a_{ik}^{(2)}, a_{ik}^{(3)}.$$

If the three changeovers have to be done sequentially, then the total setup time is

$$s_{ijk} = h_i^{(1)}(a_{ij}^{(1)}, a_{ik}^{(1)}) \; + \; h_i^{(2)}(a_{ij}^{(2)}, a_{ik}^{(2)}) \; + \; h_i^{(3)}(a_{ij}^{(3)}, a_{ik}^{(3)}).$$

If the three changeovers can be done in parallel, then the total setup time is

$$s_{ijk} = \max\Big(h_i^{(1)}(a_{ij}^{(1)}, a_{ik}^{(1)}),\; h_i^{(2)}(a_{ij}^{(2)}, a_{ik}^{(2)}),\; h_i^{(3)}(a_{ij}^{(3)}, a_{ik}^{(3)})\Big).$$

Of course, there may be situations where some of the changeovers can be done in parallel while others have to be done in series. ||

If the setup times are computed this way, they may be considered dynamic data. The total time needed for computing setup times in this manner depends on the type of algorithm. If a dispatching rule is used to determine a good schedule, this method, based on (static) job parameters, is usually more efficient than the table look-up method mentioned earlier. However, if some kind of local search routine is used, the table look-up method will become more time efficient. The decision on which method to use depends on the relative importance of memory versus CPU time.

The calendar function is often a part of the database system. It contains information with regard to holidays, number of shifts available, scheduled machine maintenance, and so on. Calendar data are sometimes static, e.g., fixed holidays, and sometimes dynamic, e.g., preventive maintenance shutdowns.

Some of the more advanced scheduling systems may rely on an object base in addition to (or instead of) a database. One of the main functions of the object base is to store the definitions of all object types, i.e., it functions as an object library and instantiates the objects when needed. In a conventional relational database, a data type can be defined as a schema of data; for example, a data type "job" can be defined as in Figure 17.3.a and an instance can be as in Figure 17.3.b. Object types and corresponding instances can be defined in the same way. For example, an object type "job" can be defined and corresponding job instances can be created. All the job instances have then the same type of attributes.

There are two crucial relationships between object types, namely, the "is-a" relationship and the "has-a" relationship. An is-a relationship indicates a generalization and the two object types have similar characteristics. The two object types are sometimes referred to as a subtype and a supertype. For example, a "machine" object type may be a special case of a "resource" object type and a "tool" object type may be another special case of a resource object type. A "has-a" relationship is an aggregation relationship; one object type contains a number of other object types. A "workcenter" object may consist of several machine objects and a "plant" object may comprise a number of workcenter objects. A "routing table" object may consist of job objects as well as of machine objects.

Object types related by is-a or has-a relationships have similar characteristics with regard to their attributes. In other words, all the attributes of a supertype object are used by the corresponding subtypes. For example, a machine object has all the attributes of a resource object and it may also have some additional attributes. This is often referred to as inheritance. A hierarchical structure that comprises all object types can be constructed. Objects can be retrieved through commands that are similar to SQL commands in relational databases.

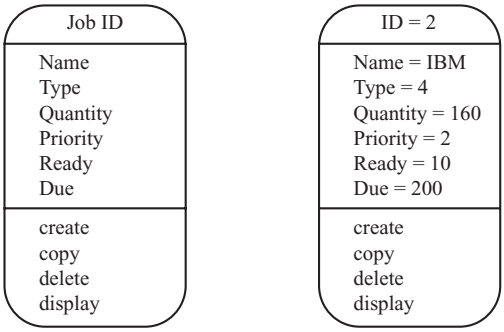
While virtually every scheduling system relies on a database or an object base, not that many systems have a module that serves specifically as a knowledge-base. However, knowledge-bases, at times also referred to as constraint stores, may become more and more important in the future.

ID	Name	Type	Quantity	Priority	Ready	Due

(a) Job data type

2	IBM	4	160	2	10	200

(b) Job instance



(c) Job object type and a job object

Fig. 17.3 Job data type, job instance, and job object type

The overall architecture of a system, in particular the module that generates the schedules, influences the design of a knowledge-base. The most important aspect of a knowledge-base is the knowledge *representation.* One form of knowledge representation is through *rules.* There are several formats for stating rules. A common format is through an *IF-THEN* statement. That is, *IF* a given condition holds, *THEN* a specific action has to be taken.

Example 17.2.3 (IF-THEN Rule for Parallel Machines)

Consider a setting with machines in parallel. The machines have different speeds; some are fast and others are slow. The jobs are subject to sequence dependent setup times that are independent of the machines, i.e., setups take the same amount of time on a fast machine as they take on a slow machine.

Because of the setup times it is advisable to assign the longer jobs to the faster machines while keeping the shorter jobs on the slower machines. One could establish a threshold rule that assigns the longer jobs to the faster machine as follows.

IF a job's processing time is longer than a given value,
THEN the job may be assigned to a fast machine.

It is easy to code such a rule in a programming language such as C++. ||

Another format for stating rules is through *predicate logic* that is based on propositional calculus. An appropriate programming language for the implementation of such rules is Prolog.

Example 17.2.4 (Logic Rule for Parallel Machines)

Consider the rule in the previous example. A Prolog version of this rule may be:

$$\text{MACHINEOK(M,L)} : - \text{long\_job(L)}\,, \text{fast\_machines(F)}\,, \text{member(M,F)}.$$

The M refers to a specific machine, the L to a long job, and the F to a list of all fast machines. The ": −" may be read as "if"and the "," may be read as "and". A translation of the rule would be: machine M is suitable for job L if L is a long job, if set F is the set of fast machines and if machine M is a member of F. ||

As stated before, the design of the module that generates the schedules affects the design of the knowledge-base and vice versa. This is discussed in more detail in the next section.

17.3 Modules for Generating Schedules

Current scheduling techniques are an amalgamation of several schools of thought that have been converging in recent years. One school of thought, predominantly followed by industrial engineers and operations researchers, is sometimes referred to as the *algorithmic* or the *optimization* approach. A second school of thought, that is often followed by computer scientists and artificial intelligence experts, include the *knowledge-based* and the *constraint programming* approaches. Recently, the two schools of thought have been converging and the differences have become blurred. Some hybrid systems combine a knowledge base with fairly sophisticated heuristics; other systems have one segment of the procedure designed according to the optimization approach and another segment according to the constraint programming approach.

Example 17.3.1 (Architecture of a Scheduling System)

A hybrid scheduling system has been designed for a particular semiconductor wafer fabrication unit as follows. The system consists of two levels. The higher level operates according to a knowledge-based approach. The lower level is based on an optimization approach; it consists of a library of algorithms.

The higher level performs the first phase of the scheduling process. At this level, the current status of the environment is analyzed. This analysis takes into consideration due date tightness, bottlenecks, and so on. The rules

embedded in this higher level determine then for each situation the type of algorithm to be used at the lower level. ||

The algorithmic approach usually requires a mathematical formulation of the problem that includes objectives and constraints. The algorithm could be based on any one or a combination of techniques. The "quality" of the solution is based on the values of the objectives and performance criteria of the given schedule. This form of solution method often consists of multiple phases. In the first phase, a certain amount of *preprocessing* is done, where the problem instance is analyzed and a number of statistics are compiled, e.g., the average processing time, the maximum processing time, the due date tightness. The second phase consists of the actual algorithms and heuristics, whose structure may depend on the statistics compiled in the first phase (for example, in the way the look-ahead parameter K in the ATC rule may depend on the due date tightness and due date range factors). The third phase may contain a *postprocessor*. The solution that comes out of the second phase is fed into a procedure such as simulated annealing or tabu-search, in order to see if improvements can be obtained. This type of solution method is usually coded in a procedural language such as Fortran, Pascal or C.

The knowledge-based and constraint programming approaches are quite different from the algorithmic approach. These approaches are often more concerned with underlying problem structures that cannot be described easily in an analytical format. In order to incorporate the decisionmaker's knowledge into the system, objects, rules or constraints are used. These approaches are often used when it is only necessary to find a *feasible* solution given the many rules or constraints; however, as some schedules are ranked "more preferable" than others, heuristics may be used to obtain a "more preferred" schedule. Through a so-called *inference engine*, such an approach tries to find solutions that do not violate the prescribed rules and satisfy the stated constraints as much as possible. The logic behind the schedule generation process is often a combination of inferencing techniques and search techniques as described in Appendixes C and D. The inferencing techniques are usually so-called *forward chaining* and *backward chaining* algorithms. A forward chaining algorithm is knowledge driven. It first analyzes the data and the rules and, through inferencing techniques, attempts to construct a feasible solution. A backward chaining algorithm is result oriented. It starts out with a promising solution and attempts to verify whether it is feasible. Whenever a satisfactory solution does not appear to exist or when the system's user thinks that it is too difficult to find, the user may want to reformulate the problem by relaxing some of the constraints. The relaxation of constraints may be done either automatically (by the system itself) or by the user. Because of this aspect, the knowledge-based and constraint programming approaches have at times also been referred to as *reformulative* approaches.

The programming style used for the development of these systems is different from the ones used for systems based on algorithmic approaches. The

programming style may depend on the form of the knowledge representation. If the knowledge is represented in the form of *IF-THEN* rules, then the system can be coded using an expert system shell. The expert system shell contains an inference engine that is capable of doing forward chaining or backward chaining of the rules in order to obtain a feasible solution. This approach may have difficulties with conflict resolution and uncertainty. If the knowledge is represented in the form of logic rules (see Example 17.2.4), then Prolog may be a suitable language. If the knowledge is represented in the form of frames, then a language with object oriented extensions is required, e.g., C++. These languages emphasize user-defined objects that facilitate a modular programming style. Examples of systems that are designed according to a constraint programming approach are described in Chapters 16 and 19 and in Appendix C.

Algorithmic approaches as well as knowledge-based and constraint programming approaches have their advantages and disadvantages. An algorithmic approach has an edge if

(i) the problem allows for a crisp and precise mathematical formulation,
(ii) the number of jobs involved is large,
(iii) the amount of randomness in the environment is minimal,
(iv) some form of optimization has to be done frequently and in real time and
(v) the general rules are consistently being followed without too many exceptions.

A disadvantage of the algorithmic approach is that if the operating environment changes (for example, certain preferences on assignments of jobs to machines), the reprogramming effort may be substantial.

The knowledge-based and constraint programming approaches may have an edge if only feasible schedules are needed. Some system developers believe that changes in the environment or in the rules or constraints can be more easily incorporated in a system that is based on such an approach than in a system that is based on the algorithmic approach. Others, however, believe that the effort required to modify any system is mainly a function of how well the code is organized and written; the effort required to modify should not depend that much on the approach used.

A disadvantage of the knowledge-based and constraint programming approaches is that obtaining reasonable schedules may require in some settings substantially more computer time than an algorithmic approach. In practice certain scheduling systems have to operate in near-real time (it is very common that schedules have to be created in minutes).

The amount of available computer time is an important factor in the selection of a schedule generation technique. The time allowed to generate a schedule varies from application to application. Many applications require real time performance: a schedule has to be generated in seconds or minutes on the available computer. This may be the case if rescheduling is required many times a day because of schedule deviations. It would also be true if the scheduling engine runs

iteratively, requiring human interaction between iterations (perhaps for adjustments of workcenter capacities). However, some applications do allow overnight number crunching. For example, a user may start a program at the end of the day and expect an output by the time he or she arrives at work the next day. Some applications do require extensive number crunching. When, in the airline industry, quarterly flight schedules have to be determined, the investments at stake are such that a week of number crunching on a mainframe is fully justified.

As stated before, the two schools of thought have been converging and many scheduling systems that are currently being designed have elements of both. One language of choice is C++ as it is an easy language for coding algorithmic procedures and it also has object-oriented extensions.

17.4 User Interfaces and Interactive Optimization

The user interfaces are very important parts of the system. The interfaces usually determine whether the system is actually going to be used or not. Most user interfaces, whether the system is based on a workstation or PC, make extensive use of window mechanisms. The user often wants to see several different sets of information at the same time. This is the case not only for the static data that is stored in the database, but also for the dynamic data that are schedule dependent.

Some user interfaces allow for extensive user interaction; a decision-maker may be allowed to modify the current status or the current information. Other user interfaces may not allow any modifications. For example, an interface that displays the values of all the relevant performance measures may not allow the user to change any of these values. However, a decision-maker may be allowed to modify the schedule in another interface which then automatically would change the values of the performance measures.

User interfaces for database modules often take a fairly conventional form and may be determined by the particular database package used. These interfaces must allow for some user interaction, because data such as due dates often have to be changed during a scheduling session.

There are often a number of interfaces that exhibit general data concerning the plant or enterprise. Examples of such interfaces are:

(i) the plant layout interface,
(ii) the resource calendar interface, and
(iii) the routing table interface.

The plant layout interface may depict graphically the workcenters and machines in a plant as well as the possible routes between the workcenters. The resource calendar displays shift schedules, holidays and preventive maintenance schedules of the machines. In this interface the user can assign shifts and schedule the servicing of the resources. The routing table typically may show static

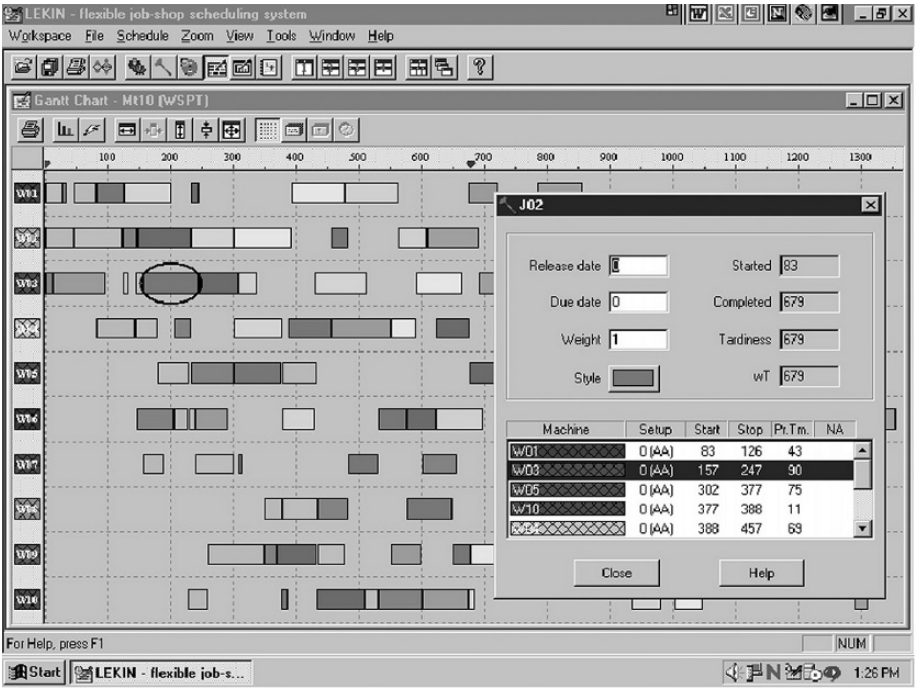


Fig. 17.4 Gantt chart interface

data associated with the jobs. It specifies the machines and/or the operators who can process a particular job or job type.

The module that generates the schedules may provide the user with a number of computational procedures and algorithms. Such a library of procedures within this module will require its own user interface, enabling the user to select the appropriate algorithm or even design an entirely new procedure.

User interfaces that display information regarding the schedules can take many different forms. Interfaces for adjusting or manipulating the schedules basically determine the character of the system, as these are used most extensively. The various forms of interfaces for manipulating solutions depend on the level of detail as well as on the scheduling horizon being considered. In what follows four such interfaces are described in more detail, namely:

(i) the Gantt Chart interface,
(ii) the Dispatch List interface,
(iii) the Capacity Buckets interface, and
(iv) the Throughput Diagram interface.

The first, and probably most popular, form of schedule manipulation interface is the Gantt chart (see Figure 17.4). The Gantt chart is the usual horizontal bar chart, with the horizontal axis representing the time and the vertical axis

the various machines. A color and/or pattern code may be used to indicate a characteristic or an attribute of the corresponding job. For example, jobs that in the current schedule are completed after their due date may be colored red. The Gantt chart usually has a number of scroll capabilities that allow the user to go back and forth in time or focus on particular machines, and is usually mouse driven. If the user is not entirely satisfied with the generated schedule, he may wish to perform some manipulations on his own. With the mouse, the user can "click and drag" an operation from one position to another. Providing the interface with a click, drag, and drop capability is not a trivial task for the following reason. After changing the position of a particular operation on a machine, other operations on that machine may have to be pushed either forward or backward in time to maintain feasibility. The fact that other operations have to be processed at different times may have an effect on the schedules of other machines. This is often referred to as *cascading* or *propagation* effects. After the user has repositioned an operation of a job, the system may call a reoptimization procedure that is embedded in the scheduling engine to deal with the cascading effects in a proper manner.

Example 17.4.1 (Cascading Effects and Reoptimization)

Consider a three machine flow shop with unlimited storage space between the successive machines and therefore no blocking. The objective is to minimize the total weighted tardiness. Consider a schedule with 4 jobs as depicted by the Gantt chart in Figure 17.5.a. If the user swaps jobs 2 and 3 on machine 1, while keeping the order on the two subsequent machines the same, the resulting schedule, because of cascading effects, takes the form depicted in Figure 17.5.b. If the system has reoptimization algorithms at its disposal, the user may decide to reoptimize the operations on machines 2 and 3, while keeping the sequence on machine 1 frozen. A reoptimization algorithm then may generate the schedule depicted in Figure 17.5.c. To obtain appropriate job sequences for machines 2 and 3, the reoptimization algorithm has to solve an instance of the two machine flow shop with the jobs subject to given release dates at the first machine. ||

Gantt charts do have disadvantages, especially when there are many jobs and machines. It may then be hard to recognize which bar or rectangle corresponds to which job. As space on the screen (or on the printout) is rather limited, it is hard to attach text to each bar. Gantt chart interfaces usually provide the capability to click on a given bar and open a window that displays detailed data regarding the corresponding job. Some Gantt charts also have a filter capability, where the user may specify the job(s) that should be exposed on the Gantt chart while disregarding all others. The Gantt chart interface depicted in Figure 17.4 is from the LEKIN system that is described in more detail in Chapter 19.

The second form of user interface for displaying schedule information is the *dispatch-list* interface (see Figure 17.6). Schedulers often want to see a list of the jobs to be processed on each machine in the order in which they are to

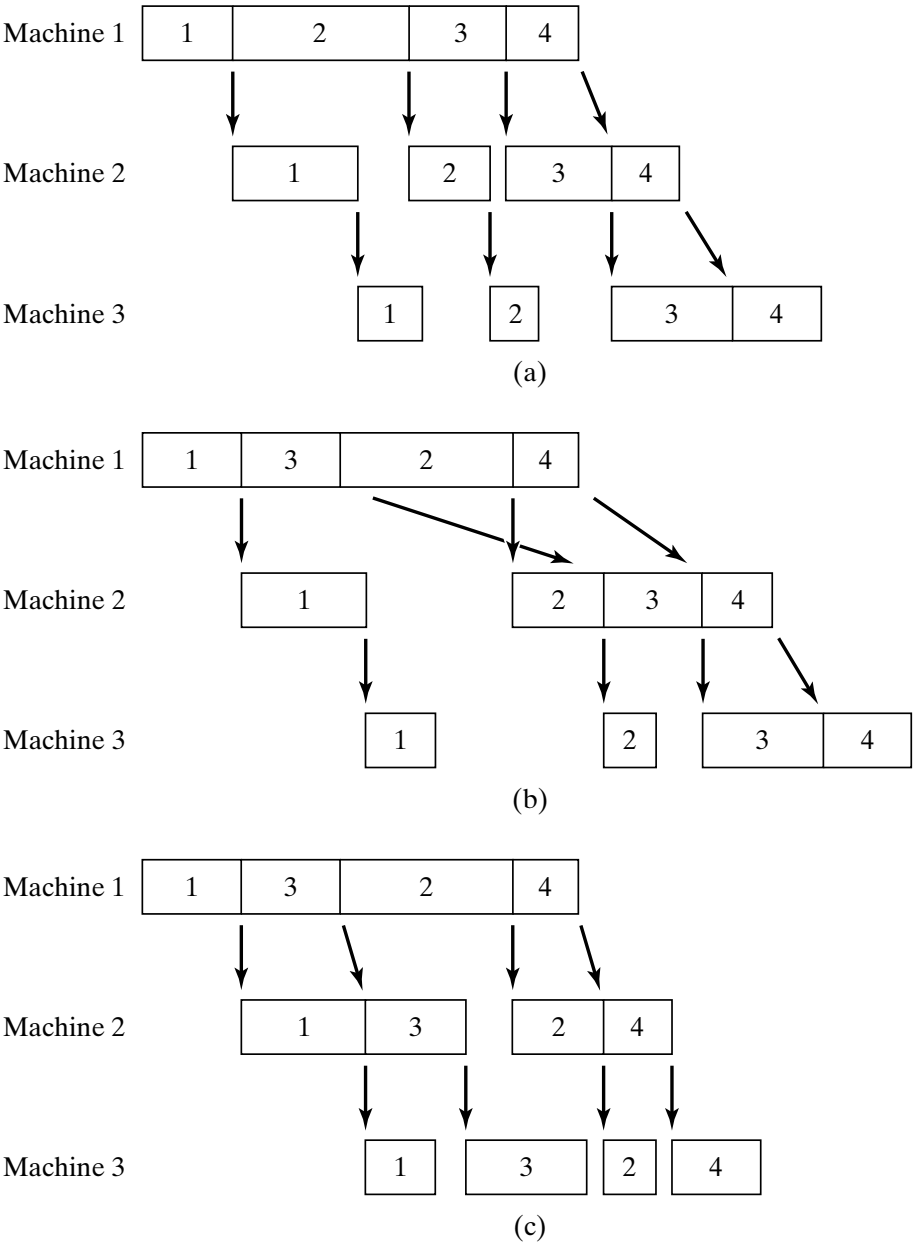


Fig. 17.5 Cascading and reoptimization after swap: (a) original schedule, (b) cascading effects after swap of jobs on machine 1, (c) schedule after reoptimization of Machines 2 and 3.

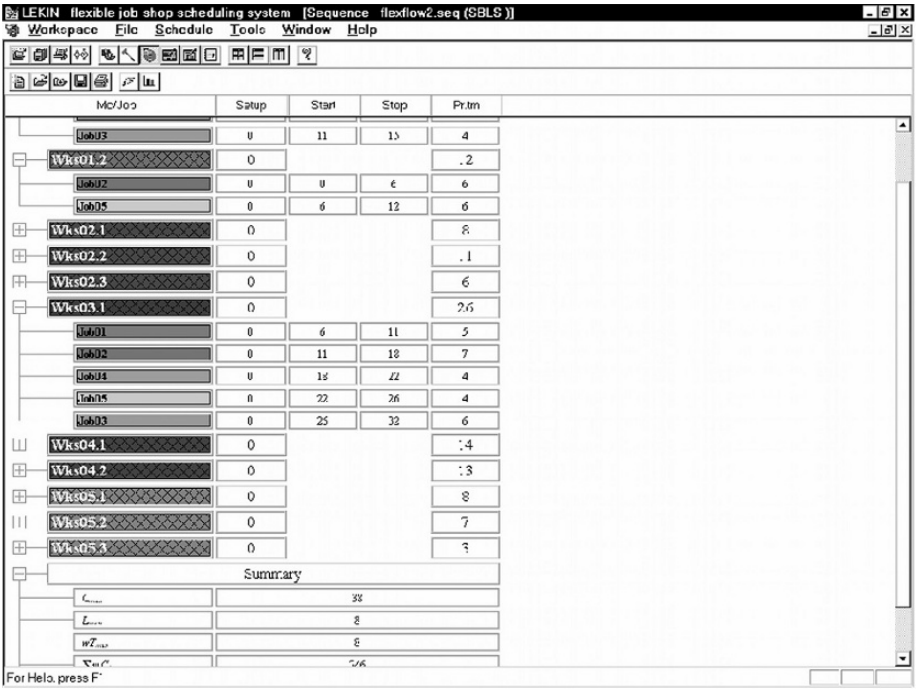


Fig. 17.6 The dispatch list interface of the LEKIN system

be processed. With this type of display schedulers also want to have editing capabilities so they can change the sequence in which jobs are processed on a machine or move a job from one machine to another. This sort of interface does not have the disadvantage of the Gantt chart, since the jobs are listed with their job numbers and the scheduler knows exactly where each job is in the sequence. If the scheduler would like more attributes (e.g., processing time, due date, completion time under the current schedule, and so on) of the jobs to be listed, then more columns can be added next to the job number column, each one with a particular attribute. The disadvantage of the dispatch-list interface is that the scheduler does not have a clear view of the schedule relative to time. The user may not see immediately which jobs are going to be late, which machine is idle most of the time, etc. The dispatch-list interface in Figure 17.6 is also from the LEKIN system.

The third form of user interface is the *capacity buckets* interface (see Figure 17.7). The time axis is partitioned into a number of time slots or buckets. Buckets may correspond to either days, weeks or months. For each machine the processing capacity of a bucket is known. The creation of a schedule may in certain environments be accomplished by assigning jobs to machines in given time segments. After such assignments are made, the capacity buckets inter-

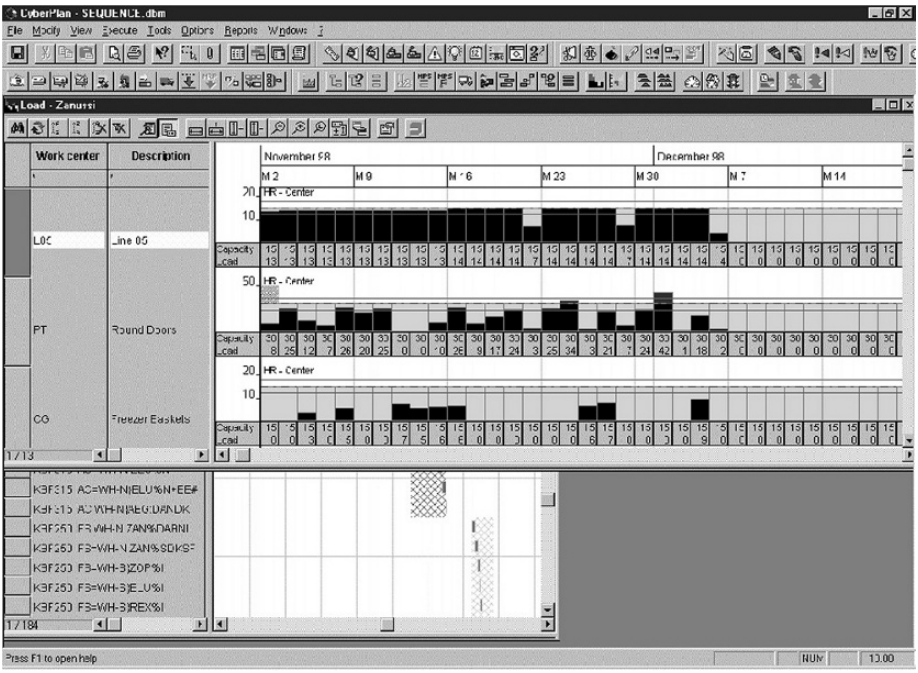


Fig. 17.7 The capacity buckets interface of the Cyberplan system

face displays for each machine the percentage of the capacity utilized in each time segment. If the decision-maker sees that in a given time period a machine is overutilized, he knows that some jobs in the corresponding bucket have to be rescheduled. The capacity buckets interface contrasts, in a sense, with the Gantt chart interface. A Gantt chart indicates the number of late jobs as well as their respective tardinesses. The number of late jobs and the total tardiness give an indication of the deficiency in capacity. The Gantt chart is thus a good indicator of the available capacity in the short term (days or weeks) when there are a limited number of jobs (twenty or thirty). Capacity buckets are useful when the scheduler is performing medium or long term scheduling. The bucket size may be either a week or a month and the total period covered three or four months. Capacity buckets are, of course, a cruder form of information as they do not indicate which jobs are completed on time and which ones are late. The capacity buckets interface depicted in Figure 17.7 is from the Cyberplan system developed by Cybertec.

The fourth form of user interface is the *input-output diagram* or *throughput diagram* interface, which are often of interest when the production is made to stock. These diagrams describe the total amount of orders received, the total amount produced and the total amount shipped, cumulatively over time. The difference, at any point in time, between the first two curves is the total amount

of orders waiting for processing and the difference between the second and the third curves equals the total amount of finished goods in inventory. This type of interface specifies neither the number of late jobs nor their respective tardinesses. It does provide the user with information concerning machine utilization and Work-In-Process (WIP).

Clearly, the different user interfaces for the display of information regarding schedules have to be strongly linked to one another. When a user makes changes in either the Gantt chart interface or the dispatch-list interface, the dynamic data may change considerably because of cascading effects or the reoptimization process. Changes made in one interface, of course, have to be displayed immediately in the other interfaces as well.

User interfaces for the display of information regarding the schedules have to be linked to other interfaces also, e.g., database management interfaces and interfaces of a scheduling engine. For example, a user may modify an existing schedule in the Gantt chart interface by clicking, dragging, and dropping; then he may want to freeze certain jobs in their respective positions. After doing this, he may want to reoptimize the remaining (unfrozen) jobs using an algorithm in the scheduling engine. These algorithms are similar to the algorithms described in Parts II and III for situations where machines are not available during given time periods (because of breakdowns or other reasons). The interfaces that allow the user to manipulate the schedules have to be, therefore, strongly linked to the interfaces for algorithm selection.

User interfaces may also have a separate window that displays the values of all relevant performance measures. If the user has made a change in a schedule the values before and after the change may be displayed. Typically, performance measures are displayed in plain text format. However, more sophisticated graphical displays may also be used.

Some user interfaces are sophisticated enough to allow the user to split a job into a number of smaller segments and schedule each of these separately. Splitting an operation is equivalent to (possibly multiple) preemptions. The more sophisticated user interfaces also allow different operations of the same job to overlap in time. In practice, this may occur in many settings. For example, a job may start at a downstream machine of a flow shop before it has completed its processing at an upstream machine. This occurs when a job represents a large batch of identical items. Before the entire batch has been completed at an upstream machine, parts of the batch may already have been transported to the next machine and may have already started their processing there.

17.5 Generic Systems vs. Application-Specific Systems

Dozens of software houses have developed systems that they claim can be implemented in many different industrial settings after only some minor customization. It often turns out that the effort involved in customizing such systems is quite substantial. The code developed in the customization process may end up

to be more than half the total code. However, some systems have very sophisticated configurations that allow them to be tailored to different types of industries without much of a programming effort. These systems are highly modular and have an edge with regard to the adjustments to specific requirements. A generic system, if it is highly modular, can be changed to fit a specific environment by adding specific modules, e.g., tailor-made scheduling algorithms. Experts can develop such algorithms and the generic scheduling software supplies standard interfaces or "hooks" that allow integration of the special functions in the package. This concept allows the experts to concentrate on the scheduling problem, while the generic software package supplies the functionalities that are less specific, e.g., user interfaces, data management, standard scheduling algorithms for less complex areas, and so on.

Generic systems may be built either on top of a commercial database system, such as Sybase or Oracle, or on top of a proprietary database system developed specifically for the scheduling system. Generic systems use processing data similar to the data presented in the framework described in Chapter 2. However, the framework in such a system may be somewhat more elaborate than the framework presented in Chapter 2. For example, the database may allow for an alphanumeric order number that refers to the name of a customer. The order number then relates to several jobs, each one with its own processing time (which often may be referred to as the *quantity* of a batch) and a routing vector that determines the precedence constraints the operations are subject to. The order number has its own due date (committed shipping date), weight (priority factor) and release date (which may be determined by a Material Requirements Planning (MRP) system that is connected to the scheduling system). The system may include procedures that translate the due date of the order into due dates for the different jobs at the various workcenters. Also, the weights of the different jobs belonging to an order may not be exactly equal to the weight of the order itself. The weights of the different jobs may be a function of the amount of value already added to the product. The weight of the last job pertaining to an order may be larger than the weight of the first job pertaining to that order.

The way the machine or resource environment is represented in the database is also somewhat more elaborate than the way it is described in Chapter 2. For example, a system typically allows a specification of workcenters and, within each workcenter, a specification of machines.

Most generic scheduling systems have routines for generating a "first" schedule for the user. Of course, such an initial solution rarely satisfies the user. That is why scheduling systems often have elaborate user interfaces that allow the user to manually modify an existing schedule. The automated scheduling capabilities generally consist of a number of different dispatching rules that are basically sorting routines. These rules are similar to the priority rules discussed in the previous chapters (SPT, LPT, WSPT, EDD and so on). Some generic systems rely on more elaborate procedures, such as *forward loading* or *backward loading*. Forward loading implies that the jobs are inserted one at the

time starting at the beginning of the schedule, that is, at the current time. Backward loading implies that the schedule is generated starting from the back of the schedule, that is, from the due dates, working its way towards the current time (again inserting one job at the time). These insertions, either forward or backward in time, are done according to some priority rule. Some of the more sophisticated automated procedures first identify the bottleneck workcenter(s) or machine(s); they compute time windows during which jobs have to be processed on these machines and then schedule the jobs on these machines using some algorithmic procedure. After the bottlenecks are scheduled, the procedure schedules the remaining machines through either forward loading or backward loading.

Almost all generic scheduling systems have user interfaces that include Gantt charts and enable the user to manipulate the solutions manually. However, these Gantt chart interfaces are not always perfect. For example, most of them do *not* take into account the cascading and propagation effects referred to in the previous section. They may do some automatic rescheduling on the machine or workcenter where the decision maker has made a change, but they usually do not adapt the schedules on other machines or workcenters to this change. The solutions generated may at times be infeasible. Some systems give the user a warning in case, after the modifications, the schedule turns out to be infeasible.

Besides the Gantt chart interface, most systems have at least one other type of interface that either displays the actual schedule or provides important data that is related. The second interface is typically one of those mentioned in the previous section.

Generic systems usually have fairly elaborate report generators, that print out the schedule with alphanumeric characters; such printouts can be done fast and on an inexpensive printer. The printout may then resemble what is displayed, for example, in the dispatch-list interface described in the previous section. It is possible to list the jobs in the order in which they will be processed at a particular machine or workcenter. Besides the job number, other relevant job data may be printed out as well. There are also systems that print out entire Gantt charts. But Gantt charts have the disadvantage mentioned before, namely that it may not be immediately obvious which rectangle or bar corresponds to which job. Usually the bars are too small to append any information to.

Generic systems have a number of advantages over application-specific systems. If the scheduling problem is a fairly standard one and only minor customization of a generic system suffices, then this option is usually less expensive than developing an application-specific system from scratch. An additional advantage is that an established company will maintain the system. On the other hand, most software houses that develop scheduling systems do not provide the source code. This makes the user of the system dependent on the software house even for very minor changes.

In many instances generic systems are simply not suitable and application-specific systems (or modules) have to be developed. There are several good reasons for developing application-specific systems. One reason may be that

the scheduling problem is simply so large (because of the number of machines, jobs, or attributes) that a PC-based generic system simply would not be able to handle it. The databases may be very large and the required interface between the shopfloor control system and the scheduling system has to be of a kind that a generic system cannot handle. An example of an environment where this is often the case is semiconductor manufacturing.

A second reason to opt for an application-specific system is that the environment may have so many idiosyncrasies that no generic system can be modified in such a way that it can address the problem satisfactorily. The processing environment may have certain restrictions or constraints that are hard to attach to or build into a generic system. For example, certain machines at a workcenter have to start with the processing of different jobs at the same time (for one reason or another) or a group of machines may have to sometimes act as a single machine and, at other times, as separate machines. The order portfolio may also have many idiosyncrasies. That is, there may be a fairly common machine environment used in a fairly standard way (that would fit nicely into a generic system), but with too many exceptions on the rules as far as the jobs are concerned. Coding in the special situations represents such a large amount of work that it may be advisable to build a system from scratch.

A third reason for developing an application-specific system is that the user may insist on having the source code in house and on being able to maintain the system within his own organization.

An important advantage of an application-specific system is that manipulating a solution is usually considerably easier than with a generic system.

17.6 Implementation and Maintenance Issues

During the last two decades a large number of scheduling systems have been developed, and many more are under development. These developments have made it clear that a certain proportion of the theoretical research done over the last couple of decades is of very limited use in real world applications. Fortunately, the system development that is going on in industry is currently encouraging theoretical researchers to tackle scheduling problems that are more relevant to the real world. At various academic institutions in Europe, Japan and North America, research is focusing on the development of algorithms as well as on the development of systems; significant efforts are being made in integrating these developments.

Over the last two decades many companies have made large investments in the development and implementation of scheduling systems. However, not all the systems developed or installed appear to be used on a regular basis. Systems, after being implemented, often remain in use only for a limited time; after a while they may be ignored altogether.

In those situations where the systems are in use on a more or less permanent basis, the general feeling is that the operations do run smoother. A system

that is in place often does *not* reduce the time the decision-maker spends on scheduling. However, a system usually does enable the user to produce better solutions. Through an interactive Graphics User Interface (GUI) a user is often able to compare different solutions and monitor the various performance measures. There are other reasons for smoother operations besides simply better schedules. A scheduling system imposes a certain "discipline" on the operations. There are now compelling reasons for keeping an accurate database. Schedules are either printed out neatly or displayed on monitors. This apparently has an effect on people, encouraging them to actually even follow the schedules.

The system designer should be aware of the reasons why some systems have never been implemented or are never used. In some cases, databases are not sufficiently accurate and the team implementing the system does not have the patience or time to improve the database (the people responsible for the database may be different from those installing the scheduling system). In other cases, the way in which workers' productivity is measured is not in agreement with the performance criteria the system is based upon. User interfaces may not permit the user of the system to reschedule sufficiently fast in the case of unexpected events. Procedures that enable rescheduling when the main user is absent (for example, if something unexpected happens during the night shift) may not be in place. Finally, systems may not be given sufficient time to "settle" or "stabilize" in their environment (this may require many months, if not years).

Even if a system gets implemented and used, the duration during which it remains in use may be limited. Every so often, the organization may change drastically and the system is not flexible enough to provide good schedules for the new environment. Even a change in a manager may derail a system.

In summary, the following points should be taken into consideration in the design, development, and implementation of a system.

1. Visualize how the operating environment will evolve over the lifetime of the system before the design process actually starts.

2. Get all the people affected by the system involved in the design process. The development process has to be a team effort and all involved have to approve the design specifications.

3. Determine which part of the system can be handled by off-the-shelf software. Using an appropriate commercial code may speed up the development process considerably.

4. Keep the design of the software modular. This is necessary not only to facilitate the entire programming effort, but also to facilitate changes in the system after its implementation.

5. Make the objectives of the algorithms embedded in the system consistent with the performance measures by which people who must act according to the schedules are being judged.

6. Do not take the data integrity of the database for granted. The system has to be able to deal with faulty or missing data and provide the necessary safeguards.

7. Capitalize on potential side benefits of the system, e.g., spin-off reports for distribution to key people. This enlarges the supporters base of the system.

8. Make provisions to ensure easy rescheduling, not only by the main scheduler but also by others, in case the main user is absent.

9. Keep in mind that installing the system requires a considerable amount of patience. It may take months or even years before the system runs smoothly. This period should be a period of continuous improvement.

10. Do not underestimate the necessary maintenance of the system after its installation. The effort required to *keep* the system in use on a regular basis is considerable.

It appears that in the near future, an even larger effort will be made in the design, development and implementation of scheduling systems and that such systems will play an important role in Computer Integrated Manufacturing.

Exercises

17.1. Consider a job shop with machines in parallel at each workcenter (i.e., a flexible job shop). Hard constraints as well as soft constraints play a role in the scheduling of the machines. More machines may be installed in the near future. The scheduling process does not have to be done in real time, but can be done overnight. Describe the advantages and disadvantages of an algorithmic approach and of a knowledge-based approach.

17.2. Consider a factory with a single machine with sequence dependent setup times and hard due dates. It does not appear that changes in the environment are imminent in the near future. Scheduling and rescheduling has to be done in real time.

(a) List the advantages and disadvantages of an algorithmic approach and of a knowledge-based approach.

(b) List the advantages and disadvantages of a commercial system and of an application-specific system.

17.3. Design a schedule generation module that is based on a composite dispatching rule for a parallel machine environment with the jobs subject to sequence dependent setup times. Job j has release date r_j and may only be processed on a machine that belongs to a given set M_j. There are three objectives, namely $\sum w_j T_j$, $C_{\max}$ and $L_{\max}$. Each objective has its own weight and the weights are time dependent; every time the scheduler uses the system

he puts in the relative weights of the various objectives. Design the composite dispatching rule and explain how the scaling parameters depend on the relative weights of the objectives.

17.4. Consider the following three measures of machine congestion over a given time period.

(i) the number of late jobs during the period;
(ii) the average number of jobs waiting in queue during the given period.
(iii) the average time a job has to wait in queue during the period.

How does the selection of congestion measure depend on the objective to be minimized?

17.5. Consider the following scheduling alternatives:

(i) forward loading (starting from the current time);
(ii) backward loading (starting from the due dates);
(iii) scheduling from the bottleneck stage first.

How does the selection of one of the three alternatives depend on the following factors:

(i) degree of uncertainty in the system.
(ii) balanced operations (not one specific stage is a bottleneck).
(iii) due date tightness.

17.6. Consider the ATC rule. The K factor is usually determined as a function of the due date tightness factor τ and the due date range factor R. However, the process usually requires extensive simulation. Design a learning mechanism that refines the function f that maps τ and R into K during the regular (possibly daily) use of the system's schedule generator.

17.7. Consider an interactive scheduling system with a user interface for schedule manipulation that allows "freezing" of jobs. That is, the scheduler can click on a job and freeze the job in a certain position. The other jobs have to be scheduled around the frozen jobs. Freezing can be done with tolerances, so that in the optimization process of the remaining jobs the frozen jobs can be moved a little bit. This facilitates the scheduling of the unfrozen jobs. Consider a system that allows freezing of jobs with specified tolerances and show that freezing in an environment that does not allow preemptions requires tolerances of at least half the maximum processing time in either direction in order to avoid machine idle times.

17.8. Consider an interactive scheduling system with a user interface that only allows for freezing of jobs with no (zero) tolerances.

(a) Show that in a nonpreemptive environment the machine idle times caused by frozen jobs are always less than the maximum processing time.

(b) Describe how procedures can be designed that minimize in such a scenario machine idle times in conjunction with other objectives, such as the total completion time.

17.9. Consider a user interface of an interactive scheduling system for a bank of parallel machines. Assume that the reoptimization algorithms in the system are designed in such a way that they optimize each machine separately while they keep the current assignment of jobs to machines unchanged. A move of a job (with the mouse) is said to be *reversible* if the move, followed by the reoptimization procedure, followed by the *reverse* move, followed once more by the reoptimization procedure, results in the original schedule. Suppose now a job is moved with the mouse from one machine to another. Show that such a move is reversible if the reoptimization algorithm minimizes the total completion time. Show that the same is true if the reoptimization algorithm minimizes the sum of the weighted tardinesses.

17.10. Consider the same scenario as in the previous exercise. Show that with the type of reoptimization algorithms described in the previous exercise moves that take jobs from one machine and put them on another are *commutative*. That is, the final schedule does not depend on the sequence in which the moves are done, even if all machines are reoptimized after each move.

Comments and References

Many papers and books have been written on the various aspects of production information systems; see, for example, Gaylord (1987), Scheer (1988) and Pimentel (1990).

With regard to the issues concerning the overall development of scheduling systems a relatively large number of papers have been written, often in proceedings of conferences, e.g., Oliff (1988), Karwan and Sweigart (1989), Interrante (1993). See also Kanet and Adelsberger (1987), Kusiak and Chen (1988), Solberg (1989), Adelsberger and Kanet (1991), Pinedo, Samroengraja and Yen (1994), and Pinedo and Yen (1997).

For research focusing specifically on knowledge-based systems, see Smith, Fox and Ow (1986), Shaw and Whinston (1989), Atabakhsh (1991), Noronha and Sarma (1991), Lefrancois, Jobin and Montreuil (1992) and Smith (1992, 1994).

For work on the design and development of user interfaces for scheduling systems, see Kempf (1989), Woerner and Biefeld (1993), Chimani, Lesh, Mitzenmacher, Sidner and Tanaka (2005), and Derthick and Smith (2007).

Chapter 18

Design and Implementation of Scheduling Systems: More Advanced Concepts

This chapter focuses on a number of issues that have come up in recent years in the design, development, and implementation of scheduling systems. The first section discusses issues concerning uncertainty, robustness and reactive decision making. In practice, schedules often have to be changed because of random events. The more *robust* the original schedule is, the easier the rescheduling is. This section focuses on the generation of robust schedules as well as on the measurement of their robustness. The second section considers machine learning mechanisms. No system can consistently generate good solutions that are to the liking of the user. The decision-maker often has to tweak the schedule generated by the system in order to make it usable. A well-designed system can learn from past adjustments made by the user; the mechanism that enables the system to do this is called a learning mechanism. The third section focuses on the design of scheduling engines. An engine often contains an entire library of algorithms. One procedure may be more appropriate for one type of instance or data set, while another procedure may be more appropriate for another type of instance. The user should be able to select, for each instance, which procedure to apply. It may even be the case that a user would like to tackle an instance using a combination of various procedures. This third section discusses how a scheduling engine should be designed in order to enable the user to adapt and combine algorithms in order to achieve maximum effectiveness. The fourth section goes

M.L. Pinedo, *Scheduling*, DOI: 10.1007/978-0-387-78935-4_18,

into reconfigurable systems. Experience has shown that system development and implementation is very time consuming and costly. In order to reduce the costs, efforts have to be made to maintain a high level of modularity in the design of the system. If the modules are well designed and sufficiently flexible, they can be used over and over again without any major changes. The fifth section focuses on design aspects of web-based scheduling systems. This section discusses the effects of networking on the design of such systems. The sixth and last section discusses a number of other issues and presents a view of how scheduling systems may look like in the future.

18.1 Robustness and Reactive Decision Making

In practice, it often happens that soon after a schedule has been generated, an unexpected event happens that forces the decision-maker to make changes. Such an event may, for example, be a machine breakdown or a rush job that suddenly has to be inserted. Many schedulers believe that in practice, most of the time, the decision making process is a *reactive* process. In a reactive process, the scheduler tries to accomplish a number of objectives. He tries to accommodate the original objectives, and also tries to make the new schedule look, as much as possible, like the original one in order to minimize confusion.

The remaining part of this section focuses primarily on reactive decision making in short term scheduling processes. The number of random events that can occur in a short term may, in certain environments, be very high. Rescheduling is in many environments a way of life. One way of doing the rescheduling is to put all the operations not yet started back in the hopper, and generate a new schedule from scratch while taking into account the disruptions that just occurred. The danger is that the new schedule may be completely different from the original schedule, and a big difference may cause confusion.

If the disruption is minor, e.g., the arrival of just one unexpected job, then a simple change may suffice. For example, the scheduler may insert the unexpected arrival in the current schedule in such a way that the total additional setup is minimized and no other high priority job is delayed. A major disruption, like the breakdown of an important machine, often requires substantial changes in the schedule. If a machine goes down for an extended period of time, then the entire workload allocated to that machine for that period has to be transferred to other machines. This may cause extensive delays.

Another way of dealing with the rescheduling process is to somehow anticipate the random events. In order to do so, it is necessary for the original schedule to be robust so that the changes after a disruption are minimal.

Schedule robustness is a concept that is not easy to measure or even define. Suppose the completion time of a job is delayed by δ (because of a machine breakdown or the insertion of a rush job). Let $C'_j(\delta)$ denote the new completion time of job j (i.e., the new time when job j leaves the system), assuming the sequences of all the operations on all the machines remain the same. Of course,

the new completion times of all the jobs are a function of δ. Let Z denote the value of the objective function before the disruption occurred and let $Z'(\delta)$ denote the value of the objective function after the disruption. So $Z'(\delta) - Z$ is the difference due to the disruption. One measure of schedule robustness is

$$\frac{Z'(\delta) - Z}{\delta},$$

which is a function of δ. For small values of δ the ratio may be low whereas for larger values of δ the ratio may get progressively worse. It is to be expected that this ratio is increasing convex in δ.

A more accurate measure of robustness can be established when the probabilities of certain events can be estimated in advance. Suppose a perturbation of a random size Δ may occur and the probability that the random variable Δ assumes the value δ, i.e., $P(\Delta = \delta)$, can be estimated. If Δ can assume only integer values, then

$$\sum_{\delta=0}^{\infty} \Big(Z'(\delta) - Z\Big) P(\Delta = \delta)$$

is an appropriate measure for the robustness. If the random variable Δ is a continuous random variable with a density function $f(\delta)$, then an appropriate measure is

$$\int_{\delta=0}^{\infty} (Z'(\delta) - Z) f(\delta) d\delta.$$

In practice, it may be difficult to make a probabilistic assessment of random perturbations and one may want to have more practical measures of robustness. For example, one measure could be based on the amount of slack between the completion times of the jobs and their respective due dates. So a possible measure for the robustness of schedule $\mathcal{S}$ is

$$\mathcal{R}(\mathcal{S}) = \frac{\sum_{j=1}^{n} w_j (d_j - C_j)}{\sum w_j d_j}.$$

The larger $\mathcal{R}(\mathcal{S})$, the more robust the schedule. Maximizing this particular measure of robustness is somewhat similar to maximizing the total weighted earliness.

When should a decision-maker opt for a more robust schedule? This may depend on the probability of a disruption as well as on his or her ability to reschedule.

Example 18.1.1 (Measures of Robustness)

Consider a single machine and three jobs. The job data are presented in the table below.

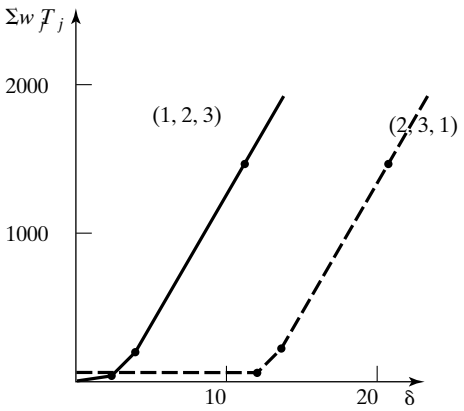


Fig. 18.1 Increase in objective value as function of disruption level (Example 18.1.1)

jobs	1	2	3
p_j	10	10	10
d_j	10	22	34
w_j	1	100	100

The schedule that minimizes the total weighted tardiness is schedule $1, 2, 3$ with a total weighted tardiness of 0. It is clear that this schedule is not that robust, since two jobs with very large weights are scheduled for completion very close to their respective due dates. Suppose that immediately after the decision-maker has decided upon schedule $1, 2, 3$ (i.e., at time $0+\epsilon$) a disruption occurs and the machine goes down for $\delta = 10$ time units. The machine can start processing the three jobs at time $t = 10$. If the original job sequence $1, 2, 3$ is maintained, then the total weighted tardiness is 1410. The manner in which the total weighted tardiness of sequence $1, 2, 3$ depends on the value of δ is depicted in Figure 18.1.

If the original schedule is $2, 3, 1$, then the total weighted tardiness, with no disruptions, is 20. However, if a disruption does occur at time $0+\epsilon$, then the impact is considerably less severe than with schedule $1, 2, 3$. If $\delta = 10$, then the total weighted tardiness is 30. The way the total weighted tardiness under sequence $2, 3, 1$ depends on δ is also depicted in Figure 18.1. From Figure 18.1 it is clear that schedule $2, 3, 1$ (even though originally suboptimal) is more robust than schedule $1, 2, 3$.

Under schedule $1, 2, 3$ the robustness is

$$\mathcal{R}(1,2,3) = \frac{\sum_{j=1}^{n} w_j(d_j - C_j)}{\sum w_j d_j} = \frac{600}{5610} = 0.11,$$

whereas

$$\mathcal{R}(2,3,1) = \frac{2580}{5610} = 0.46.$$

So according to this particular measure of robustness schedule $2,3,1$ is considerably more robust.

Suppose that with probability 0.01 a rush job with processing time 10 arrives at time $0+\epsilon$ and that the decision-maker is not allowed, at the completion of this rush job, to change the original job sequence. If at the outset he had selected schedule $1,2,3$, then the total expected weighted tardiness is

$$0 \times 0.9 + 1410 \times 0.1 = 141.$$

If he had selected schedule $2,3,1$, then the total expected weighted tardiness is

$$20 \times 0.9 + 30 \times 0.1 = 21.$$

So with a 10% probability of a disruption it is better to go for the more robust schedule.

Even if a scheduler is allowed to reschedule after a disruption, he still may not choose at time 0 a schedule that is optimal with respect to the original data. ||

Several other measures of robustness can be defined. For example, assume again that the completion of one job is delayed by δ. However, before computing the effect of the disruption on the objective, each machine sequence is reoptimized separately, i.e., the machine sequences are reoptimized one by one on a machine by machine basis. After this reoptimization the difference in the objective function is computed. The measure of robustness is then a similar ratio as the one defined above. The impact of the disruption is now harder to compute, since different values of δ may result in different schedules. This ratio is, of course, less than the ratio without reoptimization. An even more complicated measure of robustness assumes that after a disruption a reoptimization is done on a more global scale rather than on a machine by machine basis, e.g., under this assumption a disruption may cause an entire job shop to be reoptimized. Other measures of robustness may even allow preemptions in the reoptimization process.

Totally different measures of robustness can be defined based on the capacity utilization of the bottleneck machines (i.e., the percentages of time the machines are utilized) and on the levels of WIP inventory that are kept in front of these machines.

How can one generate robust schedules? Various rules can be followed when creating schedules, for example,

(i) insert idle times,

(ii) schedule less flexible jobs first,

(iii) do not postpone the processing of any operation unnecessarily, and

(iv) keep always a number of jobs waiting in front of highly utilized machines.

The first rule prescribes the insertion of idle periods on given resources at certain points in time. This is equivalent to scheduling the machines below capacity. The durations of the idle periods as well as their timing within the schedule depend on the expected nature of the disruptions. One could argue that the idle periods in the beginning of the schedule may be kept shorter than the idle periods later in the schedule, since the probability of an event occurring in the beginning may be smaller than later on. In practice, some schedulers follow a rule whereby at any point in time in the current week the machines are utilized up to 90% of capacity, the next week up to 80% and the week after that up to 70%. However, one reason for keeping the idle periods in the beginning of the schedule at the same length may be the following: even though the probability of a disruption is small, its relative impact is more severe than that of a disruption that occurs later on in the process.

The second rule suggests that less flexible jobs should have a higher priority than more flexible jobs. If a disruption occurs, then the more flexible jobs remain to be processed. The flexibility of a job is determined, for example, by the number of machines that can do its processing (e.g., the machine eligibility constraints described in Chapter 2). However, the flexibility of a job may also be determined by the setup time structure. Some jobs may require setups that do *not* depend on the sequence. Other jobs may have sequence dependent setup times that are highly variable. The setup times are short only when they follow certain other jobs; otherwise the setup times are very long. Such jobs are clearly less flexible.

The third rule suggests that the processing of a job should not be postponed unnecessarily. Because of inventory holding costs and earliness penalties, it may be desirable to start operations as late as possible. However, from a robustness point of view, it may be desirable to start operations as early as possible. So there is a trade-off between robustness and earliness penalties or inventory holding costs.

The fourth rule tries to make sure that a bottleneck machine never starves because of random events that occur upstream. It makes sense to have always a number of jobs waiting for processing at a bottleneck machine. The reason is the following: if no inventory is kept in front of the bottleneck and the machine feeding the bottleneck suddenly breaks down, then the bottleneck may have to remain idle and may not be able to make up for the lost time later on.

Example 18.1.2 (Starvation Avoidance)

Consider a two machine flow shop with 100 identical jobs. Each job has a processing time of 5 time units on machine 1 and of 10 time units on machine 2. Machine 2 is therefore the bottleneck. However, after each job completion on machine 1, machine 1 may have to undergo a maintenance service for a duration of 45 time units during which it cannot do any processing. The probability that such a service is required is 0.01.

The primary objective is the minimization of the makespan and the secondary objective is the average amount of time a job remains in the system, i.e., the time in between the start of a job on machine 1 and its completion on machine 2 (this secondary objective is basically equivalent to the minimization of the Work-In-Process). However, the weight of the primary objective is 1000 times the weight of the secondary objective.

Because of the secondary objective it does not make sense to let machine 1 process the 100 jobs one after another and finish them all by time 500. In an environment in which machine 1 never requires any servicing, the optimal schedule processes the jobs on machine 1 with idle times of 5 time units in between. In an environment in which machine 1 needs servicing with a given probability, it is necessary to have at all times some jobs ready for processing on machine 2. The optimal schedule is to keep consistently 5 jobs waiting for processing on machine 2. If machine 1 has to be serviced, then machine 2 does not lose any time and the makespan does not go up unnecessarily.

This example illustrates the trade-off between capacity utilization and minimization of Work-In-Process. ||

Robustness and rescheduling have a strong influence on the design of the user interfaces and on the design of the scheduling engine (multi-objective scheduling where one of the performance measures is robustness). Little theoretical research has been done on these issues. This topic may become an important research area in the near future.

18.2 Machine Learning Mechanisms

In practice, the algorithms embedded in a scheduling system often do not yield schedules that are acceptable to the user. The inadequacy of the algorithms is based on the fact that scheduling problems (which often have multiple objectives) are inherently intractable. It is extremely difficult to develop algorithms that can provide for any instance of a problem a reasonable and acceptable solution in real time.

New research initiatives are focusing on the design and development of learning mechanisms that enable scheduling systems which are in daily use to improve their solution generation capabilities. This process requires a substantial amount of experimental work. A number of machine learning methods have been studied with regard to their applicability to scheduling. These methods can be categorized as follows:

(i) rote learning,
(ii) case-based reasoning,
(iii) induction methods and neural networks,
(iv) classifier systems.

These four classes of learning mechanisms are in what follows described in some more detail.

Rote learning is a form of brute force memorization. The system saves old solutions that gave good results together with the instances on which they were applied. However, there is no mechanism for generalizing these solutions. This form of learning is only useful when the number of possible scheduling instances is limited, i.e., a small number of jobs of very few different types. It is not very effective in a complex environment, when the probability of a similar instance occurring again is very small.

Case-based reasoning attempts to exploit experience gained from similar problems solved in the past. A scheduling problem requires the identification of salient features of past schedules, an interpretation of these features, and a mechanism that determines which case stored in memory is the most useful in the current context. Given the large number of interacting constraints inherent in scheduling, existing case indexing schemes are often inadequate for building the case base and subsequent retrieval, and new ways have to be developed. The following example shows how the performance of a composite dispatching rule can be improved using a crude form of case-based reasoning; the form of case-based reasoning adopted is often referred to as the parameter adjustment method.

Example 18.2.1 (Case-Based Reasoning: Parameter Adjustment)
Consider the $1 \mid s_{jk} \mid \sum w_j T_j$ problem (i.e., there is a single machine with n jobs and the total weighted tardiness $\sum w_j T_j$ has to be minimized). Moreover, the jobs are subject to sequence dependent setup times s_{jk}. This problem has been considered in Example 14.2.1. A fairly effective composite dispatching rule for this scheduling problem is the ATCS rule. When the machine has completed the processing of job l at time t, the ATCS rule calculates the ranking index of job j as

$$I_j(t,l) = \frac{w_j}{p_j} \exp\Big(-\frac{\max(d_j - p_j - t, 0)}{K_1 \bar{p}}\Big) \exp\Big(-\frac{s_{lj}}{K_2 \bar{s}}\Big),$$

where $\bar{s}$ is the average setup time of the jobs remaining to be scheduled, K_1 the scaling parameter for the function of the due date of job j and K_2 the scaling parameter for the setup time of job j. As described in Chapter 14, the two scaling parameters K_1 and K_2 can be regarded as functions of three factors:

(i) the due date tightness factor τ,
(ii) the due date range factor R,
(iii) the setup time severity factor $\eta = \bar{s}/\bar{p}$.

However, it is difficult to find suitable functions that map the three factors into appropriate values for the scaling parameters K_1 and K_2.

At this point a learning mechanism may be useful. Suppose that in the scheduling system there are functions that map combinations of the three

factors τ, R and η onto two values for K_1 and K_2. These do not have to be algebraic functions; they may be in the form of tables. When a scheduling instance is considered, the system computes τ, R and η and looks in the current tables for the appropriate values of K_1 and K_2. (These values for K_1 and K_2 may then have to be determined through an interpolation). The instance is then solved using the composite dispatching rule with these values for K_1 and K_2. The objective value of the schedule generated is computed as well. However, in that same step, without any human intervention, the system also solves the same scheduling instance using values $K_1+\delta$, $K_1-\delta$, $K_2+\delta$, $K_2-\delta$ (various combinations). Since the dispatching rule is very fast, this can be done in real time. The performance measures of the schedules generated with the perturbed scaling parameters are also computed. If any of these schedules turns out to be substantially better than the one generated with the original K_1 and K_2, then there may be a reason for changing the mapping from the characteristic factors onto the scaling parameters. This can be done internally by the system without any input from the user of the system. ||

The learning mechanism described in the example above is an online mechanism that operates without supervision. This mechanism is an example of case-based reasoning and can be applied to multi-objective scheduling problems as well, even when a simple index rule does not exist.

The third class of learning mechanisms are of the induction type. A very common form of an induction type learning mechanism is a neural network. A neural net consists of a number of interconnected neurons or units. The connections between units have weights, which represent the strengths of these connections. A multi-layer feedforward net is composed of input units, hidden units and output units (see Figure 18.2). An input vector is processed and propagated through the network starting at the input units and proceeding through the hidden units all the way to the output units. The activation level of input unit i is set to the ith component of the input vector. These values are then propagated to the hidden units via the weighted connections. The activation level of each hidden unit is then computed by summing these weighted values, and by transforming the sum through a function f, that is,

$$a_l = f(q_l, \sum_k w_{kl} a_k),$$

where a_l is the activation level of unit l, q_l is the bias of unit l, and w_{kl} is the connection weight between nodes k and l. These activation levels are propagated to the output units via the weighted connections between the hidden units and the output units and transformed again by means of the function f above. The neural net's response to a given input vector is composed of the activation levels of the output units which is referred to as the output vector. The dimension of the output vector does not have to be the same as the dimension of the input vector.

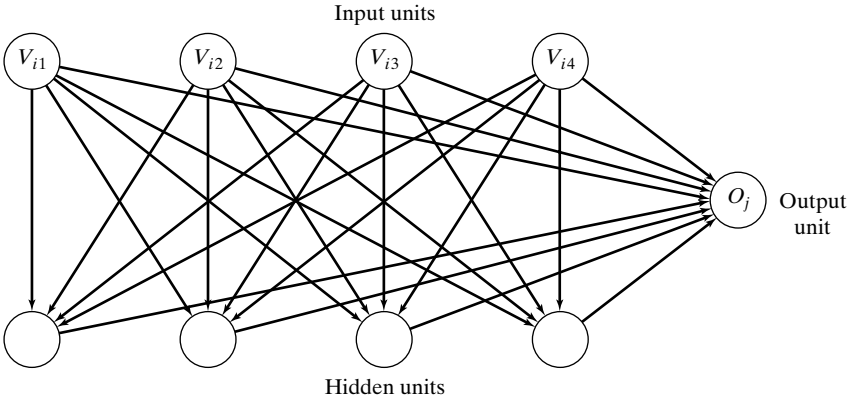


Fig. 18.2 A three-layer neural network

The knowledge of the net is stored in the weights and there are well known methods for adjusting the weights of the connections in order to obtain appropriate responses. For each input vector there is a most appropriate output vector. A learning algorithm computes the difference between the output vector of the current net and the most appropriate output vector and suggests incremental adjustments to the weights. One such method is called the backpropagation learning algorithm.

The next example illustrates the application of a neural net to machine scheduling.

Example 18.2.2 (Neural Net for Parallel Machine Scheduling)
Consider the problem $Qm \mid r_j, s_{jk} \mid \sum w_j T_j$. Machine i has speed v_i and if job j is processed on machine i, then its processing time is $p_{ij} = p_j/v_i$. The jobs have different release dates and due dates. The jobs on a machine are subject to sequence dependent setup times s_{jk}. The objective is to minimize the total weighted tardiness (note that the weights in the objective function are not related to the weights in the neural net).

Suppose that for each machine there is already a partial schedule in place that consists of jobs that already have been assigned. New jobs come in as time goes on. At each new release it has to be decided to which machine the job should be assigned. The neural net has to support this decision-making process.

Typically, the encoding of the data in the form of input vectors is crucial to the problem solving process. In the parallel machines application, each input pattern represents a description of the attributes of a sequence on one of the machines. The values of the attributes of a sequence are determined as follows. Each new job is first positioned where its processing time is the shortest (including the setup time immediately preceding it and the setup

time immediately following it). After this insertion the following attributes are computed with regard to that machine:

(i) the increase in the total weighted completion time of all jobs already scheduled on the machine;
(ii) the average increase in tardiness of the jobs already scheduled on the machine;
(iii) the number of additional jobs that are tardy on the machine;
(iv) the current number of jobs on the machine.

However, the importance of each individual attribute is relative. For example, knowing that the number of jobs on a machine is five does not mean much without knowing the number of jobs on the other machines. Let N_{il} be attribute l with respect to machine i. Two transformations have to be applied to N_{il}.

Translation: The N_{il} value of attribute l under a given sequence on machine i is transformed as follows.

$$N'_{il} = N_{il} - \min(N_{1l}, \ldots, N_{ml}) \qquad i = 1, \ldots, m, \quad l = 1, \ldots, k.$$

In this way, $N'_{i^*l} = 0$ for the best machine, machine i^*, with respect to attribute l and the value N'_{il} corresponds to the difference with the best value.

Normalization: The N'_{il} value is transformed by normalizing over the maximum value in the context.

$$N''_{il} = \frac{N'_{il}}{\max(N'_{1l}, \ldots, N'_{ml})} \qquad i = 1, \ldots, m, \quad l = 1, \ldots, k.$$

Clearly,

$$0 \leq N''_{il} \leq 1.$$

These transformations make the comparisons of the input patterns corresponding to the different machines significantly easier. For example, if the value of attribute l is important in the decision making process, then a machine with a low l-th attribute is more likely to be selected.

A neural net architecture to deal with this problem can be of the structure described in Figure 18.2. This is a three layer network with four input nodes (equal to the number of attributes k), four hidden nodes and one output node. Each input node is connected to all the hidden nodes as well as to the output node. The four hidden nodes are connected to the output node as well.

During the training phase of this network an extensive job release process has to be considered, say 1000 jobs. Each job release generates m input patterns (m being the number of machines) which have to be fed into the network. During this training process the *desired* output of the net is set equal to 1 when the machine associated with the given input is selected

by the expert for the new release and equal to 0 otherwise. For each input vector there is a desired output and the learning algorithm has to compute the error or difference between the current neural net output and the desired output in order to make incremental changes in the connection weights. A well-known learning algorithm for the adjustment of connection weights is the backpropagation learning algorithm; this algorithm requires the choice of a so-called learning rate and a momentum term. At the completion of the training phase the connection weights are fixed.

Using the network after the completion of the training phase requires that every time a job is released the m input patterns are fed into the net; the machine associated with the output closest to 1 is then selected for the given job. ||

In contrast to the learning mechanism described in Example 18.2.1 the mechanism described in Example 18.2.2 requires off-line learning with supervision, i.e., training.

The fourth class of learning mechanisms are the so-called classifier systems. A common form of classifier system can be implemented via a genetic algorithm (see Chapter 14). However, a chromosome (or string) in such a algorithm now does not represent a schedule, but rather a list of rules (e.g., priority rules) that are to be used in the successive steps (or iterations) of an algorithmic framework designed to generate schedules for the problem at hand. For example, consider a framework for generating job shop schedules similar to Algorithm 7.1.3. This algorithm can be modified by replacing its Step 3 with a priority rule that selects an operation from set Ω'. A schedule for the job shop can now be generated by doing nm successive iterations of this algorithm where nm is the total number of operations. Every time a new operation has to be scheduled, a given priority rule is used to select the operation from the current set Ω'. The information that specifies the priority rule that should be used in each iteration can be stored in a string of length nm for a genetic algorithm. The fitness of such a string is the value of the objective function obtained when all the (local) rules are applied successively in the given framework. This representation of a solution (by specifying rules), however, requires relatively intricate cross-over operators in order to get feasible off-spring. The genetic algorithm is thus used to search over the space of rules and not over the space of actual schedules. The genetic algorithm serves as a meta-strategy that optimally controls the use of priority rules.

18.3 Design of Scheduling Engines and Algorithm Libraries

A scheduling engine in a system often contains a library of algorithmic procedures. Such a library may include basic dispatching rules, composite dispatching rules, shifting bottleneck techniques, local search techniques, branch-and-bound

procedures, beam search techniques, mathematical programming routines, and so on. For a specific instance of a problem one procedure may be more suitable than another. The appropriateness of a procedure may depend on the amount of CPU time available or the length of time the user is willing to wait for a solution.

The user of such a scheduling system may want to have a certain flexibility in the usage of the various types of procedures in the library. The desired flexibility may simply imply an ability to determine which procedure to apply to the given instance of the problem, or it may imply more elaborate ways of manipulating a number of procedures. A scheduling engine may have modules that allow the user

(i) to analyze the data and determine algorithmic parameters,
(ii) to set up algorithms in parallel,
(iii) to set up algorithms in series,
(iv) to integrate algorithms.

For example, an algorithm library allows a user to do the following: he may have statistical procedures at hand which he can apply to the data set in order to generate some statistics, such as average processing time, range of processing times, due date tightness, setup time severity, and so on. Based on these statistics the user can select a procedure and specify the appropriate levels for its parameters (e.g., scaling parameters, lengths of tabu-lists, beam widths, number of iterations, and so on).

If a user has more than one computer or processor at his disposal, he may want to apply different procedures concurrently (i.e., in parallel), since he may not know in advance which one is the most suitable for the instance being considered. The different procedures function then completely independently from one another.

A user may also want to concatenate procedures i.e., set various procedures up in series. That is, he or she would set the procedures up in such a way that the output of one serves as an input to another, e.g., the outcome of a dispatching rule serves as the initial solution for a local search procedure. The transfer of data from one procedure to the next is usually relatively simple. For example, it may be just a schedule, which, in the case of a single machine, is a permutation of the jobs. In a parallel machine environment or in a job shop environment it may be a collection of sequences, one for each machine.

Example 18.3.1 (Concatenation of Procedures)

Consider a scheduling engine that allows a user to feed the outcome of a composite dispatching rule into a local search procedure. This means that the output of the first stage, i.e., the dispatching rule, is a complete schedule. The schedule is feasible and the starting times and completion times of all the operations are determined. The output data of this procedure (and the input data for the next procedure) may contain the following information:

(i) the sequence of operations on each machine;

(ii) the start time and completion time of each operation;

(iii) the values of specific objective functions.

The output data does not have to contain all the data listed above; for example, it may include only the sequence of operations on each machine. The second procedure, i.e., the local search procedure, may have a routine that can compute the start and completion times of all operations given the structure of the problem and the sequences of the operations. ||

Another level of flexibility allows the user not only to set procedures up in parallel or in series, but to integrate the procedures in a more complex manner. When different procedures are integrated within one framework, they do not work independently from one another; the effectiveness of one procedure may depend on the input or feedback received from another. Consider a branch-and-bound procedure or a beam search procedure for a scheduling problem. At each node in the search tree, one has to obtain either a lower bound or an estimate for the total penalty that will be incurred by the jobs that have not yet been scheduled. A lower bound can often be obtained by assuming that the remaining jobs can be scheduled while allowing preemptions. A preemptive version of a problem is often easier to solve than its nonpreemptive counterpart and the optimal solution of the preemptive problem provides a lower bound for the optimal solution of the nonpreemptive version.

Another example of an integration of procedures arises in decomposition techniques. A machine-based decomposition procedure is typically a heuristic designed for a complicated scheduling problem with many subproblems. A framework for a procedure that is applicable to the main problem can be constructed. However, the user may want to be able to specify, knowing the particular problem or instance, which procedure to apply to the subproblem.

If procedures have to be integrated, then one often has to work within a general framework (sometimes also referred to as a control structure) in which one or more specific types of subproblems have to be solved many times. The user may want to have the ability to specify certain parameters within this framework. For example, if the framework is a search tree for a beam search, then the user would like to be able to specify the beam width as well as the filter width. The subproblem that has to be solved at each node of the search tree has to yield (with little computational effort) a good estimate for the contribution to the objective by those jobs that have not yet been scheduled.

The transfer of data between procedures in an integrated framework may be complicated. It may be the case that data concerning a subset of jobs or a subset of operations has to be transferred. It may also be the case that the machines are not available at all times. The positions of the jobs already scheduled on the various machines may be fixed, implying that the procedure that has to schedule the remaining jobs must know when the machines are still available. If there are sequence dependent setup times, then the procedure also has to know which job was the last one processed on each machine, in order to compute the sequence dependent setup time for the next job.

Example 18.3.2 (Integration of Procedures in a Branching Scheme)

Consider a branch-and-bound approach for $1 \mid s_{jk}, brkdwn \mid \sum w_j T_j$. The jobs are scheduled in a forward manner, i.e., a partial schedule consists of a sequence of jobs that starts at time zero. At each node of the branching tree a bound has to be established for the total weighted tardiness of the jobs still to be scheduled. If a procedure is called to generate a lower bound for all schedules that are descendants from a particular node, then the following input data has to be provided:

(i) the set of jobs already scheduled and the set of jobs still to be scheduled;

(ii) the time periods that the machine remains available;

(iii) the last job in the current partial schedule (in order to determine the sequence dependent setup time).

The output data of the procedure may contain a sequence of operations as well as a lower bound. The required output may be just a lower bound; the actual sequence may not be of interest.

If there are no setup times then a schedule can also be generated in a backward manner (since the value of the makespan is then known in advance). ||

Example 18.3.3 (Integration of Procedures in a Decomposition Scheme)

Consider a shifting bottleneck framework for a flexible job shop with at each workcenter a number of machines in parallel.

At each iteration a subset of the workcenters has already been scheduled and an additional workcenter must be scheduled. The sequences of the operations at the workcenters already scheduled imply that the operations of the workcenter to be scheduled in the subproblem is subject to delayed precedence constraints. When the procedure for the subproblem is called, a certain amount of data has to be transferred. These data may include:

(i) the release date and due date of each operation;

(ii) the precedence constraints between the various operations;

(iii) the necessary delays that go along with the precedence constraints.

The output data consists of a sequence of operations as well as their start times and completion times. It also contains the values of given performance measures.

It is clear that the type of information and the structure of the information is more complicated than in a simple concatenation of procedures. ||

These forms of integration of procedures have led to the development of so-called scheduling description languages. A description language is a high level language that enables a scheduler to write the code for a complex integrated algorithm using only a limited number of concise statements or commands. Each

statement in a description language involves the application of a relatively powerful procedure. For example, a statement may carry the instruction to apply a tabu-search procedure on a given set of jobs in a given machine environment. The input to such a statement consists of the set of jobs, the machine environment, the processing restrictions and constraints, the length of the tabu-list, an initial schedule, and the total number of iterations. The output consists of the best schedule obtained with the tabu-search procedure. Other statements may be used to set up various different procedures in parallel or concatenate two different procedures.

18.4 Reconfigurable Systems

The last two decades have witnessed the development of a large number of scheduling systems in industry and in academia. Some of these systems are application-specific, others are generic. In implementations application-specific systems tend to do somewhat better than generic systems that are customized. However, application-specific systems are often hard to modify and adapt to changing environments. Generic systems are usually somewhat better designed and more modular. Nevertheless, any customization of such systems typically requires a significant investment.

Considering the experience of the last two decades, it appears useful to provide guidelines that facilitate and standardize the design and the development of scheduling systems. Efforts have to be made to provide guidelines as well as system development tools. The most recent designs tend to be highly modular and object-oriented.

There are many advantages in following an object-oriented design approach for the development of a scheduling system. First, the design is modular, which makes maintenance and modification of the system relatively easy. Second, large segments of the code are reusable. This implies that two systems that are inherently different still may share a significant amount of code. Third, the designer thinks in terms of the behavior of objects, not in lower level detail. In other words, the object-oriented design approach can speed up the design process and separate the design process from its implementation.

Object oriented systems are usually designed around two basic entities, namely *objects* and *methods*. Objects refer to various types of entities or concepts. The most obvious ones are jobs and machines or activities and resources. However, a schedule is also an object and so are user-interface components, such as buttons, menus and canvasses. There are two basic relationships between object types, namely the *is-a* relationship and the *has-a* relationship. According to an is-a relationship one object type is a special case of another object type. According to a has-a relationship an object type may consist of several other object types. Objects usually carry along static information, referred to as attributes, and dynamic information, referred to as the state of the object. An object may have several attributes that are descriptors associated with the ob-

ject. An object may be in any one of a number of states. For example, a machine may be *busy*, *idle*, or *broken down*. A change in the state of an object is referred to as an *event*.

A method is implemented in a system by means of one or more operators. Operators are used to manipulate the attributes corresponding to objects and may result in changes of object states, i.e., events. On the other hand, events may trigger operators as well. The sequence of states of the different objects can be described by a state-transition or event diagram. Such an event diagram may represent the links between operators and events. An operator may be regarded as the way in which a method is implemented in the software. Any given operator may be part of several methods. Some methods may be very basic and can be used for simple manipulations of objects, e.g., a pairwise interchange of two jobs in a schedule. Others may be very sophisticated, such as an intricate heuristic that can be applied to a given set of jobs (objects) in a given machine environment (also objects). The application of a method to an object usually triggers an *event*.

The application of a method to an object may cause information to be transmitted from one object to another. Such a transmission of information is usually referred to as a message. Messages represent information (or content) that are transmitted from one object (for example, a schedule) via a method to another object (for example, a user interface display). A message may consist of simple attributes or of an entire object. Messages are transmitted when events occur (caused by the application of methods to objects). Messages have been referred to in the literature also as memos. The transmission of messages from one object to another can be described by a transition event diagram, and requires the specification of protocols.

A scheduling system may be object-oriented in its basic conceptual design and/or in its development. A system is object-oriented in its conceptual design if the design of the system is object-oriented throughout. This implies that every concept used and every functionality of the system is either an object or a method of an object (whether it is in the data or knowledge base, the algorithm library, the scheduling engine or the user interfaces). Even the largest modules within the system are objects, including the algorithm library and the user interface modules. A system is object-oriented in its development if only the more detailed design aspects are object-oriented and the code is based on a programming language with object-oriented extensions such as C++.

Many scheduling systems developed in the past have object-oriented aspects and tend to be object-oriented in their development. A number of these systems also have conceptual design aspects that are object-oriented. Some rely on inference engines for the generation of feasible schedules and others are constraint based relying on constraint propagation algorithms and search. These systems usually do not have engines that perform very sophisticated optimization.

Not many systems have been designed from top to bottom according to an object-oriented philosophy. Some of the aspects that are typically not object-oriented include:

(i) the design of scheduling engines,
(ii) the design of the user interfaces and
(iii) the specification of the precedence, routing and layout constraints.

Few existing engines have extensive libraries of algorithms at their disposal that are easily reconfigurable and that would benefit from a modular object-oriented design (an object-oriented design would require a detailed specification of operators and methods). Since most scheduling environments would benefit from highly interactive optimization, schedule generators have to be strongly linked to interfaces that allow schedulers to manipulate schedules manually. Still, object-oriented design has not had yet a major impact on the design of user interfaces for scheduling systems. The precedence constraints, the routing constraints, and the machine layout constraints are often represented by rules in a knowledge base and an inference engine must generate a schedule that satisfies the rules. However, these constraints can be modeled conceptually easily using graph and tree objects that then can be used by an object oriented scheduling engine.

18.5 Web-Based Scheduling Systems

With the ongoing development in information technology, conventional single-user stand-alone systems have become available in networks and on the Internet. Basically there are three types of web-based systems:

(i) information access systems,
(ii) information coordination systems,
(iii) information processing systems.

In information access systems, information can be retrieved and shared through the Internet, through EDI or through other electronic systems. The server acts as an information repository and distribution center, such as a homepage on the Internet.

In information coordination systems, information can be generated as well as retrieved by many users (clients). The information flows go in many directions and the server can synchronize and manage the information, such as in project management and in electronic markets.

In information processing systems the servers can process the information and return the results of this processing to the clients. In this case, the servers function as application programs that are transparent to the users.

Web-based scheduling systems are information processing systems that are very similar to the interactive scheduling systems described in previous sections, except that a web-based scheduling system is usually a strongly distributed system. Because of the client-server architecture of the Internet, all the important components of a scheduling system, i.e., the database, the engine, and the user interface, may have to be adapted. The remaining part of this section focuses on some of the typical design features of web-based scheduling systems.

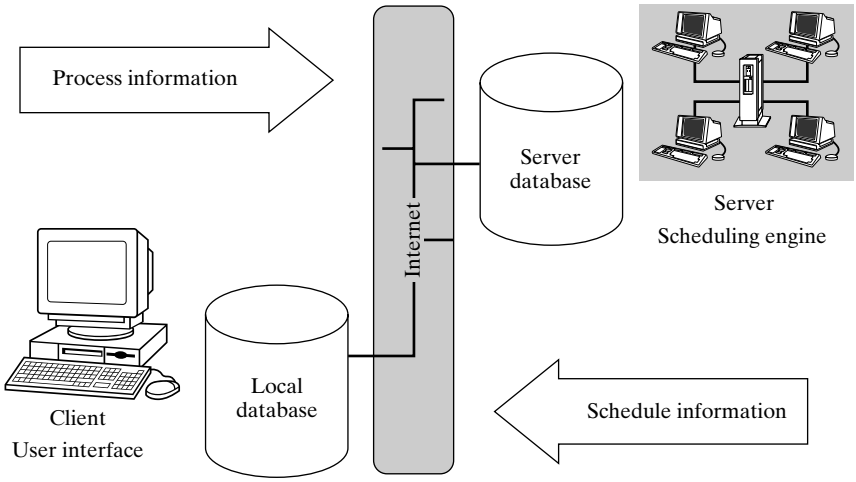


Fig. 18.3 Information flow between client and server

The advantages of having servers that make scheduling systems available on the web are the following. First, the input-output interfaces (used for the graphical displays) can be supported by local hosts rather than by servers at remote sites. Second, the server as well as the local clients can handle the data storage and manipulation. This may alleviate the workload at the server sites and give local users the capability and flexibility to manage the database. Third, multiple servers can collaborate on the solution of large-scale and complicated scheduling problems. A single server can provide a partial solution and the entire problem can be solved using distributed computational resources.

In order to retain all the functions inherent in an interactive scheduling system, the main components of a system have to be restructured in order to comply with the client-server architecture and to achieve the advantages listed above. This restructuring affects the design of the database, the engine as well as the user interface.

The design of the database has the following characteristics: The process manager as well as the scheduling manager reside at the servers. However, some data can be kept at the client for display or further processing. Both the Gantt chart and the dispatch lists are representations of the solution generated by the engine. The local client can cache the results for fast display and further processing, such as editing. Similarly, both the server and the client can process the information. Figure 18.3 exhibits the information flow between the server and local clients. A client may have a general purpose database management system (such as Sybase or Excel) or an application-specific scheduling database for data storage and manipulation.

The design of the scheduling engine has the following characteristics: A local client can select for a problem he has to deal with an algorithm from a library

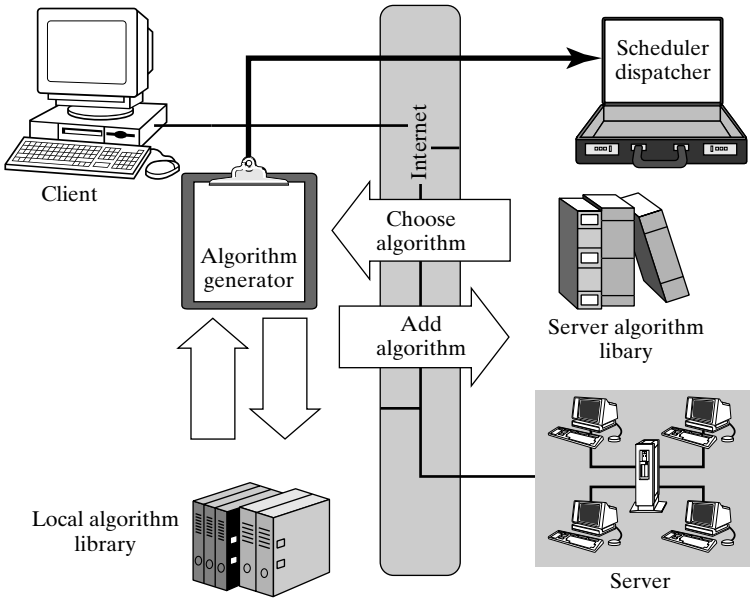


Fig. 18.4 Process of constructing new procedures

that resides at a remote server. Often, there is no algorithm specifically designed for his particular scheduling problem and he may want to create a composite procedure using some of the algorithms that are available in the library. The server or client algorithm generator may function as a workplace for users to create new procedures. Figure 18.4 shows how a new composite procedure can result in a new algorithm that then can be included in both the server and the client libraries. This local workplace can speed up the process of constructing intermediate and final composite methods and extend the server and client libraries at the same time.

The Internet also has an effect on the design of the user interfaces. Using existing Internet support, such as HTML (HyperText Markup Language), Java, Java script, Perl and CGI (Common Gateway Interface) functions, the graphical user interfaces of scheduling systems can be implemented as library functions at the server sites. Through the use of appropriate browsers, users can enter data or view schedules with a dynamic hypertext interface. Moreover, a user can also develop interface displays that link server interface functions to other applications. In Figure 18.3 it is shown how display functions can be supported either by remote servers or by local clients.

Thus it is clear that servers can be designed in such a way that they can help local clients solve their scheduling problems. The local clients can manipulate data and construct new scheduling methods. The servers function as regular

interactive scheduling systems except that now they can be used in a multi-user environment on the Internet.

Web-based scheduling systems can be used in several ways. One way is based on personal customization and another on unionization. Personal customization implies that a system can be customized to satisfy an individual user's needs. Different users may have different requirements, since each has his own way of using information, applying scheduling procedures, and solving problems. Personalized systems can provide shortcuts and improve system performance. Unionization means that a web-based scheduling system can be used in a distributed environment. A distributed system can exchange information efficiently and collaborate effectively in solving hard scheduling problems.

With the development of Internet technology and client-server architectures, new tools can be incorporated in scheduling systems for solving large-scale and complicated problems. It appears that web-based systems may well lead to viable personalized interactive scheduling systems.

18.6 Discussion

Many teams in industry and academia are currently developing scheduling systems. The database (or object base) management systems are usually off-the-shelf, developed by companies that specialize in these systems, e.g., Oracle. These commercial databases are typically not specifically geared for scheduling applications; they are of a more generic nature.

Dozens of software development and consulting companies specialize in scheduling applications. They may specialize even in certain niches, e.g., scheduling applications in the process industries or in the microelectronics industries. Each of these companies has its own systems with elaborate user interfaces and its own way of doing interactive optimization.

Research and development in scheduling algorithms and in learning mechanisms will most likely only take place in academia or in large industrial research centers. This type of research needs extensive experimentation; software houses often do not have the time for such developments.

In the future, the Internet may allow for the following types of interaction between software companies and universities that develop systems and companies that need scheduling services (clients). A client may use a system that is available on the web, enter its data and run the system. The system gives the client the values of the performance measures of the solution generated. However, the client cannot yet see the schedule. If the performance measures of the solution are to the liking of the client, then he may decide to purchase the solution from the company that generated the schedule.

Exercises

18.1. One way of constructing robust schedules is by inserting idle times. Describe all the factors that influence the timing, the frequency and the duration of the idle periods.

18.2. Consider all the nonpreemptive schedules on a single machine with n jobs. Define a measure for the "distance" (or the "difference") between two schedules.

(a) Apply the measure when the two schedules consist of the same set of jobs.

(b) Apply the measure when one set of jobs has one more job than the other set.

18.3. Consider the same set of jobs as in Example 18.1.1. Assume that there is a probability p that the machine needs servicing starting at time 2. The servicing takes 10 time units.

(a) Assume that neither preemption nor resequencing is allowed (i.e., after the servicing has been completed, the machine has to continue processing the job it was processing before the servicing). Determine the optimal sequence(s) as a function of p.

(b) Assume preemption is not allowed but resequencing is allowed. That is, after the first job has been completed the scheduler may decide not to start the job he originally scheduled to go second. Determine the optimal sequence(s) as a function of p.

(c) Assume preemption as well as resequencing are allowed. Determine the optimal sequence(s) as a function of p.

18.4. Consider two machines in parallel that operate at the same speed and two jobs. The processing times of each one of the two jobs is equal to one time unit. At each point in time each machine has a probability 0.5 of breaking down for one time unit. Job 1 can only be processed on machine 1 whereas job 2 can be processed on either one of the two machines. Compute the expected makespan under the Least Flexible Job first (LFJ) rule and under the Most Flexible Job first (MFJ) rule.

18.5. Consider a single machine scheduling problem with the jobs being subject to sequence dependent setup times. Define a measure of job flexibility that is based on the setup time structure.

18.6. Consider the following instance of a single machine with sequence dependent setup times. The objective to be minimized is the makespan. There are 6 jobs. The sequence dependent setup times are specified in the table below.

k	0	1	2	3	4	5	6
s_{0k}	-	1	$1+\epsilon$	K	$1+\epsilon$	$1+\epsilon$	K
s_{1k}	K	-	1	$1+\epsilon$	K	$1+\epsilon$	$1+\epsilon$
s_{2k}	$1+\epsilon$	K	-	1	$1+\epsilon$	K	$1+\epsilon$
s_{3k}	$1+\epsilon$	$1+\epsilon$	K	-	1	$1+\epsilon$	K
s_{4k}	K	$1+\epsilon$	$1+\epsilon$	K	-	1	$1+\epsilon$
s_{5k}	$1+\epsilon$	K	$1+\epsilon$	$1+\epsilon$	K	-	1
s_{6k}	1	$1+\epsilon$	K	$1+\epsilon$	$1+\epsilon$	K	-

Assume K to be very large. Define as the neighbourhood of a schedule all schedules which can be obtained through an adjacent pairwise interchange.

(a) Find the optimal sequence.

(b) Determine the makespans of all schedules that are neighbors of the optimal schedule.

(c) Find a schedule, with a makespan less than K, of which all neighbors have the same makespan. (The optimal sequence may be described as a "brittle" sequence, while the last sequence may be described as a more "robust" sequence.)

18.7. Consider a flow shop with limited intermediate storages that is subject to a cyclic schedule as described in Section 16.2. Machine i now has at the completion of each operation a probability p_i that it goes down for an amount of time x_i.

(a) Define a measure for the congestion level of a machine.

(b) Suppose that originally there are no buffers between machines. Now a total of k buffer spaces can be inserted between the m machines and the allocation has to be done in such a way that the schedules are as robust as possible. How does the allocation of the buffer spaces depend on the congestion levels at the various machines?

18.8. Explain why rote learning is an extreme form of case-based reasoning.

18.9. Describe how a branch-and-bound approach can be implemented for a scheduling problem with m identical machines in parallel, the jobs subject to sequence dependent setup times and the total weighted tardiness as objective. That is, generalize the discussion in Example 18.3.2 to parallel machines.

18.10. Consider Example 18.3.3 and Exercise 18.9. Integrate the ideas presented in an algorithm for the flexible job shop problem.

18.11. Consider a scheduling description language that includes statements that can call different scheduling procedures for a scheduling problem with m identical machines in parallel, the total weighted tardiness objective and the n jobs released at different points in time. Write the specifications for the input

and the output data for three statements that correspond to three procedures of your choice. Develop also a statement for setting the procedures up in parallel and a statement for setting the procedures up in series. Specify for each one of these last two statements the appropriate input and output data.

18.12. Suppose a scheduling description language is used for coding the shifting bottleneck procedure. Describe the type of statements that are required for such a code.

Comments and References

There is an extensive literature on scheduling under uncertainty (i.e., stochastic scheduling). However, the literature on stochastic scheduling, in general, does not address the issue of robustness per se. But robustness concepts did receive special attention in the literature; see, for example, the work by Leon and Wu (1994), Leon, Wu and Storer (1994), Mehta and Uzsoy (1999), Wu, Storer and Chang (1991), Wu, Byeon and Storer (1999), and Policella, Cesta, Oddi and Smith (2005, 2007). For an overview of research in reactive scheduling, see the excellent survey by Smith (1992) and the framework presented by Vieira, Herrmann and Lin (2003). For more detailed work on reactive scheduling and rescheduling in job shops, see Bierwirth and Mattfeld (1999) and Sabuncuoglu and Bayiz (2000). For applications of robustness and reactive scheduling in the real world, see Elkamel and Mohindra (1999) and Oddi and Policella (2005).

Research on learning mechanisms in scheduling systems started in the late eighties; see, for example, Shaw (1988), Shaw and Whinston (1989), Yih (1990), Shaw, Park, and Raman (1992), and Park, Raman and Shaw (1997). The parametric adjustment method for the ATCS rule in Example 18.2.1 is due to Chao and Pinedo (1992). The book by Pesch (1994) focuses on learning in scheduling through genetic algorithms (classifier systems). For a description of the reinforcement approach applied to job shop scheduling, see Zhang and Dietterich (1995). For more general overviews of machine learning in production scheduling, see Aytug, Bhattacharyya, Koehler and Snowdon (1994) and Priore, de la Fuente, Gomez and Puente (2001).

A fair amount of development work has been done recently on the design of adaptable scheduling engines. Akkiraju, Keskinocak, Murthy and Wu (1998, 2001) discuss the design of an agent based approach for a scheduling system developed at IBM. Feldman (1999) describes in detail how algorithms can be linked and integrated and Webster (2000) presents two frameworks for adaptable scheduling algorithms.

The design, development and implementation of modular or reconfigurable scheduling systems is often based on objects and methods. For objects and methods, see Booch (1994), Martin (1993), and Yourdon (1994). For modular design with regard to databases and knowledge bases, see, for example, Collinot, LePape and Pinoteau (1988), Fox and Smith (1984), Smith (1992), and Smith,

Muscettola, Matthys, Ow and Potvin (1990)). For an interesting design of a scheduling engine, see Sauer (1993). A system design proposed by Smith and Lassila (1994) extends the modular philosophy for scheduling systems farther than any previous system. This is also the case with the approach by Yen (1995) and Pinedo and Yen (1997).

The paper by Yen (1997) contains the material concerning web-based scheduling systems presented in Section 18.5.

Chapter 19

Examples of System Designs and Implementations

From the previous chapters it is evident that there are many different types of scheduling problems. It is not likely that a system can be designed in such a way that it could be made applicable to any scheduling problem with only minor customization. This suggests that there is room as well as a need for many different scheduling systems. The variety of available platforms, databases, Graphic User Interfaces (GUI's) and networking capabilities enlarges the number of possibilities even more.

This chapter describes the architectures and implementations of seven scheduling systems. The first section describes the Production Planning and Detailed Scheduling System (PP/DS) that is part of the Advanced Planning and Optimization (APO) software developed by SAP, which is a company headquartered in Germany. The PP/DS system is a flexible system that can be adapted easily to many industrial settings. The second system had been developed at IBM's T.J. Watson Research Center. This system had been installed at a number of sites, primarily in the paper industry. The third system is the Production Scheduler system developed by i2 Technologies, which is based in Dallas, Texas. Currently, the i2 Production Scheduler is one of the more widely used systems in industry. The fourth system is a commercial system developed by Taylor Scheduling Software. This system is quite generic and can be adapted to many

M.L. Pinedo, *Scheduling*, DOI: 10.1007/978-0-387-78935-4_19,

different manufacturing settings. The fifth section describes a scheduling architecture developed by Advanced Micro Devices for their semiconductor manufacturing plants. The sixth system is an application-specific system developed by Cybertec in Italy for a facility that produces jars of yogurt. The seventh and last system is an academic system that has been developed at New York University (NYU) for educational purposes. This system has been in use for several years at many universities all over the world.

19.1 SAP's Production Planning and Detailed Scheduling System

SAP has been from the outset a company that specializes in the development of Enterprise Resource Planning (ERP) systems. The ERP2005 system is still one of their most important products. In 1998 the company started to develop decision support systems for manufacturing as well as for service industries. For example, they decided to develop their own supply chain planning and scheduling software rather than depend on alliances with third parties. This development resulted ultimately in a division that creates a suite of business solutions for Supply Chain Management (SCM) applications. This suite of solutions is referred to as SAP SCM. The supply chain planning and scheduling software is referred to in SAP as Applied Planning and Optimization (APO).

APO provides a set of specially tailored optimization routines that can be applied to all aspects of supply chain planning and scheduling. APO offers the following planning and scheduling steps:

(i) Supply Network Planning,
(ii) Production Planning and Material Requirements Planning, and
(iii) Detailed Scheduling.

The Supply Network Planning step (which is equivalent to a crude form of production planning) generates a production plan across the different production facilities (including subcontractors) in order to meet (customer) demand in the required time frames and according to the standards expected by the customer. This is accomplished either through their Capable-To-Match (CTM) planning procedure or through their optimizer. The CTM procedure uses constraint based heuristics to conduct multi-site checks of production capacities and transportation capabilities based on predefined supply categories and demand priorities. The objective of a CTM planning run is to generate a feasible solution that meets all the demands. The CTM planning run is powered by the CTM engine, which matches the prioritized demands to the available supplies in two phases. First, it builds the CTM application model based on the master data that have been entered. Second, it matches the demands to the supplies on a first come first served basis, taking production capacities and transportation capabilities into account. The Optimizer does a rough cut planning over a medium and long term horizon, based on time buckets; it specifies the demands

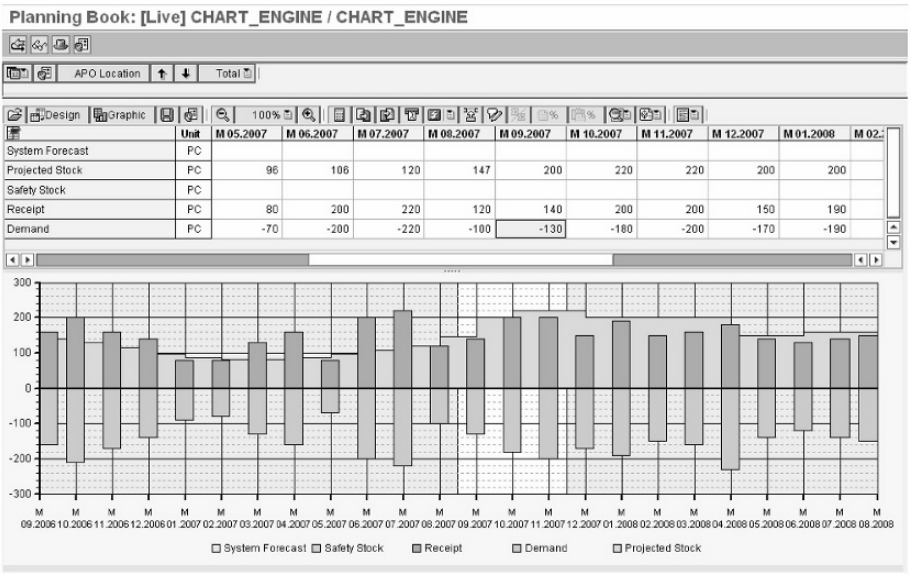


Fig. 19.1 SAP Supply Network Planning User Interface

on the resources (machines, people, production resource tools) and material requirements. Real-time data, solvers, and high supply chain visibility support the planner's decision-making process. The medium term planning problem can, for example, be solved through a Linear Programming relaxation. This LP relaxation can be solved with the CPLEX LP solver of the ILOG library. In order to deal with integrality constraints it has a discretization heuristic that can take into account the actual meaning of each one of the integer decision variables. After solving the LP relaxation, the variables are stepwise discretized using again in each step an LP relaxation. The discretization process is done gradually: lot sizes for later time buckets are discretized later. The planning problem may include linear constraints as well as integer constraints. Linear constraints may be necessary because of due date constraints, maximum delay constraints, storage capacity constraints, and so on. The integer constraints may be necessary because of minimal lot sizes, full truck loads, and so on. Such optimization problems are modeled as Mixed Integer Programs (MIPs). A user interface for the Supply Network Planning is depicted in Figure 19.1.

The Production Planning and Material Requirements Planning step is an important part of the production planning process. It generates replenishment schedules for all manufactured components, intermediates, purchased parts, and raw materials. This step sets due dates for production orders and purchase requisitions through lead time scheduling, depending on buffers, processing times, lot-sizing rules, and so on.

The Detailed Scheduling step generates good (and perhaps even optimal) job schedules that can be released for production. Scheduling heuristics and solvers take into account constraints and costs to optimally schedule the set of jobs under consideration, based on the business objectives. The most popular solvers in this step are Genetic Algorithms (GA). What-if simulations and evaluations of the order sequences provide the scheduler with a certain amount of flexibility and control. Critical resource situations can be adjusted either automatically or manually via a well-designed user interface, see Figure 19.2. The detailed scheduling step can be applied in process industries as well as in discrete manufacturing industries. Customer-specific scheduling needs can be served with individual heuristics and optimizers to extend the standard scheduling tools with user and industry-specific components (like for example trim optimization algorithms in mill industries). These individual heuristics and algorithms can be called directly from the Detailed Scheduling user interfaces. Combined with the standard optimizers and heuristics, they form an integrated scheduling system. The detailed scheduling problem is modeled in its most generic form as a so-called Multi-Mode Resource Constrained Project Scheduling Problem with minimum and maximum time lags. Maximum time constraints such as deadlines or shelf life (expiration dates), storage capacities, sequence dependent setup times, precedence constraints, processing interruptions due to breakdowns, and objectives such as the minimization of setup times, setup costs, due date delays can all be included.

The Production Planning and Detailed Scheduling steps are typically considered one module and are referred to as the APO-PP/DS module.

APO has at its disposal a tool kit that contains a suite of algorithms and heuristics, namely:

(i) Genetic Algorithms,
(ii) Multi-Level Planning Heuristics, and
(iii) Manual Planning Heuristics (including drag and drop).

The Genetic Algorithms (GA) are based on the evolutionary approach. The genetic representation contains the schedule information that is used by a fast scheduler for generating new solutions. Because its scheduler uses no dynamic constraint propagation and only limited backtracking this approach has limitations on the use of maximal time constraints.

The performance of each type of algorithm depends on the setting as well as on the instance under consideration. The user of the system may select the most appropriate algorithm after some experimental analysis.

In its generic framework APO provides an option to combine the algorithms and heuristics above while applying at the same time one or more decomposition techniques. The decomposition techniques enable the user to partition a problem instance according to

(i) time,
(ii) workcenter or machine,

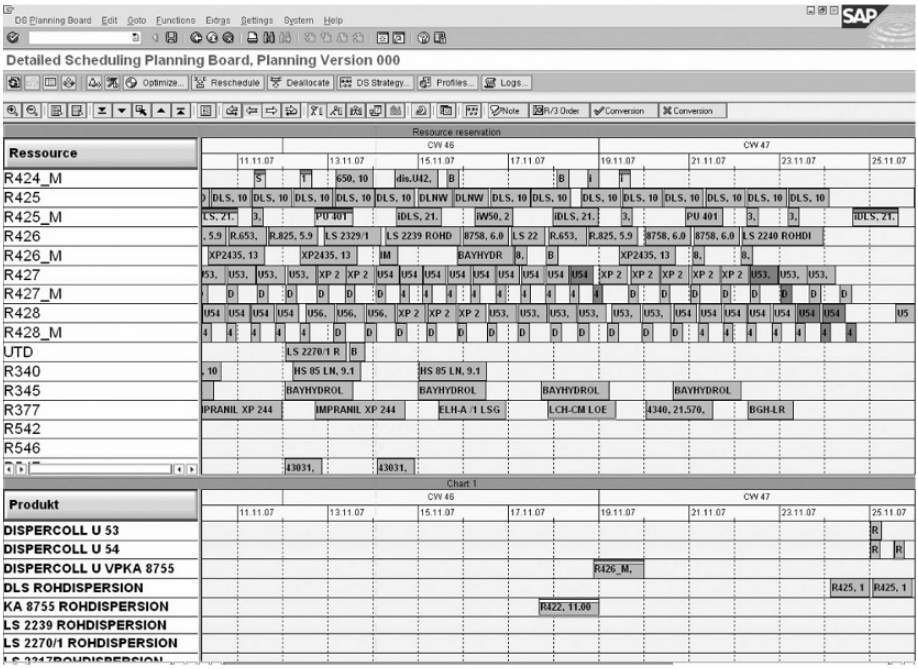


Fig. 19.2 SAP Detailed Scheduling User Interface

(iii) product type or job,

(iv) job priority.

The decomposition techniques enable a user also to scale the neighbourhood up or down, i.e., the user can adjust the decomposition width. APO has a feature that allows fine-tuning of the decomposition width.

Furthermore, there is a parallelization option, either through grid computing or through multi-threading. This is often required because of the size of the problems and the tightness of the time constraints. There is also an explanation tool that tries to explain to the user some of the characteristics of the schedule that had been generated (why there is a delay, etc.).

The biggest problem size solved with APO-PP/DS involved an instance with more than 1,000,000 jobs and 1,000 resources. The procedure used for this instance was based on a Genetic Algorithm.

19.2 IBM's Independent Agents Architecture

Production scheduling and distribution of paper and paper products is an extremely complex task that must take into account numerous objectives and constraints. The complexity of the problem is compounded by the interactions

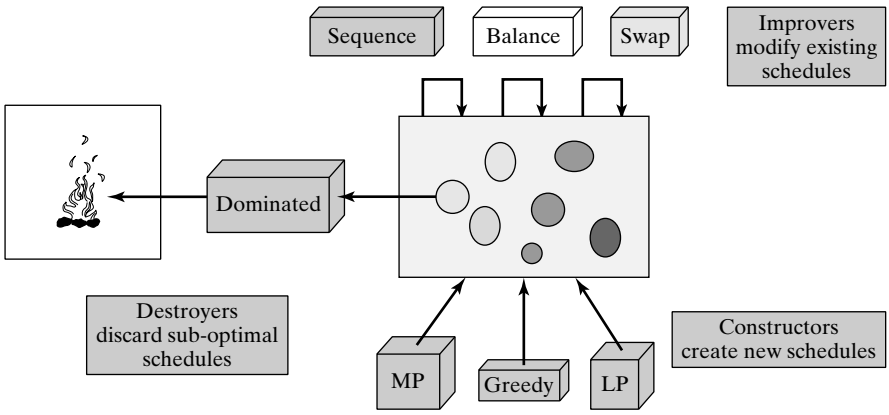


Fig. 19.3 Essential Features of an IBM A-Team

between the schedules of consecutive stages in the production process. Often, a good schedule for the jobs at one stage is a terrible one at the next stage. This makes the overall optimization problem hard.

IBM developed a cooperative multi-objective decision-support system for the paper industry. The system generates multiple scheduling alternatives using several types of algorithms that are based on linear and integer programming, network flow methods and heuristics. In order to generate multiple solutions and facilitate interactions within the scheduler, IBM's system has been implemented using the agent-based Asynchronous Team (A-Team) architecture in which multiple scheduling methods cooperate in generating a shared population of schedules (see Figures 19.3 and 19.4).

There are three types of agents in an A-Team:

(i) Constructors,
(ii) Improvers, and
(iii) Destroyers.

Constructors have as input a description of the problem. Based on this description they generate new schedules. Improvers attempt to improve upon the schedules in the current population by modifying and combining existing schedules. Destroyers attempt to keep the population size in check by removing bad schedules from the population. A human scheduler can interact with the A-Team by assuming the role of an agent; the scheduler can add new schedules, modify and improve existing schedules, or remove existing schedules.

One of the more important algorithms in the system considers a set of jobs in an environment with a number of non-identical machines in parallel. The jobs have due dates and tardiness penalties and are subject to sequence dependent setup times. The problem setting is similar to the $Rm \mid r_j, s_{jk} \mid \sum w_j T_j$ model. For this particular problem IBM's algorithmic framework provides constructor agents as well as improver agents.

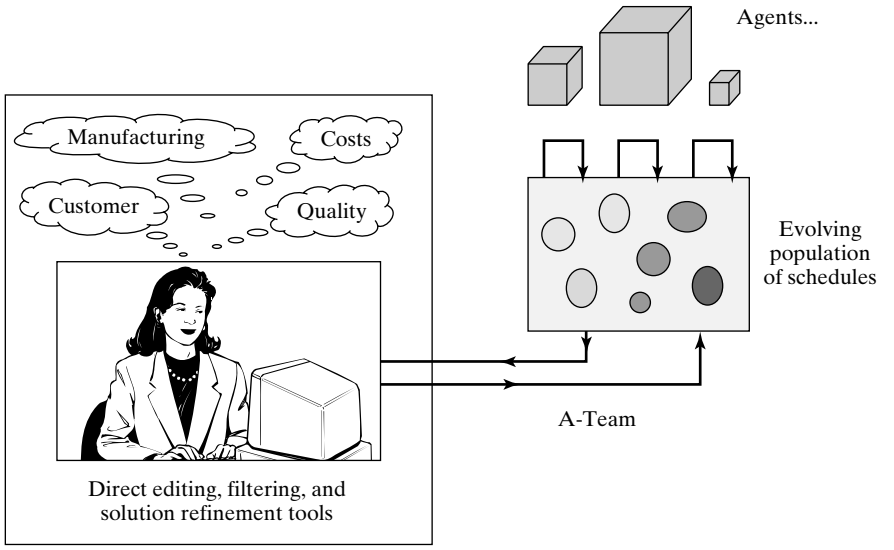


Fig. 19.4 Interactive Optimization with IBM's A-Team

There are basically two types of constructor agents for this problem, namely:

(i) Agents that allocate jobs to machines and do an initial sequencing.

(ii) Agents that do a thorough sequencing of all the jobs on a given machine.

The algorithmic framework used for the job-machine allocation and initial sequencing is called Allocate-and-Sequence. The objective is to optimize the allocation of the jobs to the machines based on due dates, machine load balance, and transportation costs considerations (the machines may be at various different locations).

The main ideas behind the Allocate-and-Sequence heuristic include a sorting of all the jobs according to various criteria, e.g., due date, processing time, tardiness penalty. The job at the top of the list of remaining jobs is allocated to the machine with the lowest index $I_1(i, j, k)$ of the machine-job (i, j) pair. The index $I_1(i, j, k)$ of job j on machine i is computed by considering several combinations of job properties and machine properties (assuming that job j will be processed on machine i immediately after job k). Some examples of indices used in actual implementations are:

(i) The processing time of job j on machine i plus the setup time required between jobs k and j.

(ii) The completion time of job j.

(iii) The weighted tardiness of job j.

(iv) The absolute difference between the due date of job j and its completion time on machine i.

The Allocate-and-Sequence heuristic can be summarized as follows.

Algorithm 19.2.1 (Allocate and Sequence Heuristic)

Step 1.

Rank the jobs according to the given sort criteria.

Step 2.

Select the next job on the list.
Compute the machine-job index for all machines that can process this job.
Assign the job to the machine with the lowest machine-job index. ||

The goal of the second type of constructor agent is to sequence the jobs on a given machine based on their due dates and then group jobs of the same kind in batches (in the case of paper machine scheduling this is analogous to the aggregation of orders for items of the same grade). The algorithmic framework used for sequencing jobs on a given machine is called Single-Dispatch. Assume the set of jobs allocated to machine i is already known (as an output of Allocate-and-Sequence). Call this set U_i.

An index $I_2(j,k)$ for a job j in U_i can be computed by considering several combinations of job properties. Some of the index computations use the slack, which is defined as the difference between the due date of the job and the earliest possible completion time. Other examples of this index used in implementations of the Single-Dispatch framework include the processing time of job j on machine i, the setup time required between jobs k and j, the tardiness of job j, and a weighted combination of any of these measures with the setup time.

The Single-Dispatch heuristic can be summarized as follows.

Algorithm 19.2.2 (Single-Dispatch Heuristic)

Step 1.

Let the set U_i initially be equal to all jobs allocated to machine i.

Step 2.

For each job in U_i, compute the index $I_2(j,k)$ that corresponds to the scheduling of job j as the next one on machine i.

Step 3.

Schedule the job with the smallest index as the next job and remove the job from U_i.
If U_i is empty, then STOP;
otherwise go to Step 2. ||

Improver agents consider the schedules in the current population and attempt to improve on them in various different ways. For example, an improver agent may either move a single job to improve its tardiness, or aggregate batches of jobs in order to decrease the number of batches and reduce the total setup time.

A job can be moved to a new position on the same machine, or it can be moved to a different machine. The goal of any exchange may be to improve one

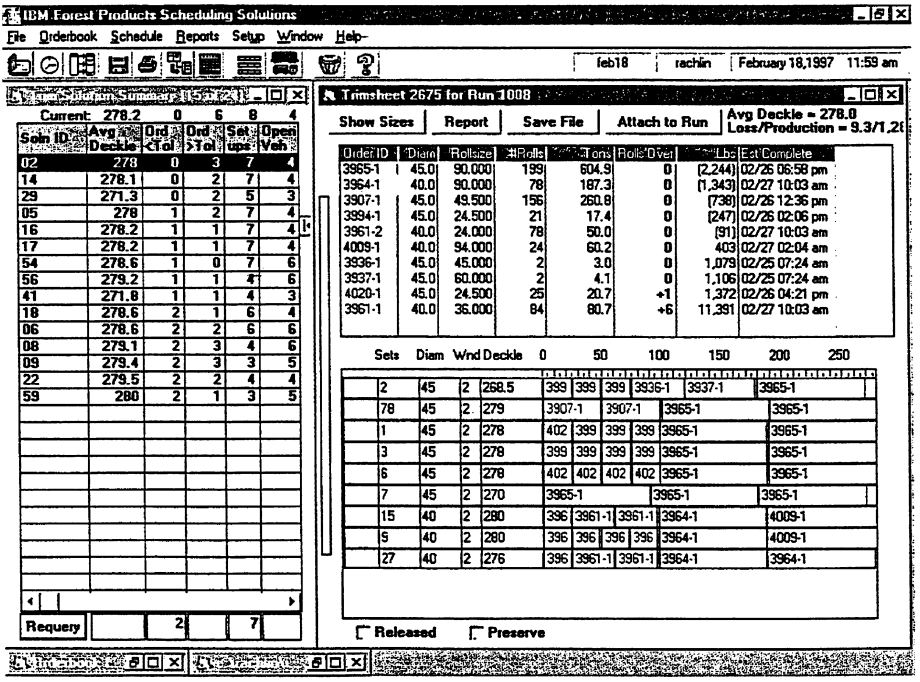


Fig. 19.5 IBM Decision Support User Interface

particular objective, such as the total weighted tardiness, or a combination of several objectives such as machine load balance and total weighted tardiness. Each agent selectively picks a solution to work on based on certain criteria. For example, a tardiness improver agent picks a schedule from the population that needs an improvement with regard to tardiness. A load balancing agent picks a schedule of which the load is unevenly balanced and attempts to rebalance, and so on.

This scheduling system has been implemented in several paper mills. Figure 19.5 shows a user interface of such an implementation.

19.3 i2's Production Scheduler

This section describes the i2 Production Scheduler, which is a commercial scheduling system developed by i2 Technologies. This product has in the past also been referred to as the i2 Tradematrix Production Scheduler. The i2 Production Scheduler is one of several solutions in the Scheduler suite of products offered by i2 to meet the scheduling needs of various different manufacturing industries. The Scheduler suite includes Sequencer and Master Scheduler for

automotive and industrial companies, Mill Scheduler for the metals industries and Paper Mill Manager for the paper and board industries. All these products are part of a complete and integrated supply chain management solution. These products are designed to capture the specific needs and complexities of the different manufacturing environments and use a patented Genetic Algorithm (GA) technology to generate optimized manufacturing schedules. This technology has been licensed for hundreds of implementations and enables representation of a wide range of production situations, easily adapting to new and changing constraints. This section describes the elements of the i2 Production Scheduler model, the key functionalities and system requirements of Production Scheduler, the functionalities of the user interfaces and some implementation examples.

Production Scheduler is a configurable scheduling system that can produce detailed schedules for multi-stage manufacturing processes. The patented technology combines a fast constraint calculation engine and an optimization algorithm (GA) to schedule all manufacturing stages simultaneously. Production Scheduler has the capability to model either discrete (batch) or continuous processes, with multiple resources at each stage. The core product is Java-based, with an easily customizable, proprietary optimal scheduler language defining much of its functionality. The manufacturing model data is maintained in a database, typically MS Access or Oracle. The product is configured to run in client-server mode, with a Windows NT client and either a Windows NT or Unix operating system for the server. Demand data may be entered directly into Production Scheduler's database, or imported from an ERP or customer order entry system.

Production Scheduler is a tool for detailed scheduling that is appropriate for a variety of manufacturing industries, including consumer goods, textile, apparel, footwear, automotive stamping plants, gear and transmission shops, pharmaceuticals and chemicals. A given manufacturing process can be modeled in such a way that the various business rules, or constraints, that specify how the process is run, are captured. Typical constraints include capacity limits, setup times and costs, due date requirements, Work-In-Process (WIP) minimization, labor restrictions and material availability. The Production Scheduler generates optimal schedules that balance all of these different constraints while generating the best feasible solution.

A model is basically a data representation of the rules that govern the given manufacturing process. This representation includes:

(i) the products and component parts that the factory produces (items and item families);
(ii) the physical process of production (Bills of Material (BOMs), routings, operations, and manufacturing resources);
(iii) production preferences, policies and requirements (constraints).

Given the basic building blocks of the model, Production Scheduler generates production schedules that satisfy the demand (in the form of work orders),

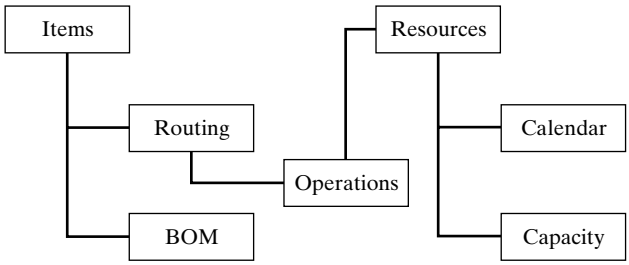


Fig. 19.6 Relationships between the Modeling Elements in i2's Production Scheduler

subject to the operating constraints specified. The basic Production Scheduler model contains the following elements:

Items: An item is a part, component, subassembly or finished good that is produced or consumed by the factory. In a Production Scheduler model an item can be a finished part, an intermediate part, or raw material. The Bill Of Material is a list of intermediate items that are needed for producing a given item.

Operations: An operation is a production activity that is carried out by one or more resources (e.g., a mixing tank and the labor resource required in the mixing process). It is one step in a route that produces an item. An operation may be partitioned into time segments that all share a common resource (e.g., the mixing tank is the common resource used throughout the mixing process, but the labor is used just initially to fill the tank).

Resources: A resource can be a machine, a tool, a labor pool or a storage container that is needed for processing an item during an operation.

Routes: A route is a sequence of one or more operations that produce one or more items.

Capacity Profiles: A capacity profile specifies the capacity of a resource over time.

Calendars: A calendar specifies the days and times a resource is unavailable for processing an item. It specifies which regions of the defined capacity calendar are unavailable for processing. If this unavailability calendar is not defined, then the resource is assumed to be available all day every day throughout the scheduling horizon.

Figure 19.6 depicts the relationships between the modelling elements described above. To generate a good schedule, Production Scheduler uses a set of scheduling criteria and constraints that enforce preferences and policies and impose physical limitations on the operations of the factory. Production Scheduler maintains a suite of common constraints to handle requirements such as:

(i) tardiness objectives,

(ii) minimization of WIP,
(iii) enforcement of routing constraints,
(iv) selection of resources for given operations,
(v) setup times and resource costs due to changeovers.

Production Scheduler has the flexibility to capture process limitations and constraints across a range of diverse industries from automotive to, for example, pharmaceutical. This may include upper bounds on the number of product changeovers on a set of resources within a given time period, or constraints on the amount of time a certain item may be held in a storage vessel before it begins to spoil. The list above is only a small sample of the large library of common constraints maintained within Production Scheduler, enabling easy implementation of many detailed and complex manufacturing constraints.

Constraints may be regarded as either *hard* or *soft*. Hard constraints represent rules that, if violated, could result in a production problem. Soft constraints, when violated, results in a feasible schedule that is less than ideal (see Appendix C).

Given a set of constraints with associated penalties Production Scheduler uses a genetic algorithm (GA) to generate schedules. For a given model, Production Scheduler finds the best scheduling strategy by determining trade-offs among soft constraints. Therefore, while Production Scheduler never violates a hard constraint when it schedules a task, it will invariably violate one or more soft constraints. However, the optimizer typically generates a schedule that minimizes the total number of soft constraint violations. Assigning violation penalties to constraints is an important aspect of the fine-tuning of a given model and gives an indication of the relative priorities of the various conflicting rules in a typical manufacturing environment.

In addition to the basic set of constraints, Production Scheduler is configured for incorporating customized constraints easily in order to model the unique characteristics of any particular implementation.

Production Scheduler has two User Interfaces (UIs). The first is the Modelling UI, through which most of the model can be specified (resources, resource calendars, capacity profiles, items, operations, routes, and constraints). While defining modelling elements, Production Scheduler provides the flexibility to customize the names of the resources and the constraints, as well as the violation messages displayed when constraints are violated. Once a satisfactory model has been built and the demand has been imported, all interactions with the model occur through the Scheduling UI, presented in Figure 19.7.

The Scheduling UI consists of three grids (shown on the left of the screen), three display panes (shown at the bottom of the screen) and the schedule board (Gantt chart). This part of the UI is quite frequently used in practice and is highly interactive. It contains a lot of useful information and tools designed to make the job of coming up with a good schedule easy and intuitive.

The three grids display the following three types of information:

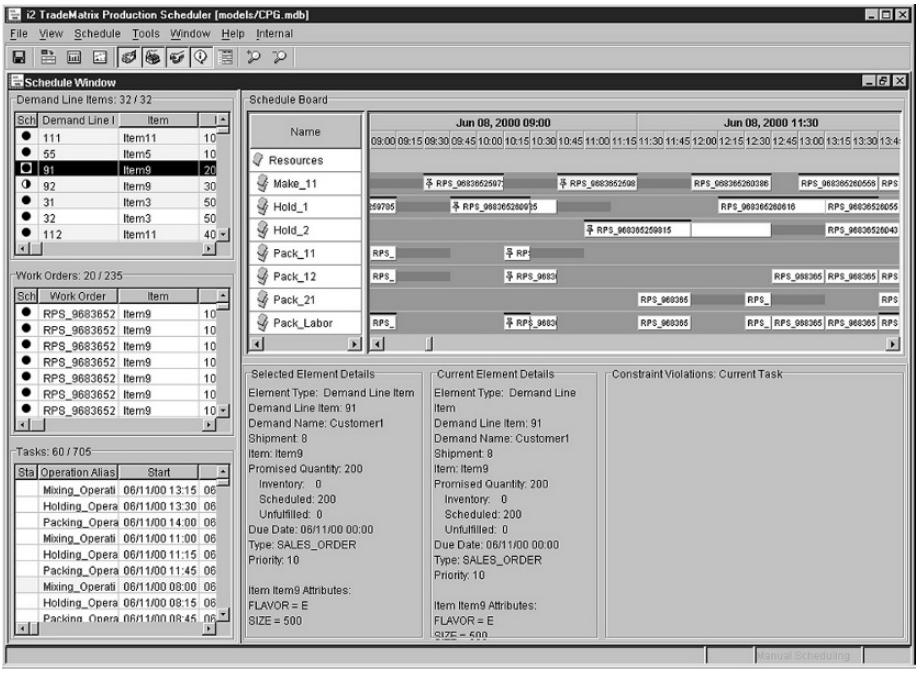


Fig. 19.7 i2's Production Scheduler User Interface

Demand Line Items: A list of all the current demand, along with promised quantities, current inventory levels, due dates and priorities.

Work Orders: The production scheduler aggregates demand based on the model-defined manufacturing quantity. Each demand line item may be broken down into a number of work orders, or aggregated with other demand line items (depending on the relative size of the demand quantity compared with the manufacturing quantity).

Tasks: Each work order has a number of tasks that must be scheduled in order to satisfy the demand. Each individual task represents an operation in the routing which produces the final item.

Each one of the grids can be manipulated (through sorting and filtering mechanisms) to help the user organize and display information as necessary.

The three display panes also provide an abundance of information: The first display pane can provide selected details concerning any element; when one clicks on a demand line item, work order or task, this pane displays information about that element. The second display pane provides details concerning the current element; this pane provides details regarding the object (demand line item, work order, task, etc.) on which the cursor is positioned. The third display pane shows constraint violations associated with the manual or automatic

scheduling of a given task. The violation message consists of the penalty points associated with the constraint, violation information, the constraint type and the specific constraint name.

The Schedule Board (or Gantt Chart) is the standard two dimensional grid with time on the X-axis and resources on the Y-axis. The time increments are defined by the model. For example, the time line may be divided into 15 minute intervals, i.e., 4 time slots per hour. The Schedule Board can be configured to zoom out to various coarser granularities (for example, 30 minutes, 1 hour, etc.). Tasks are assigned to slots that correspond to resources and time intervals.

The Schedule Board can be configured to display both a frozen history of scheduled tasks that have been released to the shop floor, as well as the current scheduling horizon. This enables the user to view previously scheduled tasks as they relate to the current schedule. In addition, Production Scheduler supports defining an end buffer in which tasks that start within the scheduling horizon may be completed. The Schedule Board may be in either one of two modes: the Manual Scheduling may be either "on" or "off". The current mode is indicated in the bottom right corner of the Schedule Window. When the Manual Scheduling is "off", the user has a view of the current schedule with certain tasks having violation colors. So, for example, if two tasks of differently flavored items are scheduled one after another in a mixing tank, the second task appears in yellow if there is a changeover penalty associated with a change in flavor in the mixing tank. When the Manual Scheduling mode is "on", the user can, after selecting a task in the tasks pane or on the Schedule Board, consider "What if?" scenarios associated with the manual scheduling of the selected task. That is, the user can see potential violations on the Schedule Board when he places the task on a particular resource in a given time slot. This is accomplished by a unique dynamic constraint checking environment, which provides instantaneous feedback when the Schedule Board is in the interactive manual scheduling mode. The color Red indicates that a hard constraint violation occurs if the task is moved into that position. Yellow indicates a soft violation and white indicates no violations. Again, as the user moves the cursor over the highlighted area in the Gantt chart, the Constraint Violations pane provides feedback with regard to the associated violations (based on the customized messages created in the modelling UI).

Any schedule generated by the system can be manipulated manually. The "What If?" mode provides the necessary guidance and feedback with regard to the consequences of scheduling a particular task on a particular resource at a particular time. The Scheduling UI provides simple drag-and-drop capabilities to move tasks that still have to be scheduled from the Task Grid to the Schedule Board, or to move tasks from resource to resource, or from one time slot to another on the Schedule Board. Production Scheduler also has the ability to pin (or freeze) certain tasks within a time range or on a specific resource. This gives the end user the flexibility to reschedule a set of tasks, while keeping the pinned tasks in their positions.

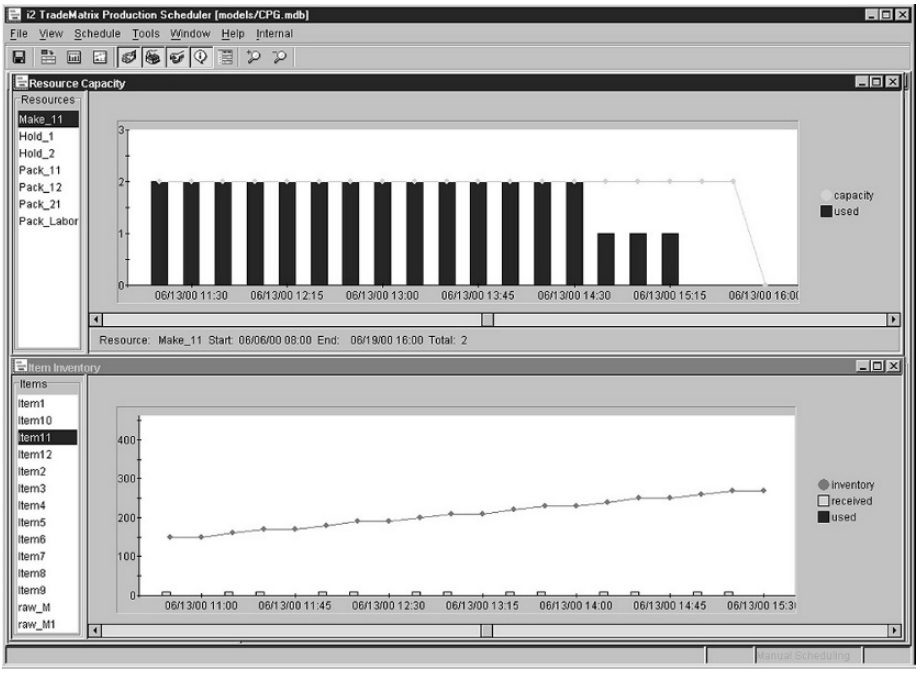


Fig. 19.8 i2's Production Scheduler Capacity Buckets Interface

The Scheduling UI also provides a number of performance measures and metrics with respect to the schedule (number of tasks and work orders scheduled versus number not yet scheduled; a summary of hard and soft constraint violations; resource utilization information; number of late work orders, etc.). The Scheduling UI provides an item production grid that displays the demand versus the produced quantities. The UI also provides time-varying charts that display resource capacity and item inventory over the scheduling horizon (see the Figure 19.8).

The remaining part of this section describes two implementations of the Production Scheduler.

Example 19.3.1 (An Implementation at Ford Motor Co.)

Production Scheduler has been installed at the Woodhaven Stamping Plant of the Ford Motor Co. This stamping plant has over 175 production lines in three basic stages of operation: Blanking, Die Pressing, and Assembly. The final products are fenders, roofs and doors used in the final assembly of Ford cars and trucks. The sheet metal (coils) are converted into blanks in the Blanking stage or Press stage (for direct coil feed lines). The Press lines shape the blanks into a form that is ready for Assembly. These parts

are either sent to the Assembly line or to other assembly plants. The goals of the scheduling implementation are:

(i) better utilization of plant resources (both labor and machine) in order to reduce direct overtime and increase throughput,
(ii) reduction of premium freight charges,
(iii) improved customer service,
(iv) schedules that minimize the number of die setups, and
(v) reduction of end item inventory.

Production Scheduler is used at the Woodhaven Stamping Plant to do cycle planning as well as detailed scheduling. The cycle planning lays out a month long production plan for each line with the goal of maintaining a minimal WIP and utilizing staffing levels optimally. It does this by considering due-dates, die-sets, labor requirements, storage rack availability, etc. The end result of the planning is a series of manufacturing codes denoting frequency of production of each part and the run days. This information is uploaded to the Ford MRP system on a monthly basis. The detailed scheduling lays out a detailed week long schedule for all three stages at a finer time granularity, considering the same constraints outlined above and a few additional constraints related to the finer time granularity. The schedule is uploaded daily into the Ford MRP system.

Since i2's inception at Ford Woodhaven, the plant has seen a significant reduction in average daily WIP inventories and has achieved further savings through increased throughput and reduced overhead from unplanned overtime, and a tighter control on inventories and cycle times. ||

Example 19.3.2 (An Implementation of Production Scheduler in a Gear Manufacturing Facility)

Production Scheduler has also been installed in a gear manufacturing facility of one of the world's leading heavy equipment manufacturers, with plans for implementation in other facilities of the company.

This gear manufacturing facility is based on cellular manufacturing. A typical route consists of seven primary operations: Heat treatment, gear cell, heat treatment, grinding, heat treatment, washing and checkout. This route is a generic one, as some parts could skip some of these operations and others can go through additional operations. The goal of the Production Scheduler implementation was to schedule the gear shop, minimizing the number of changeovers and taking into consideration runtimes, demand priorities and the flexibility of alternate resource choices. The implementation went live in less than 6 months, including both the initial knowledge acquisition sessions and complete end-to-end testing of the solution. Since implementing Production Scheduler, the company has realized an increased productivity and improved its service levels.

An increase in the productivity was achieved as follows. Prior to implementing Production Scheduler, the plant required 4-5 foremen to schedule

their shop floor area. Now a dedicated scheduler uses Production Scheduler and generates detailed schedules, freeing up resources for other productive uses in the plant.

The improvement in the service levels were a result of an increased visibility. Prior to implementing Production Scheduler, each foreman only had access to information about his/her shop floor and not upstream and downstream operations. With Production Scheduler, the dedicated scheduler has visibility across different areas of the plant, resulting in more efficient utilization of resources and, as a result, improved customer service levels. ||

19.4 Taylor Scheduling Software

Taylor Scheduling Software is one of the few remaining companies that are still exclusively in the business of providing scheduling solutions to manufacturers worldwide. In 1989 Taylor released the first version of its current scheduler for manufacturing.

The company has been focusing on batch manufacturing (e.g., chemicals, pharmaceuticals), on discrete manufacturing (e.g., aerospace parts), as well as on "mixed mode" manufacturing (i.e., combinations of batch and discrete manufacturing). The software lends itself to these different types of manufacturing, because it can be reconfigured fairly easily to reflect the types of conditions that are prevalent in the environment under consideration. In general, if it is possible to describe the manufacturing process clearly and accurately, then the Taylor software should be able to do the scheduling.

Currently, the system enables the user to specify independent calendars for machines as well as resources; the resources may be people, tools or materials. The system also enables the user to specify operating conditions and rules for the equipment as a function of the products being manufactured. Also, the Taylor Scheduler allows for completely different products to be "related" to one another through the use of attribute relationships (e.g., color, size, chemistry, and so on).

All these functionalities are built in the engine that generates the schedules. The engine automatically sequences the operations while minimizing setup times, cleanup times and other performance measures. It is also possible to specify a "preferred" machine for an operation with an alternate machine only being used when necessary in order to complete a job on time. When a partial schedule has been created, it can be frozen and released immediately to the people on the plant level, while the scheduler continues his work with regard to the jobs or operations that fall outside the frozen range.

A schedule is considered "good" if it satisfies all manufacturing rules and constraints. A schedule is considered "great" if it not only satisfies the manufacturing rules, but also the business needs of the plant (e.g., setup time reduction, and so on).

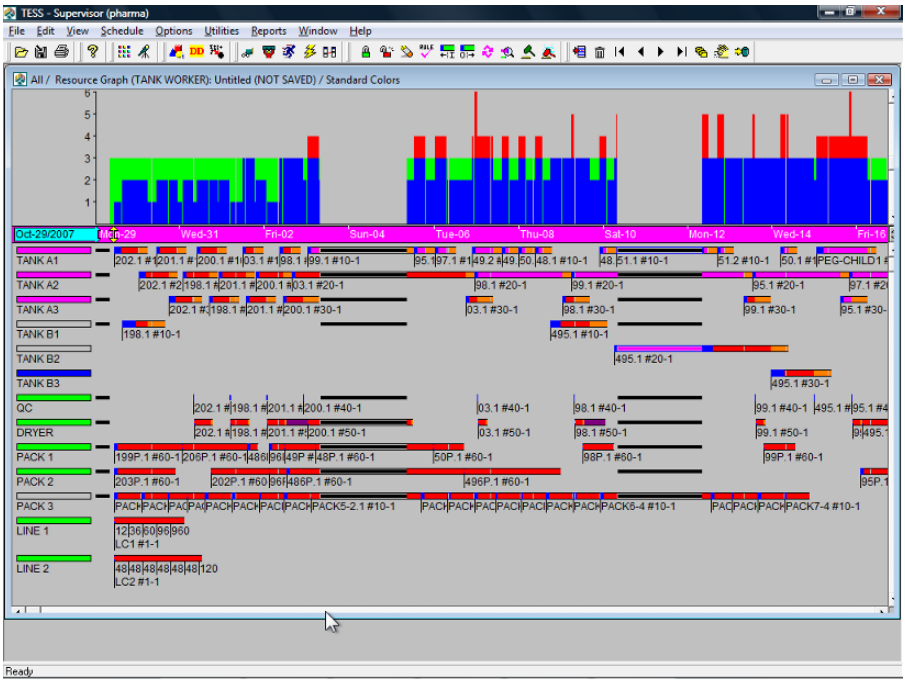


Fig. 19.9 Taylor Scheduler Gantt Chart Interface

The Taylor Scheduling Engine has a number of generic optimization procedures and heuristics built in, including priority rules (e.g., Earliest Due Date first, Shortest Processing Time first, etc.) and local search procedures (e.g., simulated annealing, genetic algorithms). The rules and procedures can be used in various different modes, namely in a forward scheduling mode, in a backward scheduling mode, or in a multi-pass mode.

The user interface allows the decision-maker to schedule interactively in a sophisticated manner. The schedule generated by the engine can be displayed in the form of a Gantt chart (see Figure 19.9), allowing the user to drag and drop operations in order to fine-tune the schedule. Moving an operation causes the automatic rescheduling of every other operation that is affected by the change; the system also deals with the changes in the resource requirements caused by the move. The user interface is quick to schedule and quick to respond to a user's request for a change and can show the impact of the change as it is occurring.

Example 19.4.1 (An Implementation in a Large Pharmaceutical Company)

The Taylor Scheduler has been implemented in a large pharmaceutical manufacturer based in Europe. The pharmaceutical company was going through the process of reducing the number of manufacturing facilities worldwide. This consolidation was putting pressure on the remaining plants which were now required to increase their output in order to compensate for the loss in production capacity. The burden was actually two-fold: it implied an increase in production volume as well as an increase in product variety.

The first facility targeted for the implementation of the Taylor Scheduler was supposedly the most complex manufacturing site. This site was responsible for the management of the so-called "presentations" for more than 80 countries (a presentation is a specific configuration of an end product for a specific market; it affects the packaging, the inserts, the language requirements, as well as the legal chemical make-up of the product.) The site had 900 different product/packaging configurations (SKU's) that had to go through 37 workcenters and 42 machines. Prior to the consolidation, the facility was responsible for only 150 presentations. If the installation of the software at this site would turn out successful, then it should be possible to install the system in every facility of the manufacturing division.

The manner in which the manual scheduling was done before the system was installed was clearly not satisfactory. It was time consuming and one could update the schedules only once a week. There were no simulation tools available that would enable the scheduler to consider what-if scenarios. Frequent changes on the shop floor added more time to the production process which in turn caused delivery delays for the end-products. There was clearly a disconnect between planning and execution.

For these reasons it had been decided that the scheduling process had to be automated. A number of goals were stated with regard to the selection and implementation of the scheduling system, namely

(i) a reduction in the number of employees needed by the facility's planning group.
(ii) a reduction in manufacturing labor.
(iii) a reduction in Work-In-Process (WIP).
(iv) a reduction in total setup time.
(v) an increase in production capacity.
(vi) a high conformance to planned schedules.
(vii) a high conformance to established short term plans.
(viii) a reduction in throughput time.

All these objectives were meticulously quantified by the plant's management. Figure 19.10 depicts the work flow of the daily activities on the shop floor level and the role of the scheduling system in this process. Figure 19.11

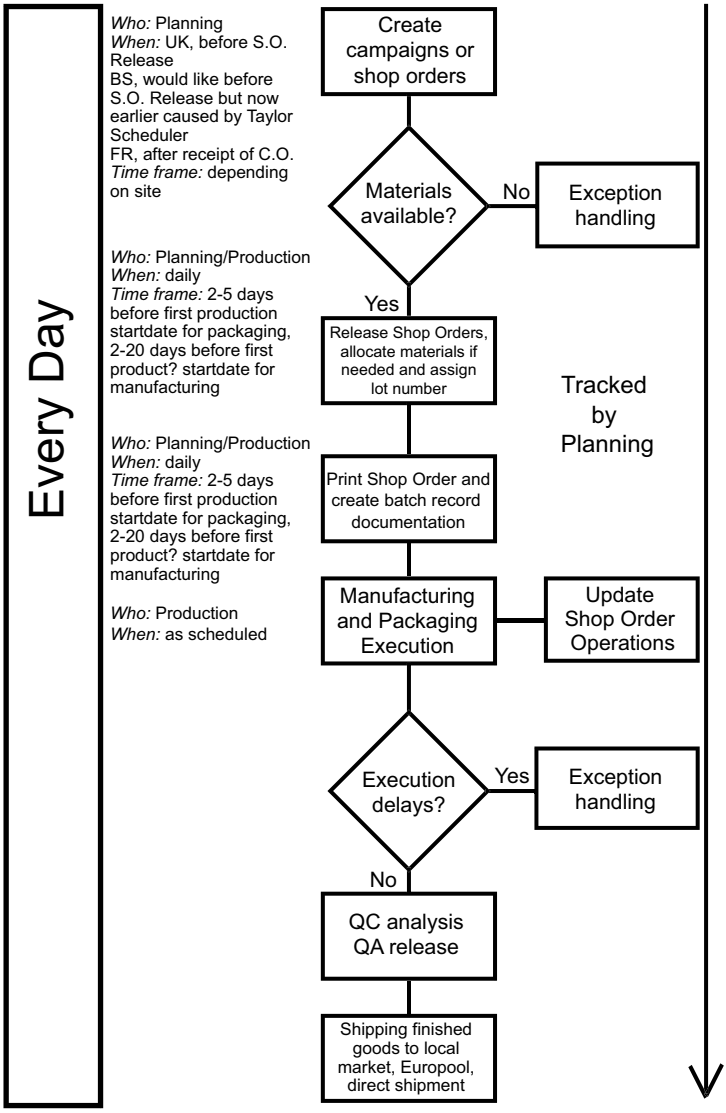


Fig. 19.10 Process Flow in Daily Shop Floor Execution

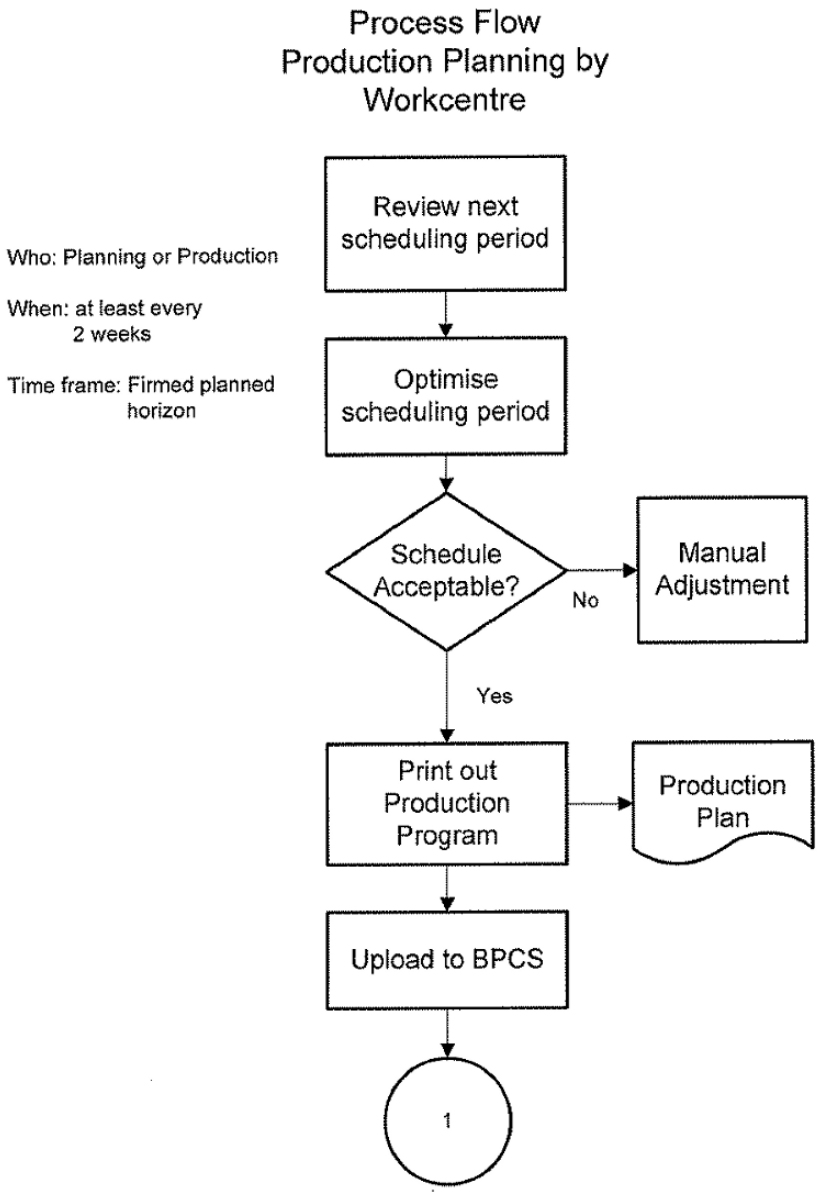


Fig. 19.11 Process Flow of Production Planning by Workcenter

describes the semi-monthly planning process for each workcenter and the role of the scheduling system in this process.

Several major tasks in the implementation process involved the company's ERP system. The data in the ERP system had to be cleaned up in order to ensure a perfect Bill of Materials and a clean router for every product. Also, an interface had to be designed between the ERP system and the Taylor Scheduler.

After the system had become operational it turned out that the Taylor scheduling system not only met most of the plant management's expectations, it also provided several unexpected benefits. The scheduling system enabled the planning group to

(i) consider alternative routings for the 900 products;
(ii) consider process constraints such as labor, materials and tooling;
(iii) improve the tracking of in-process orders. ||

Besides implementations in the pharmaceutical industry, Taylor Scheduling Software has been implemented in various other industries as well. For example, Taylor implemented a scheduling system in a Lexmark plant in Boulder (Colorado) that produces toner cartridges for laser printers and it implemented a scheduling system in a plant belonging to Fountain Set Holdings in Dongguan City (China) that produces dyed yarns, sewing threads, and garments.

19.5 Real Time Dispatching and Agent Scheduling at AMD

Scheduling plays a very important role in semiconductor manufacturing. All major semiconductor manufacturing enterprises, including Advanced Micro Devices (AMD), Intel, and Samsung, make consistently major investments in their planning and scheduling processes. The characteristics of semiconductor manufacturing (e.g., lengthy process flows, recirculation, equipment subject to breakdowns, multiple job families, and so on) make the scheduling process inherently very complicated and of critical importance. The manufacturing environment resembles a flexible job shop. Some of the details of the manufacturing process are described in Example 1.1.2 and a corresponding scheduling problem is described in Example 2.2.2.

While scheduling is considered to be an integral component of AMD's Automated Precision Manufacturing (APM) philosophy and a key to predictability as well as an enabler to the control of a synchronized manufacturing environment, there are many impediments in realizing these goals. Factors that make the scheduling process complicated include:

Manufacturing complexity: Fab manufacturing is a complex process that involves hundreds of processing steps on a shopfloor with hundreds of machines.

Randomness and variability: The fab environment is subject to a significant amount of randomness and has a high variability due to potential failures in manufacturing equipment and automated delivery systems.

Long time horizons: The complexity of the process brings about schedules that are very extended. The variability in the process renders such a schedule obsolete triggering rescheduling rather quickly.

Data Inconsistency and Availability: In some instances certain key data may not be available in time or in the right form for schedule or reschedule generation. (While the industry is showing an improving trend in this area, there are still some parts of the fab where manual insertion and overrides are necessary to ensure that schedules are consistent and achievable.)

Lack of execution mechanisms: In many plants there is not a mechanism in place that ensures that schedules generated by the system are actually being adopted and followed. Good schedules that are not implemented or are overridden are useless.

Lack of a framework: A good framework is necessary as the scheduling objectives and algorithms change with the manufacturing and business climate. As characteristics change, they should not necessitate wholesale changes or major developmental activities.

In 1997, Advanced Micro Devices spearheaded an industry effort in the use of a commercial tool called "Real Time Dispatch". The Real Time Dispatch (RTD) tool functions as a dynamic dispatching rule based on priority rules such as Earliest Release Date first (ERD), Earliest Due Date first (EDD), Minimum Slack first (MS), and so on. There were several reasons why the RTD tool became so popular in semiconductor fabs. First, the automation level of this mechanism is fairly minimal (there is a significant amount of manual involvement in the movement of lots within a bay: from stockers or WIP racks to tools, from tools to stockers or WIP racks, the checking whether tools are qualified to run existing lots, and so on). Second, while the physical steps followed by the lots are manual (as described above) the decision-making processes are understandably even more manual. However, it was observed that RTD had some drawbacks as well, namely

(i) a lack of comprehension of the time element (implying a lack of visibility of events that need to happen at scheduled times);
(ii) an inability to react to changes or unforeseen events such as lot shuffling at loadports;
(iii) a lack of compliance tracking;
(iv) an insufficient capability to assess the impact of the rules;
(v) an inability to comprehend relevant factory events and act or react.

At the beginning of the 21st century, the semiconductor industry started to migrate from 150mm and 200mm wafers to a 300mm wafers. This transition had, for various reasons, a significant impact on the scheduling process.

RTD had worked very well in the 150mm - 200mm wafer environment, in spite of the fact that it is based on a very myopic form of decision-making. However, it became clear that the ability to consider a longer time horizon and the ability to schedule tasks ahead of time are necessary conditions for ensuring an effective utilization of the resources in a 300mm manufacturing environment. Several semiconductor manufacturing companies had noticed this and started to experiment with other scheduling architectures (see Example 17.3.1).

So, taking future needs into consideration as well as the limitations encountered in past dispatch implementations, AMD embarked upon a path that it referred to as "Active Scheduling". Active scheduling refers to a pre-assignment of manufacturing resources to resource consumers and the setting up of calendar appointments in advance. Active scheduling also refers to the capability of enforcing schedules by executing tasks according to the calendar appointments.

In order to satisfy these requirements, AMD started (in conjunction with the National Institute of Standards (NIST)) working on the development of a new scheduling technology that is based on "agents". In AMD's agent based scheduling system, software agents representing different types of fab entities and events interact with one another to schedule, to react to events, and to perform other necessary functions, see Figures 19.12 and 19.13. An important function is to react to events that impact the schedule and reschedule when necessary. Mechanisms have been designed that build robustness in to the schedule so that "small" disruptions need not trigger rescheduling. Some important agent types are:

Tool agents: they represent the tools in the factory and are considered to be the service providers for lots and other "event" agents such as PMs (preventive maintenances), setups, and so on.

Resource agents: they represent other peripheral resources that are required for a service in addition to the tools. They may represent reticles for steppers, probe cards for testers, dummy wafers in furnaces, etc.

Lot agents: they represent the lots in the factory and schedule themselves while interacting with tool agents and other resource agents when needed. They are considered consumers of the services provided by the tool agents and resource agents.

Preventive maintenance (PM) agents: this is another class of resource consumers similar to the lot agents that interact with tool agents and resource agents.

Setup agents: This is another class (similar to the PM agents) which are also resource consumers that interact with tool agents and resource agents.

The agents themselves are autonomous, intelligent, state-aware, and goal-seeking and can react to a high frequency and variety of factory events and exceptions. Each agent maintains its own appointment calendar. Alarm mechanisms are built in to trigger or notify the start or the end of an appointment. The ability to schedule tasks ahead of time is a necessary requirement for enabling an effective utilization of the automation resources. Tasks that can be

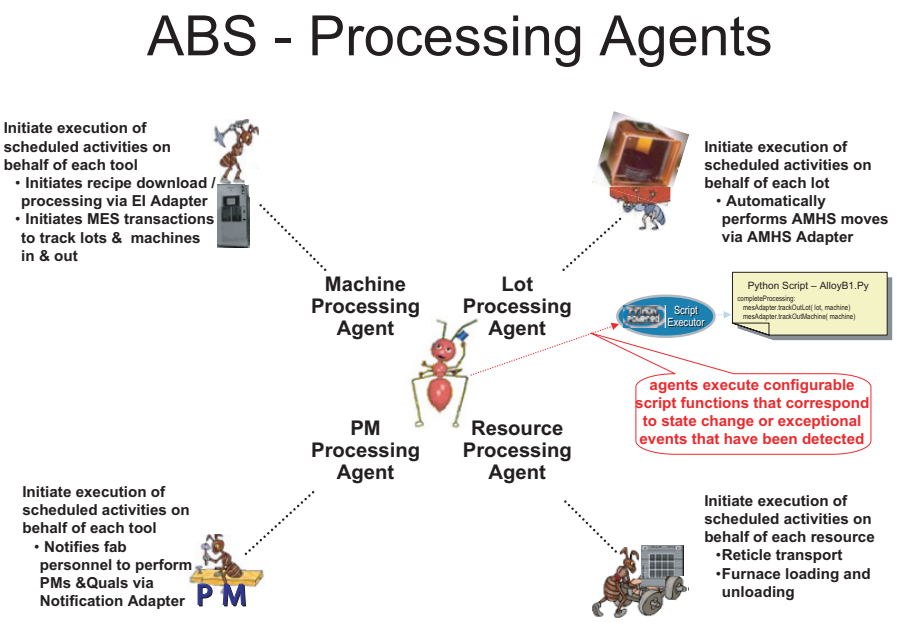


Fig. 19.12 Processing Agents in AMD's Agent Based Scheduling

"scheduled-ahead" include normal processing as well as exceptional processing, material and resource handling, loadport reservations, setups and opportunistic PMs and quals.

The system, as built, is capable of autonomous scheduling using a "market-based" approach that relies on budgets, costs and bids (similar to the approach described in Section 15.4). In this mechanism, resources or service providers and consumers of resources negotiate with each other in order to develop a schedule. Consumers of resources, such as lots, have budgets that are based on factors such as priority, due date, and so on. They try to minimize their payouts while staying on schedule. Service providers such as machines tender their services and ask resource consumers for bids that satisfy their conditions; they try to maximize their income. The negotiation between the resource consumers and resource providers resemble a market that is governed by the economic forces of supply and demand. Examples of differential bidding include open slot appointments, postponements of existing appointments, appointments for lots to join existing batches, and so on. The reactive scheduling/exception handling nature of this system means that it responds to factory events, factory definition changes, time deviations, and other unexpected events. This implies that schedules can be renegotiated based on events that will affect the start or end times of appointments. This rescheduling function is an important aspect of

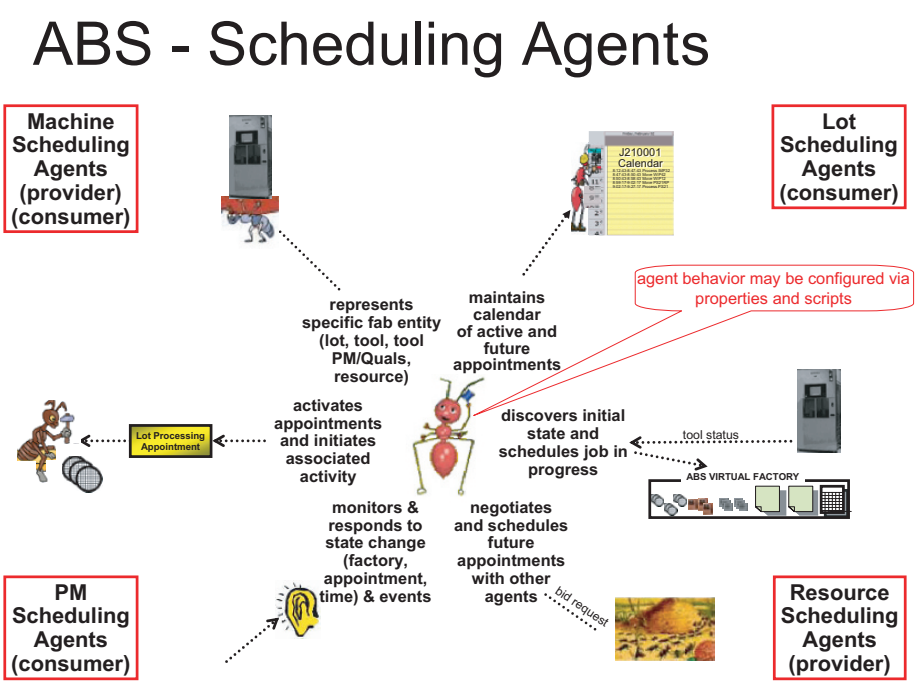


Fig. 19.13 Scheduling Agents in AMD's Agent Based Scheduling

the system, since the enforcement or execution systems cannot work with outdated schedules. The tasks, the events, and the exceptions are handled through configurable scripts with the ability to invoke factory system functions within the Manufacturing Execution System (MES), Automated Material Handling System (AMHS) and notification systems. A key component of the system is the metrics module that continuously monitors and tracks actual durations of tasks and appointments. This module provides the system also with a learning function.

As designed, the agent based scheduling system can maintain an efficient, high-volume, fully automated factory as it is geared to support massively-parallel operations in a timely and robust manner.

For the initial implementation of the new system, the end-users in the fabs requested a gradual approach that would allow the scheduling logic to be coded using the existing expertise in Real Time Dispatching. The open architecture of the new framework allowed this request to be handled in a matter of days. So, prior to using the built-in negotiation mechanisms and the market based approach, the transitionary approach is based on the RTD framework for assigning lots to tools and sequencing the lots for a particular tool. The agents handle the schedule, the calendar creation, the necessary automation, and the task execution. The execution component is integrated within the MES in such

a way that it is transparent to the end-user. It may seem to the end-user that the automation modes of the MES are defined in the same manner as if the agents did not exist even though the automation is actually handled by the agents. This enables a seamless migration to either environment (MES based automation or agent based automation) without the user having to switch from one system to another.

AMD's agent based scheduling architecture is currently being evaluated in a pilot mode at a wafer fab. This system has been developed while taking into consideration the transitions in wafer manufacturing from manual to automated processing and delivery to lean manufacturing with small lots and minimal queueing. The agent architecture is expected to comprehend and address these transitions.

19.6 An Implementation of Cybertec's Cyberplan

Cybertec is the largest planning and scheduling software house in Italy. It is based in Trieste. Its main product is a software system called Cyberplan, which is a suite of six software modules, namely

(i) Supply Chain Design,
(ii) Capacity Requirements Planning,
(iii) Material Requirements Planning,
(iv) Master Production Scheduling,
(v) Finite Capacity Planning and
(vi) Finite Capacity Scheduling.

Many different combinations of the various modules have been implemented in numerous companies. All modules have elaborate user interfaces, not only the usual interfaces such as Gantt charts (see Figure 19.14), but also the less common interfaces such as capacity buckets (see Figure 17.5). The optimization techniques used include constraint programming, local search (genetic algorithms), and mathematical programming procedures.

The remaining part of this section describes an implementation of Cybertec's Finite Capacity Planning and Finite Capacity Scheduling Modules

Example 19.6.1 (Scheduling a Yogurt Production Facility)

The Finite Capacity Planning and Scheduling modules have been implemented in a facility that produces jars of yogurt. The plant consists of three basic areas, namely the preparation area (in which the so-called white masses are prepared), the production area (in which the yogurt is produced in bulk), and the packaging area (in which the yogurt is packaged in jars).

The incoming milk (light and fat) are stored in the preparation area. From this storage the milk goes to the concentration area where it is condensed. The condensed milk enters the production area that consists of two separate flexible flow lines that are identical. The production process consists of four

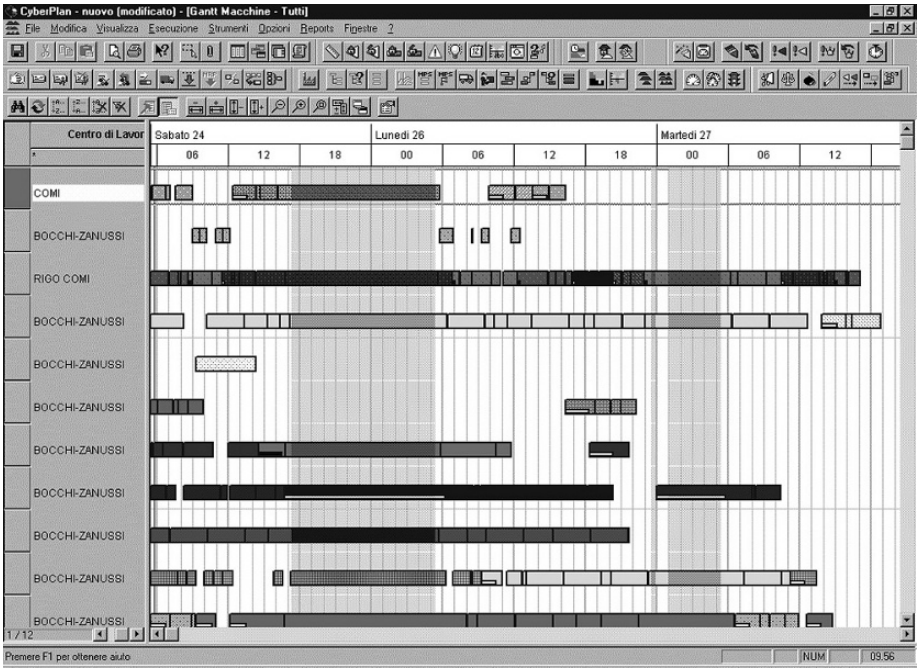


Fig. 19.14 Gantt Chart Interface of Cyberplan

stages: mixing, pasteurization, fermentation, and cooling (see Figure 19.15). The mixing stage in each one of the two flow lines has two mixers. The pasteurization stage in each one of the lines has a single pasteurizer. The fermentation stage of each line has ten tanks. Each fermentation stage is followed by a cooling center.

Both lines feed into a buffer area. There are two buffer areas and each area has four buffers. The yogurt may remain in a buffer for a maximum of 6 hours.

The two buffer areas feed into the packaging line. The input of the packaging line consists of the white mass, the fruit and other raw material. The output consists of the "packages", each of which containing a number of yogurt jars. There are almost 100 different finished products. The differences in packaging are due to the number of jars in a package, the weight of the jar, the flavor, and so on.

The production plan is established as follows. Sales forecasts and orders arrive at the plant on a weekly basis. This information is incorporated in the Master Production Schedule. Planning and scheduling is done on a daily basis. The daily input into the production scheduling module includes the following information:

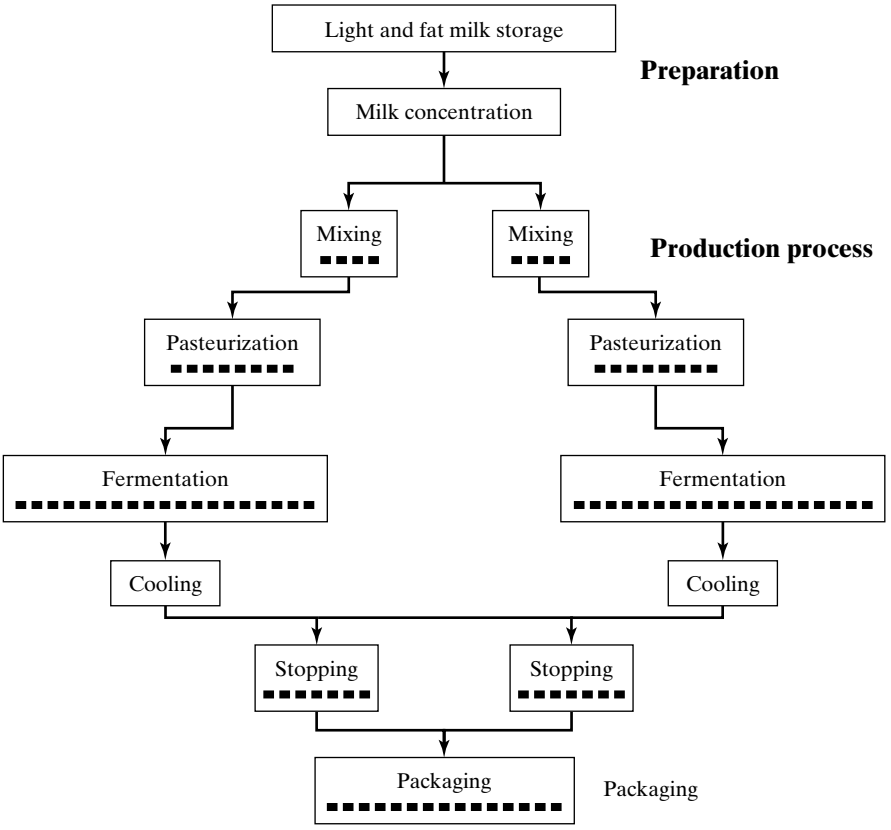


Fig. 19.15 Yogurt Production Process

(i) machines availability;
(ii) milk availability;
(iii) personnel availability.

There are several objectives that the scheduling system takes into consideration. The primary objective is to meet the due dates of the orders. An important secondary objective is the minimization of the number of different pieces of equipment used, since this helps control energy costs. The schedules are subject to several classes of constraints, an important class being the environmental constraints. The daily output of the scheduling system includes, besides the schedules, also the data that are needed to maintain quality control.

The scheduling procedure is a backwards procedure (see Figure 19.16). Based on the orders and their delivery dates (the finished products requirements), the net material requirements are determined. The basic input to the scheduling module is the delivery data and the material availability; the out-

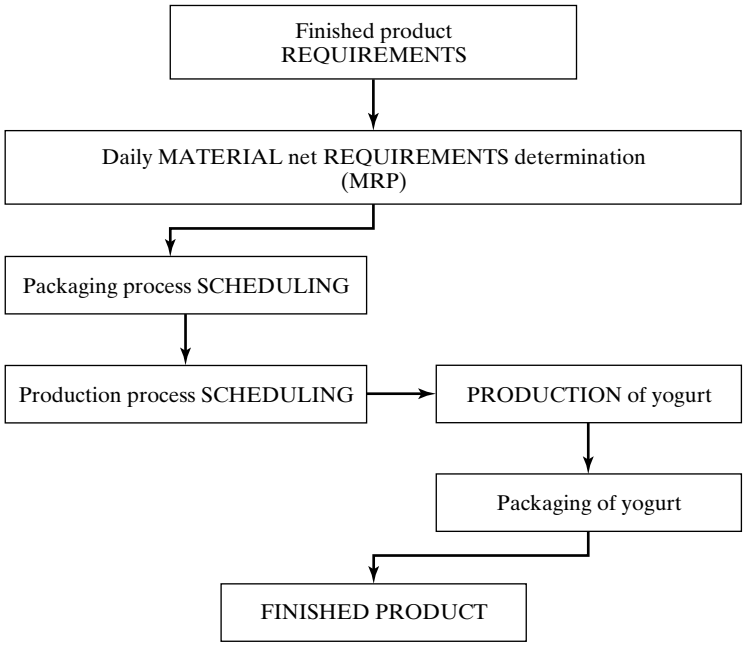


Fig. 19.16 Scheduling Process in Yogurt Facility

put is a packaging schedule. The packaging scheduling problem is basically a parallel machine scheduling problem. There are certain heuristic rules that have to be followed. Each packaging line has to follow a preferred sequence: natural yogurt, white puree, dark puree, white pieces, dark pieces. The reason for following such a sequence on a packaging line is obvious: this way the number of jars rejected because of quality considerations is minimized. This problem is comparable to the $Pm \mid r_j, s_{jk} \mid \sum w_j T_j$ problem that is considered in Example 18.2.2.

The solution of the packaging scheduling problem is an input to the production process scheduling problem. The production process is a form of continuous production (in contrast to the packaging which is a form of discrete production). In the production process the quantities are measured in weights or volumes rather than in units. The process has a large number of constraints and a limited amount of data (the number of different types of products to be produced is significantly smaller than the number of different items to be produced on the packaging line). The scheduling of the production process is done via a constraint-guided heuristic search procedure.

The system is used on a daily basis in the following manner:

Phase 1: Every day, before scheduling the production, the machine availabilities are entered into the program.

Phase 2: The packaging program function is selected and after the packaging schedule has been determined, the final result is displayed on the screen.

Phase 3: Before calling the procedure for scheduling the production, the settings of the pasteurizer are verified and set, i.e., the number of washing machines, the duration of the washing, the minimum time between washing, and the maximum pasteurization time.

Phase 4: After setting the parameters of the pasteurizer, the production scheduling procedure is called. The resulting schedules for the pasteurizers and fermenters are displayed on the screen separately.

Before the installation of the Cybertec system, schedules were generated manually after determining the daily requirements of finished products, taking into consideration the scheduled arrivals of milk, the amount of milk on-hand, and the personnel present. The resulting orders, still without a starting time and finishing time, were assigned to the packaging lines taking into account the product types. For each line a sequence was generated manually, subject to production sequence constraints. This resulted then in a packaging sequence with corresponding starting times and completion times. This information was then entered into a scheduler that generated a Gantt chart and determined the total requirements of white mass. The quantities of white mass were divided in different lots and a manual schedule of the production process mixes was done. This manual scheduling took into account the daily recipes, the fermenter and pasteurizer capacities, and the constraints on the time and on the buffers. The recipes could change every day because of the different protein contents of the milk. When the fermentation of the mixtures on the fermentation lines had to be scheduled, constraints with regard to milk preservation had to be taken into account.

Manual scheduling was a complicated process. It took at least two hours a day, and was error-prone. Also, constraints were at times violated, resulting in quality problems, reduced plant capacity utilization, and environmental problems.

After the installation of the Cybertec system the situation improved considerably. Schedules are generated in less than 2 or 3 minutes and no human errors are made. The pasteurizers and fermenters are used at capacity and the utilization of the plant capacity and the energy consumption are optimized. ||

19.7 LEKIN - A System Developed in Academia

The LEKIN system contains a number of scheduling algorithms and heuristics and is designed to allow the user to link and test his or her own heuristics and compare their performances with heuristics and algorithms that are embedded in the system. The system can handle a number of different machine environments, namely:

(i) single machine
(ii) parallel machines
(iii) flow shop
(iv) flexible flow shop
(v) job shop
(vi) flexible job shop

Furthermore, it is capable of dealing with sequence dependent setup times in all the environments listed above. The system can handle up to 50 jobs, up to 20 workcenters or workstations, and up to 100 machines.

The educational version of the LEKIN system is a teaching tool for job shop scheduling and is available on the CD that comes with this book. The system has been designed for use in either a Windows 98 or a Windows NT environment. Installation on a network server in a Windows NT environment may require some (minor) system adjustments. The program will attempt to write in the directory of the network server (which is usually read-only). The program can be installed in one of the following two ways. The system administrator can create a public directory on the network server where the program can write. Or, a user can create a new directory on a local drive and write a link routine that connects the new directory to the network server.

When LEKIN is run for the first time a "Welcome" page appears. Closing the welcome page makes the main menu appear. The main menu can also be accessed during a scheduling session by clicking on "start over" under "file".

The main menu allows the user to select the machine environment he is interested in. If the user selects a machine environment, he has to enter all the necessary machine data and job data manually. However, the user also has the option of opening an existing file in this window. An existing data file contains data with regard to one of the machine environments and a set of jobs. The user can open such an existing file, make changes in the file and work with the modified file. At the end of the session the user may save the modified file under a new name.

If the user wants to enter a data set that is completely new, he first must select a machine environment, and then a dialog box appears where he has to enter the most basic information, i.e., the number of workstations and the number of jobs to be scheduled. After the user has done this, a second dialog box appears and he has to enter the more detailed workstation information, i.e., the number of machines at the workstation, their availability, and the details needed to determine the setup times on each machine (if there are setup times). In the third dialog box the user has to enter the detailed information with regard to the jobs, i.e., release dates, due dates, priorities or weights, routing and processing times of the various operations. If the jobs require sequence dependent setup times, then the machine settings that are required for the processing have to be entered. The structure of the setup times is similar to the one described in Example 16.4.1. However, in the LEKIN system every job has just a single parameter, in contrast to the three parameters in Example 16.4.1.

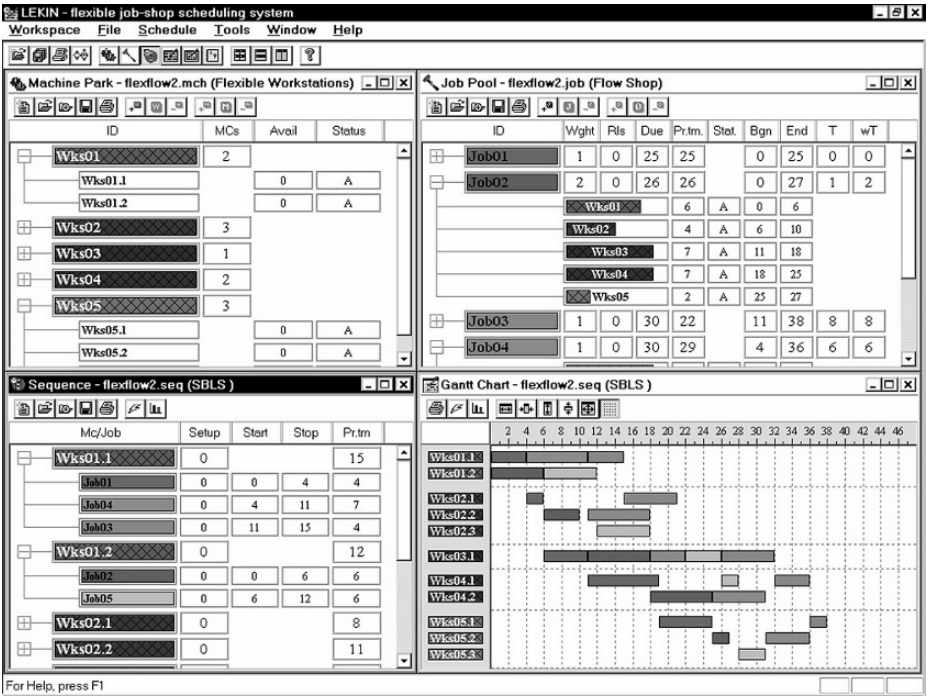


Fig. 19.17 LEKIN's four main windows

After all the data has been entered four windows appear simultaneously, namely,

(i) the machine park window,
(ii) the job pool window,
(iii) the sequence window, and
(iv) the Gantt chart window,

(see Figure 19.17).

The machine park window displays all the information regarding the workstations and the machines. This information is organized in the format of a tree. This window first shows a list of all the workstations. If the user clicks on a workstation, the individual machines of that workstation appear.

The job pool window contains the starting time, completion time, and more information with regard to each job. The information with regard to the jobs is also organized in the form of a tree. First, the jobs are listed. If the user clicks on a specific job, then immediately a list of the various operations that belong to that job appear.

The sequence window contains the lists of jobs in the order in which they are processed on each one of the various machines. The presentation here also

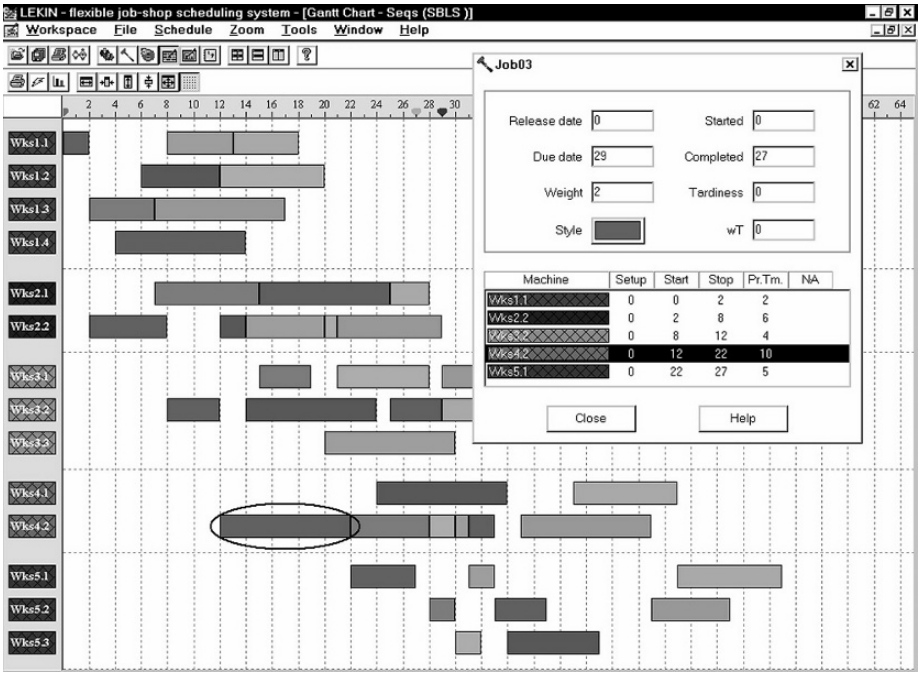


Fig. 19.18 LEKIN's Gantt Chart Window

has a tree structure. First all the machines are listed. If the user clicks on a machine, then all the operations that are processed on that machine appear in the sequence in which they are processed. This window is equivalent to the dispatch list interface described in Chapter 17. At the bottom of this sequence window there is a summary of the various performance measures of the current schedule.

The Gantt chart window contains a conventional Gantt chart. This Gantt chart window enables the user to do a number of things. For example, the user can click on an operation and a window pops up displaying the detailed information with regard to the corresponding job (see Figure 19.18). The Gantt chart window also has a button that activates a window where the user can see the current values of all the objectives.

The windows described above can be displayed simultaneously on the screen in a number of ways, e.g., in a quadrant style (see Figure 19.17), tiled horizontally, or tiled vertically. Besides these four windows there are two other windows, which will be described in more detail later on. These two windows are the log book window and the objective chart window. The user can print out the windows separately or all together by selecting the print option in the appropriate window.

The data set of a particular scheduling problem can be modified in a number of ways. First, information with regard to the workstations can be modified in the machine park window. When the user double-clicks on the workstation the relevant information appears. Machine information can be accessed in a similar manner. Jobs can be added, modified, or deleted in the job list window. Double clicks on the job displays all the relevant information.

After the user has entered the data set, all the information is displayed in the machine park window and the job pool window. However, the sequence window and the Gantt chart window remain empty. If the user in the beginning had opened an existing file, then the sequence window and the Gantt chart window may display information pertaining to a sequence that had been generated in an earlier session.

The user can select a schedule from any window. Typically the user would do so either from the sequence window or from the Gantt chart window by clicking on schedule and selecting a heuristic or algorithm from the drop-down menu. A schedule is then generated and displayed in both the sequence window and the Gantt chart window. The schedule generated and displayed in the Gantt chart is a semi-active schedule. A semi-active schedule is characterized by the fact that the start time of any operation of any given job on any given machine is determined by either the completion of the preceding operation of the same job on a different machine or the completion of an operation of a different job on the same machine (see Definition 2.3.5).

The system contains a number of algorithms for several of the machine environments and objective functions. These algorithms include

(i) dispatching rules,
(ii) heuristics of the shifting bottleneck type,
(iii) local search techniques, and
(iv) a heuristic for the flexible flow shop with the total weighted tardiness as objective (SB-LS).

The dispatching rules include EDD and WSPT. The way these dispatching rules are applied in a single machine environment and in a parallel machine environment is standard. However, they also can be applied in the more complicated machine environments such as the flexible flow shop and the flexible job shop. They are then applied as follows: each time a machine is freed the system checks which jobs have to go on that machine next. The system then uses the following data for the priority rules: the due date of a candidate job is then the due date at which the job has to leave the system. The processing time that is plugged in the WSPT rule is the sum of the processing times of all the remaining operations of that job.

The system also has a general purpose routine of the shifting bottleneck type that can be applied to each one of the machine environments and to every objective function. Since this routine is quite generic and designed for many different machine environments and objective functions, it cannot compete against

a shifting bottleneck heuristic that is tailor-made for a specific machine environment and a specific objective function.

The system also contains a neighbourhood search routine that is applicable to the flow shop and job shop (but not to the flexible flow shop or flexible job shop) with either the makespan or the total weighted tardiness as objective. If the user selects the shifting bottleneck or the local search option, then he must select also the objective he wants to minimize. When the user selects the local search option and the objective, a window appears in which the user has to enter the number of seconds he wants the local search to run.

The system has also a specialized routine for the flexible flow shop with the total weighted tardiness as objective; this routine is a combination of a shifting bottleneck routine and a local search (SB-LS). This routine tends to yield schedules that are reasonably good.

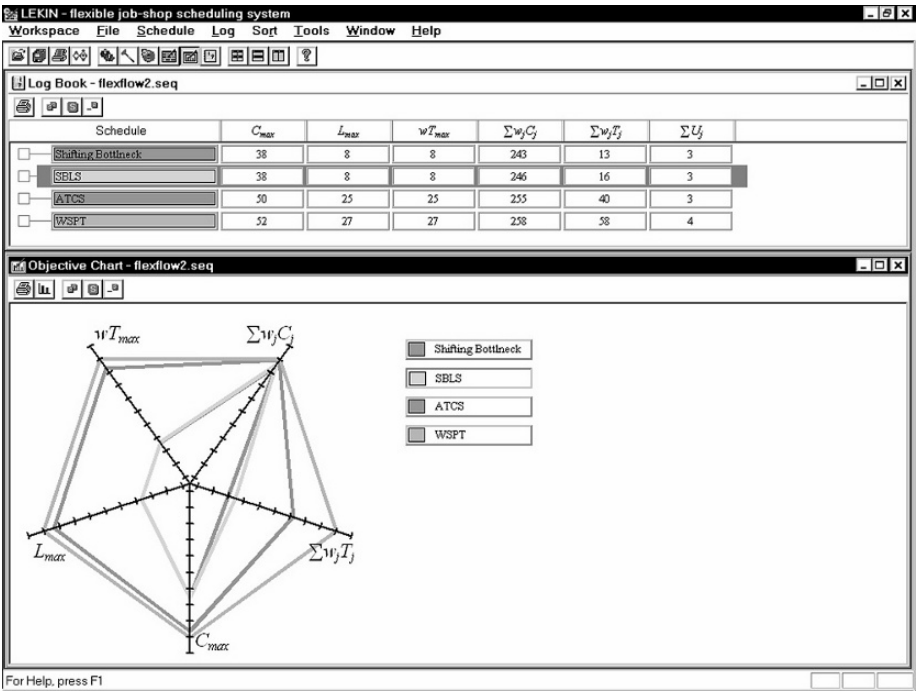
If the user wants to construct a schedule manually, he can do so in one of two ways. First, he can modify an existing schedule in the Gantt chart window as much as he wants by clicking, dragging and dropping operations. After such modifications the resulting schedule is, again, semi-active. However, the user can also construct this way a schedule that is *not* semi-active. To do this he has to activate the Gantt chart, hold down the shift button and move the operation to the desired position. When the user releases the shift button, the operation remains fixed.

A second way of creating a schedule manually is the following. After clicking on schedule, the user must select "manual entry". The user then has to enter for each machine a job sequence. The jobs in a sequence are separated from one another by a ";".

Whenever the user generates a schedule for a particular data set, the schedule is stored in a log book. The system automatically assigns an appropriate name to every schedule generated. If the user wants to compare the different schedules he has to click on "logbook". The user can change the name of each schedule and give each schedule a different name for future reference. The system can store and retrieve a number of different schedules (see Figure 19.19).

The schedules stored in the log book can be compared to one another by clicking on the "performance measures" button. The user may then select one or more objectives. If the user selects a single objective, a bar chart appears that compares the schedules stored in the log book with respect to the objective selected. If the user wants to compare the schedules with regard to two objectives, an $x-y$ coordinate system appears and each schedule is represented by a dot. If the user selects three or more objectives, then a multi-dimensional graph appears that displays the performance measures of the schedules stored in the log book.

Some users may want to incorporate the concept of setup times. If there are setup times, then the relevant data have to be entered together with all the other job and machine data at the very beginning of a session. (However, if at the beginning of a session an existing file is opened, then such a file may already contain setup data.) The structure of the setup times is as described



Schedule	C_{max}	L_{max}	wT_{max}	$\sum w_j C_j$	$\sum w_j T_j$	$\sum U_j$
Shifting Bottleneck	38	8	8	243	13	3
SBLS	38	8	8	246	16	3
ATCS	50	25	25	255	40	3
WSPT	52	27	27	258	58	4

Fig. 19.19 Logbook and comparisons of different schedules

in Example 16.4.1. Each operation has a single parameter or attribute, which is represented by a letter, e.g., A, B, and so on. This parameter represents the machine setting required for processing that operation. When the user enters the data for each machine, he has to fill in a setup time matrix for that machine. The setup time matrix for a machine specifies the time that it takes to change that machine from one setting to another, i.e., from B to E (see Figure 19.20). The setup time matrices of all the machines at any given workstation have to be the same (the machines at a workstation are assumed to be identical).

This setup time structure does not allow the user to implement *arbitrary* setup time matrices.

A more advanced user also can link his own algorithms to the system. This feature allows the developer of a new algorithm to test his algorithm using the interactive Gantt chart features of the system. The process of making such an external program recognizable by the system consists of two steps, namely, the preparation of the code (programming, compiling and debugging), and the linking of the code to the system.

The linking of an external program is done by clicking on "Tools" and selecting "Options". A window appears with a button for a New Folder and a button

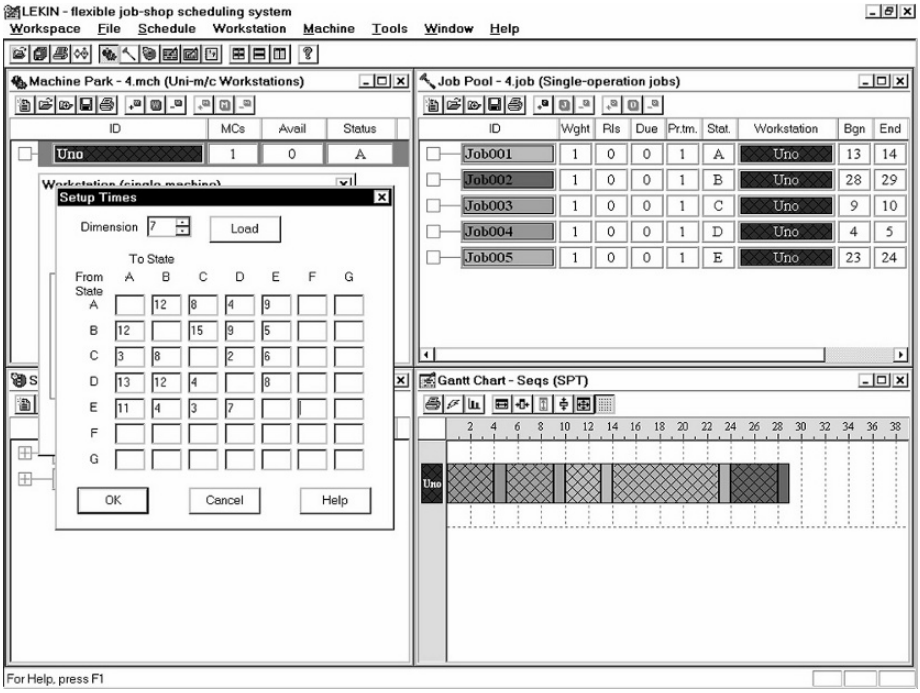


Fig. 19.20 Setup time matrix window

for a New Item. Clicking on New folder creates a new submenu. Clicking on a New Item creates a placeholder for a new algorithm. After the user has clicked on a New Item, he has to enter all the data with respect to the New Item. Under "Executable" he has to enter the full path to the executable file of the program. The development of the code can be done in any environment under Win3.2.

19.8 Discussion

This chapter presents an overview of the architectural designs of the IBM and SAP systems. A comparison of the IBM and the SAP-APO systems already makes it clear that there can be major differences in the main design characteristics of scheduling systems. The descriptions of the i2 and the Taylor Software systems highlight the importance of the role of the interfaces in scheduling systems. The evolution of the scheduling philosophy at AMD clearly illustrates the importance of scheduling in the semiconductor environment and the difficulties encountered in implementation. The description of the Cybertec system focuses on an actual implementation. From the setting of this implementation

it is clear that an implementation can have many peculiarities that may make it very hard to install a system just of the shelf. Customization seems to be always necessary.

The various systems described in this chapter clearly show how popular Genetic Algorithms are in practice. SAP-APO, i2, and Cybertec consider Genetic Algorithms an important element of their algorithm library.

Comments and References

Braun (2000) gives an overall description of the SAP-APO system. Akkiraju, Keskinocak, Murthy and Wu (1998a, 1998b) describe IBM's A-Team architecture and an application of this architecture in the paper industry. Bell (2000) gives a detailed description of i2's Production Scheduler. The scheduling systems developed and implemented at Advanced Micro Devices (AMD) are described by Krishnaswamy and Nettles (2005). Marchesi, Rusconi and Tiozzo (1999) describe the implementation of the Cybertec system in the yogurt factory. A detailed description of The LEKIN system is discussed in Feldman in Pinedo (1998).

Chapter 20

What Lies Ahead?

This chapter describes various research and development topics that are likely to receive attention in the near future. A distinction is made between theoretical research, applied research, and developments in system design.

The first section focuses on avenues for theoretical research. It describes the types of models that may become of interest as well as the types of results to be expected. The second section considers research areas that are more applied and more oriented towards real world scheduling problems. This section discusses some specific types of problems that may be investigated as well as the results to be expected. The third section focuses on systems development and integration issues. It analyzes the functional links between the scheduling algorithms, the system components, and the user.

There are many other research avenues that are not being considered. This chapter is not meant to be exhaustive; it merely tries to discuss some of the possible research directions.

20.1 Theoretical Research

In the future, theoretical research may well focus on theory and models that have not been covered in Parts I and II of this book. This section considers

(i) theoretical underpinnings of scheduling,
(ii) new scheduling formats and assumptions, and
(iii) theoretical properties of specific models.

Theoretical underpinnings of scheduling. The theoretical underpinnings will always receive a certain amount of research attention. One theoretical research area within deterministic scheduling deals with polyhedral combinatorics and

M.L. Pinedo, *Scheduling*, DOI: 10.1007/978-0-387-78935-4_20,

cutting planes. This research area has already generated for some of the basic scheduling models exact solution methods of the branch-and-cut type (see Appendix A) and also approximation algorithms with good performance guarantees. It is likely that this research will be extended to more general scheduling models. Another form of branch-and-bound, namely branch-and-price, has recently been applied successfully to several parallel machine scheduling problems.

Another research direction in deterministic scheduling is the area of Polynomial Time Approximation Schemes (PTAS) for NP-hard problems; this area has received a significant amount of attention in the recent past (see Appendix D). It is likely that this area will receive even more attention in the future and that schemes will be developed for models that are more complicated than those that have been considered up to now. However, it is not clear when these developments will start contributing to the effectiveness of heuristics used in practice. Other types of approximation methods will also be investigated; one very promising class of approximation algorithms is based on Linear Programming relaxations.

New scheduling formats and assumptions. The classical scheduling format, covering most models discussed in Part I, can be described as follows: In a given machine environment there are n jobs and all the information concerning the n jobs is available at time zero; a specific objective function has to be minimized and an optimal (or at least a very good) schedule has to be generated from scratch. There are several new and interesting research directions that concern models based on scheduling assumptions that differ significantly from those in Part I of this book.

One of the new scheduling formats concerns online scheduling. Online scheduling is important since it is in a way very different from the conventional (offline) deterministic models which assume that all information is known a priori, i.e., before any decision is made. In online scheduling, decisions have to be made based on information regarding the jobs that already have been released and not on information regarding jobs that are to be released in the future. In semi-online scheduling some, but not all, information regarding future job releases is known. The relationships between online scheduling, semi-online scheduling, and stochastic scheduling may receive some attention in the future as well. This seems to be a relatively open research area.

Another new format concerns scheduling with multiple agents. In a multi-agent scheduling problem each agent is responsible for a set of jobs and has his own objective function. The objective functions of the different agents may not be the same. The agents have to share the machine(s) with one another. A multi-agent problem is different from a multi-objective problem: in a multi-objective problem each job contributes to all objectives whereas in a multi-agent problem each job contributes to only one of the objectives (i.e., the objective of his agent). Multi-agent problems have several important applications. For example, maintenance scheduling problems can be formulated as multi-agent problems.

A third format comprises sequencing and scheduling games. In a scheduling game there are multiple players who have to share one (or more) machine(s) with each other; each player is responsible for a set of jobs and has his own objective function. Research in sequencing and scheduling games typically focuses on issues that are different from those studied in other classes of scheduling problems. When analyzing a scheduling game, one tends to be interested in certain structural properties of the game (e.g., the conditions under which a game is balanced or convex), whereas in most other types of scheduling problems one is interested in algorithms that generate optimal schedules for specific objective functions. There are various different types of scheduling games. One important class of scheduling games are the so-called cooperative scheduling games. In a cooperative scheduling game an initial schedule is given. Players can form coalitions and the players within a coalition may reschedule (or swap) their jobs among themselves; however, jobs belonging to players outside the coalition may not be completed later than they did in the initial schedule. In the new schedule, some players in the coalition may have a lower penalty cost, while others may have a higher penalty cost. A distribution mechanism has to be designed that allocates the benefits of the rescheduling over all the players within the coalition. Certain distribution mechanisms are referred to as core allocations. A cooperative scheduling game actually exhibits some similarities to competitive agent scheduling problems. The jobs that belong to a coalition may be regarded as jobs that belong to an agent and the sequence of the jobs within each coalition has to be optimized. Two coalitions can form a grand coalition and develop thus a joint schedule. The formation of a grand coalition can be compared to two competitive agents creating a joint schedule for their two sets of jobs.

A fourth, entirely different format, is based on the rescheduling concept. Rescheduling has been touched upon briefly in Chapter 18. However, a formal theoretical framework for the analysis of rescheduling problems has not yet been established. A rescheduling problem may have multiple objectives: the objective of the original problem (e.g., the total weighted tardiness) and the minimization of the difference between the new schedule (after rescheduling) and the old schedule (before rescheduling). It may be necessary to have formal definitions or functions that measure the "difference" or the "similarity" between two schedules for the same job set or between two schedules for two slightly different job sets, e.g., one job set having all the jobs of a second set plus one additional job (a rush job). The rescheduling process may also have to deal with "frozen" jobs, i.e., jobs that have been assigned earlier to certain time slots and that may not be moved. Sometimes the constraints on the frozen jobs do allow the scheduler to make some minor adjustments in the timing of the processing of these jobs. A frozen job may be started slightly earlier or slightly later. That is, there may be a time range in which a frozen job can be processed (there may be a limited amount of slack). Scheduling around frozen jobs tends to be similar to dealing with machine breakdowns or with preventive maintenance schedules (i.e., scheduling subject to availability constraints). However,

there may be a difference due to the fact that data regarding frozen jobs tend to be deterministic, whereas data concerning machine breakdowns tend to be stochastic.

The concept of robustness is closely related to rescheduling and deserves more attention in the future as well. Chapter 18 gives a relatively short treatment of this topic. It presents a number of robustness measures as well as several practical rules for constructing robust schedules. However, very little theoretical research has been done with regard to these rules. At this point it is not clear how useful these rules are in the various different scheduling environments.

Theoretical properties of specific models. One such research area deals with models that combine deterministic features with stochastic features. For example, consider a model with jobs that have deterministic processing times and due dates, and machines that are subject to a random breakdown process. Such a model may be quite realistic in many environments. Only very special cases are tractable. For example, a single machine with up times that are i.i.d. exponential and down times that all have the same mean can be analyzed. The WSPT rule then minimizes the total expected weighted completion time. However, it may be of interest to study more complicated machine environments in which the machines are subject to more general forms of breakdown processes.

Since such models tend to be more complicated than the more classical models described in Parts I and II of this book, the types of results one can expect may be of a more structural nature. Structural results may, for example, include proofs for dominance criteria or proofs for monotonicity results.

20.2 Applied Research

Applied research may go a little bit deeper into some of the topics covered in Part III of this book. The applied topics described in this section include

(i) performance analyses of heuristics,
(ii) robustness and reactive scheduling, and
(iii) integrated scheduling models.

Performance analysis of heuristics is a very important area of empirical and experimental research. Currently, many job shop problems can only be dealt with on a small scale. For example, it is still very hard to find an optimal solution for an instance of $Jm \mid\mid \sum w_j T_j$ with, say, 10 machines and 30 jobs. If there are multiple objectives and a parametric analysis has to be done, then the problem becomes even harder. Many different types of heuristics are available, but it is not clear how effective they are in dealing with large scale scheduling problems, e.g., job shop scheduling problems with two or three objectives and with, say, 50 machines and 1000 jobs. The heuristics for these large scale job shops may require continuous improvement and fine-tuning in the future.

Heuristics can be compared to one another with respect to several criteria. In practice, three criteria are important. First, the quality of the solution obtained;

second, the amount of computer time needed to generate a good solution; third, the time required to develop and implement the heuristic. Comparative studies of heuristics conducted in academic environments typically take only the first two criteria into account. However, in an industrial environment the third criterion is of critical importance. In industry it is important that the time needed to develop a heuristic be short. This is one of the reasons why in practice local search heuristics are often more popular than very sophisticated decomposition techniques such as the shifting bottleneck heuristic.

The performance of any heuristic depends, of course, on the structure of the scheduling problem, e.g., the type of routing constraints in job shops. The performance may even depend on the particular data set. Often, when one deals with a unary NP-hard problem, it turns out that most instances can be solved to optimality in a reasonably short time; however, some instances may turn out be very hard to solve and may require an enormous amount of computing time before reaching optimality. It is of interest to find out why such instances are hard to solve. Empirical studies indicate that a heuristic often may perform quite well when the data of an instance are generated randomly, whereas that heuristic may perform quite poorly when it is applied to an instance of that same problem with data from an industrial setting. (It may be the case that industrial data have certain dependencies and correlations that make such instances hard to solve.) It would be useful to establish rules that indicate the type of algorithm that is most suitable for the type of instance under consideration.

In order to characterize a problem instance properly, one may want to have a number of suitable descriptive factors, such as, for example, the due date tightness factor τ defined in Chapter 14. One may also have to assess proper weights to each one of the factors. It would be useful to know which type of algorithm is most suitable for a given instance when the following is known: the size of the instance (the scale), the values of characteristic factors, and the computer time available.

An important class of heuristic methods comprises local search procedures. The last two decades have seen an enormous amount of work on applications and implementations of local search procedures. This research has yielded interesting results with regard to neighbourhood structures. However, most of this research has focused on nonpreemptive scheduling. Preemptive scheduling has received very little attention from researchers specializing in local search procedures. One reason is that it tends to be more difficult to design an effective neighbourhood structure for a preemptive environment than for a nonpreemptive environment. It may be of interest to focus attention first on problems that allow preemptions only at certain well-defined points in time, e.g., when new jobs are released.

It is likely that in the future there will be a certain demand for industrial strength heuristics that are applicable to scheduling problems common in industry. Consider, for example, the problem $Qm \mid r_j, s_{jk} \mid \theta_1 \sum w_j T_j + \theta_2 C_{\max}$. This scheduling problem is typical in many process industries. The two objectives are quite common: One objective focuses on the due dates and the other tries

to balance the loads over the machines and minimize setup times. In the future, heuristics may be developed that are problem-specific and that can be linked easily to a variety of scheduling systems. These industrial strength heuristics may be hybrids that make use of Operations Research (OR) techniques as well as Artificial Intelligence (AI) techniques. For example, such a hybrid may combine an integer programming procedure with a constraint-based local search procedure.

Robustness and reactive scheduling. A completely different line of empirical research involves robustness and rescheduling. As stated in the previous section, the concepts of robustness and rescheduling may lead to interesting theoretical research. However, they may lead to even more interesting empirical and experimental research. New measures for robustness have to be developed. The definition of these measures may depend on the machine environment. Rescheduling procedures may be based on some very specific general purpose procedures that may have similarities to the procedures described in Chapters 14 and 15.

Integrated Scheduling models. More practical models often combine machine scheduling aspects with other aspects, such as inventory control, workforce scheduling, maintenance scheduling, capacity control or pricing. For example, in supply chains the production scheduling function is often tied to inventory control and to transportation scheduling. The models that are useful for the analysis of such real world environments tend to be more involved than the simpler machine scheduling models considered in this book. However, in the analysis of these more complicated models one may often have to resort to decomposition methods that partition the problem into a number of different modules. The smaller modules can then be tackled more easily using procedures that are described in this book.

For example, in the airline industry, planes and crews have to be scheduled in a coherent way. An extensive amount of research has been done on pure personnel scheduling (independent of machine scheduling), but little research has been done on models that combine personnel scheduling with machine scheduling. Some more theoretical research has been done in other areas related to these types of problems, namely resource constrained scheduling (i.e., a limited number of personnel may be equivalent to a constraining resource). However, research in resource constrained scheduling has typically focused on complexity analysis and on worst case analysis of heuristics. It may be of interest in the future to study more specific models that combine machine scheduling with personnel scheduling.

There are many scheduling applications in the information systems world. Nowadays, distributed computer systems are connected to one another in so-called grid environments in which users can submit jobs that are automatically assigned to appropriate resources. The performance of a grid computing system depends heavily on the underlying scheduling procedures. A grid scheduling system operates on the premise that a new job that is in need of processing must make itself known to the "resource selector". In current systems, the

resource selector acts as a gateway to the grid. It will select resources from a global directory and then allocates the job to one of the nodes on the grid. Typically, a job allocation is done in two phases. First, a job is allocated to a node on the grid, and second, within that node, the job is scheduled onto the processor. The first phase is referred to as resource allocation, whereas the second phase is referred to as job scheduling. The last decade has seen a fair amount of development and implementation of grid scheduling systems. However, it seems that less attention has been paid to the more theoretical and algorithmic aspects of these systems.

20.3 Systems Development

Systems development may focus in the future a little bit more on some of the topics covered in Part III of this book. In this section the topics discussed include

(i) problem decomposition and distributed scheduling,
(ii) user interfaces and interactive optimization,
(iii) scheduling description languages, and
(iv) integration within supply chain management systems.

Problem decomposition and distributed scheduling. Dealing with large scale scheduling problems may lead to implementations of distributed scheduling. Many industrial problems are so large that they cannot be solved on a single workstation. The computational effort has to be divided over a number of workstations or computers that may reside at different locations. With certain procedures the computational effort can be divided up rather easily whereas with other procedures it may not be that easy. For example, when a problem is solved via branch-and-bound it may be relatively easy to decompose the branching tree and partition the computational work involved. At periodic intervals the different workstations still have to compare their progress and share information (e.g., compare their best solutions found so far). If a problem is solved via time based decomposition, then distributed scheduling may also be applicable (as long as the schedules of the different periods are somewhat independent of one another). With the latest Internet technologies, distributed scheduling may become increasingly more important in the future.

User interfaces and interactive optimization. The development of user interfaces and interactive optimization may face some interesting hurdles in the future. The designs of the user interfaces have to be such that interactive optimization can be done easily and effectively. A scheduler must maintain at all times a good overview of the schedule, even when the schedule contains over a thousand jobs. The user interface must have abilities to zoom in and out of a schedule easily. In order to allow for interactive optimization the user interface must have provisions for clicking, dragging and dropping operations, freezing operations, dealing with cascading and propagation effects, and rescheduling.

After the user makes some manual changes in the system, the system may reschedule automatically in order to maintain feasibility (without any user input). The (internal) algorithms that are used to maintain schedule feasibility may be relatively simple; they may only postpone some operations. However, internal algorithms may also be more involved and may perform some internal reoptimization (that is done automatically). On the other hand, the reoptimization process may also be managed by the user; he may want to specify the appropriate objective functions for the reoptimization process. Reoptimization algorithms may be very different from optimization algorithms that generate schedules from scratch. The main reason why reoptimizing is harder than optimizing from scratch is because an algorithm that reoptimizes has to deal with boundary conditions and constraints that are dictated by the original schedule. Embedding rescheduling algorithms in a user interface that enables the user to optimize schedules interactively is not easy.

Scheduling description languages. Composition and integration of procedures have led to the development of so-called scheduling description languages. A scheduling description language is a high level language that enables a scheduler to write the code for a complex algorithm with only a limited number of concise statements or commands. Each statement in a description language involves the application of a relatively powerful procedure. For example, a statement may give an instruction to apply a tabu search procedure on a given set of jobs in a given machine environment. The input to such a statement consists of the set of jobs, the machine environment, the processing restrictions and constraints, the length of the tabu-list, an initial schedule, and the maximum number of iterations. The output consists of the best schedule obtained with the procedure. Other statements may be used to set up various different procedures in parallel or to concatenate two different procedures. Scheduling description languages are not yet very popular. However, the existing languages are still somewhat cumbersome and need streamlining. It is likely that there will be some improvement in the future.

Integration within supply chain management systems. Many companies had started out initially with developing scheduling software for the manufacturing industry. However, they soon realized that in order to compete in the market place they had to offer software dealing with all aspects of supply chain management. The types of modules that are required in supply chain optimization include, besides planning and scheduling, forecasting, demand management, inventory control, and so on. Scheduling problems in supply chain management have to take forecasts, inventory levels and routings into consideration. These integrated scheduling problems are considerably harder than the more elementary problems studied in the research literature. The structure and the organization of the software must be well designed and modular.

Comments and References

Some research has already been done on polyhedral combinatorics of scheduling problems. Queyranne and Wang (1991) analyze the polyhedra of scheduling problems with precedence constraints and Queyranne (1993) studies the structure of another simple scheduling polyhedron. Queyranne and Schulz (1994) present a general overview of polyhedral approaches to machine scheduling. Chen, Potts and Woeginger (1998) discuss approximation algorithms. Schuurman and Woeginger (1999) present ten open problems with regard to Polynomial Time Approximation Schemes.

Pruhs, Sgall and Torng (2004) present a survey of online and semi-online scheduling and refer to some open problems. Research on multi-agent scheduling has begun only recently; see, for example, Baker and Smith (2003), Agnetis, Mirchandani, Pacciarelli and Pacifici (2004), and Cheng, Ng, and Yuan (2006). Research on sequencing and scheduling games started already in the 1980s; however, this research has tended to be game theory oriented rather than scheduling oriented. For a fairly recent overview on sequencing games, see Curiel, Hamers and Klijn (2002). Rescheduling has received lately a significant amount of attention, see Vieira, Herrmann and Lin (2003), and Hall and Potts (2004). As stated in the text, rescheduling is also closely related to scheduling subject to availability constraints; for an overview on this class of scheduling problems, see Lee (2004).

For very good overviews of heuristic design as well as performance analysis of heuristics, see Morton and Pentico (1993), Ovacik and Uzsoy (1997), Aarts and Lenstra (1997), van Hentenryck and Michel (2005), and Hoos and Stützle (2005). For a relatively new class of local search procedures, the so-called dynasearch algorithms, see Congram, Potts and Van de Velde (2002). Recently, some research has focused on the scheduling issues that are of importance in supply chain management. This research area is at times referred to as supply chain scheduling; see, for example, Hall and Potts (2003), Chen and Vairaktarakis (2005), and Chen and Pundoor (2006). For some recent papers on grid scheduling, see Kurowski, Nabrzyski, Oleksiak and Weglarz (2006) and Deng, Chen, Wang, and Deng (2006).

McKay, Pinedo and Webster (2002) present a comprehensive practice-focused agenda for scheduling research.

An enormous amount of research and development is going on in user interfaces and interactive decision-making in general. For some general results on interactive decision-making, see, for example, Kerpedjiev and Roth (2000). For some more recent papers on user interfaces for interactive scheduling, see Chimani, Lesh, Mitzenmacher, Sidner and Tanaka (2005) and Derthick and Smith (2007).

Some research groups have already started to develop scheduling description languages; see, for example, Smith and Becker (1997).

Appendices

Appendix A

Mathematical Programming: Formulations and Applications

This appendix gives an overview of the types of problems that can be formulated as mathematical programs. All the applications discussed concern scheduling problems. In order to understand these examples the reader should be familiar with the notation introduced in Chapter 2.

This appendix is aimed at people who are already familiar with elementary Operations Research techniques. It makes an attempt to put various notions and problem definitions in perspective. Relatively little will be said about the standard solution techniques for dealing with such problems.

A.1 Linear Programming Formulations

The most basic mathematical program is the *Linear Program (LP)*. The LP refers to an optimization problem in which the objective and the constraints are linear in the decision variables. It can be formulated as follows:

$$\text{minimize} \quad c_1x_1 + c_2x_2 + \cdots + c_nx_n$$

subject to

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &\leq b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &\leq b_2 \\ &\vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n &\leq b_m \\ x_j &\geq 0 \qquad \text{for } j = 1, \ldots, n. \end{aligned}$$

The objective is the minimization of costs. The $c_1, \ldots, c_n$ vector is usually referred to as the cost vector. The decision variables $x_1, \ldots, x_n$ have to be determined in such a way that the objective function $c_1x_1 + \cdots + c_nx_n$ is minimized. The column vector $a_{1j}, \ldots, a_{mj}$ is referred to as activity vector j. The value of the variable x_j refers to the level at which this activity j is utilized. The $b_1, \ldots, b_m$ is usually referred to as the resources vector. The fact that in linear programming n denotes the number of activities has nothing to do with the fact that in scheduling theory n refers to the number of jobs; that m denotes the number of resources in linear programming also has nothing to do with the fact that m refers to the number of machines in scheduling theory. Usually the representation above is given in the following matrix form:

$$\text{minimize} \quad \bar{c}\bar{x}$$

subject to

$$\begin{aligned} \mathbf{A}\bar{x} &\leq \bar{b} \\ \bar{x} &\geq 0. \end{aligned}$$

There are several algorithms or classes of algorithms for dealing with an LP. The two most important ones are

(i) the simplex methods and
(ii) the interior point methods.

Although simplex methods work very well in practice, it is not known if there is any version that solves the LP problem in polynomial time. The best known example of an interior point method is *Karmarkar's Algorithm*, which is known to solve the LP problem in polynomial time. There are many texts that cover these subjects in depth.

A special case of the linear program is the so-called *transportation* problem. In the transportation problem the matrix $\mathbf{A}$ takes a special form. The matrix has mn columns and $m + n$ rows and takes the form

$$\mathbf{A} = \begin{bmatrix} \bar{1} & 0 & \cdots & 0 \\ 0 & \bar{1} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \bar{1} \\ \mathbf{I} & \mathbf{I} & \cdots & \mathbf{I} \end{bmatrix}$$

where $\bar{1}$ denotes a row vector with n 1's and $\mathbf{I}$ denotes an $n \times n$ identity matrix. All but two entries in each column (activity) of this $\mathbf{A}$ matrix are zero; the two nonzero entries are equal to 1. This matrix is associated with the following problem. Consider a situation in which items have to be shipped from m sources to n destinations. A column (activity) in the $\mathbf{A}$ matrix represents a route from a given source to a given destination. The cost associated with this column (activity) is the cost of transporting one item from the given source to the given destination. The first m entries in the $b_1, \ldots, b_{m+n}$ vector represent the supplies at the m sources, while the last n entries of the $b_1, \ldots, b_{m+n}$ vector represent the demands at the n destinations. Usually it is assumed that the sum of the demands equals the sum of the supplies and the problem is to transport all the items from the sources to the demand points and minimize the total cost incurred. (When the sum of the supplies is less than the sum of the demands there is no feasible solution and when the sum of the supplies is larger than the sum of the demands an artificial destination can be created where the surplus is sent at zero cost).

The matrix $\mathbf{A}$ of the transportation problem is an example of a matrix with the so-called *total unimodularity* property. A matrix has the total unimodularity property if the determinant of every square submatrix within the matrix has value -1, 0 or 1. It can be easily verified that this is the case with the matrix of the transportation problem. This total unimodularity property has an important consequence: if the values of the supplies and demands are all integers, then there is an optimal solution $x_1, \ldots, x_n$, which is a vector of integers and the simplex method will find such a solution.

The transportation problem is important in scheduling theory for a number of reasons. First, there are many scheduling problems that can be formulated as transportation problems. Second, transportation problems can be used for obtaining bounds in branch-and-bound procedures that are applied to NP-hard problems (see Section 3.6).

In the following example a scheduling problem is described that can be formulated as a transportation problem.

Example A.1.1 (A Transportation Problem)

Consider $Qm \mid p_j = 1 \mid \sum h_j(C_j)$. The speed of machine i is v_i. The variable x_{ijk} is 1 if job j is scheduled as the kth job on machine i and 0 otherwise. So the variable x_{ijk} is associated with an activity. The cost of operating this activity at unit level is

$$c_{ijk} = h_j(C_j) = h_j(k/v_i).$$

Assume that there are a total of $n \times m$ positions (a maximum of n jobs can be assigned to any one machine). Clearly, not all positions will be filled. The n jobs are equivalent to the n sources in the transportation problem and the $n \times m$ positions are the destinations. The problem can be formulated easily as an LP.

$$\text{minimize} \sum_{i=1}^{m}\sum_{j=1}^{n}\sum_{k=1}^{n} c_{ijk}x_{ijk}$$

subject to

$$\sum_{i}\sum_{k} x_{ijk} = 1 \qquad \text{for } j = 1,\ldots,n,$$

$$\sum_{j} x_{ijk} \leq 1 \qquad \text{for } i = 1,\ldots,m,\ \ k = 1,\ldots,n,$$

$$x_{ijk} \geq 0 \qquad \text{for } i = 1,\ldots,m,\ \ j = 1,\ldots,n,\ \ k = 1,\ldots,n.$$

The first set of constraints ensures that job j is assigned to one and only one position. The second set of constraints ensures that each position i,k has at most one job assigned to it. Actually from the LP formulation it is not immediately clear that the optimal values of the variables x_{ijk} have to be either 0 or 1. From the constraints it may appear at first sight that an optimal solution of the LP formulation may result in x_{ijk} values between 0 and 1. Because of the total unimodularity property, the constraints do not specifically have to require the variables to be either 0 or 1. ||

An important special case of the transportation problem is the *weighted bipartite matching* problem. This problem can be described as follows. Let $G = (N_1, N_2, A)$ be an undirected bipartite graph. This implies that there are two sets of nodes N_1 and N_2 with arcs connecting nodes from N_1 with nodes from N_2. There are m nodes in N_1 and n nodes in N_2. The set A denotes a set of undirected arcs. The arc $(j,k) \in A$, that connects node $j \in N_1$ with node $k \in N_2$, has a weight w_{jk}. The objective is to find a matching for which the sum of the weights of the arcs is minimum. Let the variable x_{jk} correspond to arc (j,k). The variable x_{jk} equals 1 if the arc (j,k) is selected for the matching and 0 otherwise. The relationship with the transportation problem is clear. Without loss of generality it may be assumed that $m > n$ (if this is not the case, then sets N_1 and N_2 can be interchanged). The nodes in N_1 then correspond to the sources while the nodes in N_2 correspond to the destinations. At each source there is exactly one item available and at each destination there is a demand for exactly one item. The cost of transporting one item from one source to one destination is equal to the weight of the matching. The problem can be formulated as the following LP.

$$\text{minimize} \sum_{j=1}^{m}\sum_{k=1}^{n} w_{jk}x_{jk}$$

subject to

$$\sum_{k=1}^{n} x_{jk} \leq 1 \qquad \text{for } j = 1, \ldots, m,$$

$$\sum_{j=1}^{m} x_{jk} \leq 1 \qquad \text{for } k = 1, \ldots, n,$$

$$x_{jk} \geq 0 \qquad \text{for } j = 1, \ldots, m, \quad k = 1, \ldots, n.$$

Again, it is not necessary to explicitly require integrality for the x_{jk} variables. The internal structure of the problem is such that the solution of the linear program is integral. The weighted bipartite matching problem is also important from the point of view of scheduling.

Example A.1.2 (A Weighted Bipartite Matching Problem)

Consider $Rm \mid\mid \sum C_j$. Position $(i, 1)$ now refers to the position of the last job scheduled on machine i; position $(i, 2)$ refers to the position of the job immediately before the last on machine i. Position (i, k) refers to that job on machine i which still has $k - 1$ jobs following it. So, in contrast to Example A.1.1, the count of job positions starts at the end and not at the beginning. The variable x_{ijk} is 1 if job j is the kth last job on machine i and 0 otherwise. One set of nodes consists of the n jobs, while the second set of nodes consists of the $n \times m$ positions. The arc that connects job j with position (i, k) has a weight kp_{ij}. ||

A special case of the weighted bipartite matching problem is the *assignment* problem. A weighted bipartite matching problem is referred to as an assignment problem when $n = m$ (the number of sources is equal to the number of destinations). The assignment problem is also important in scheduling theory. Deterministic as well as stochastic single machine problems with the n jobs having identically distributed processing times can be formulated as assignment problems.

Example A.1.3 (An Assignment Problem)

Consider a special case of the problem discussed in Example A.1.1, namely $1 \mid p_j = 1 \mid \sum h_j(C_j)$. There are n jobs and n positions, and the assignment of job j to position k has cost $h_j(k)$ associated with it. ||

A.2 Integer Programming Formulations

An *Integer Program (IP)* is basically a linear program with the additional requirement that the variables $x_1, \ldots, x_n$ have to be integers. If only a subset of the variables are required to be integer and the remaining ones are allowed to be real, the problem is referred to as a *Mixed Integer Program (MIP)*. In

contrast to the LP, an efficient (polynomial time) algorithm for the IP or MIP does *not* exist (see Appendix D).

Many scheduling problems can be formulated as integer programs. In what follows, two examples of integer programming formulations are given. The first example describes an integer programming formulation for $1 \mid\mid \sum w_j C_j$. Even though the $1 \mid\mid \sum w_j C_j$ problem is quite easy and can be solved by a simple priority rule, the problem still serves as an illustrative and useful example. This formulation is a generic one and can be used for scheduling problems with multiple machines as well.

Example A.2.1 (An Integer Programming Formulation with Time-Indexed Variables)

In order to formulate $1 \mid\mid \sum w_j C_j$ as an integer program let the integer variable x_{jt} be 1 if job j starts at the integer time t and 0 otherwise. Let $l = C_{\max} - 1$.

$$\text{minimize} \quad \sum_{j=1}^{n} \sum_{t=0}^{l} w_j (t + p_j) x_{jt}$$

subject to

$$\sum_{t=0}^{l} x_{jt} = 1 \qquad \text{for } j = 1, \ldots, n,$$

$$\sum_{j=1}^{n} \sum_{s=\max(t-p_j,0)}^{t-1} x_{js} = 1 \qquad \text{for } t = 0, \ldots, l,$$

$$x_{jt} \in \{0,1\} \qquad \text{for } j = 1, \ldots, n, \;\; t = 0, \ldots, l.$$

The first set of constraints ensures that a job can start only at one point in time. The second set of constraints ensures that only one job can be processed at any point in time, while the last set contains the integrality constraints on the variables. The major disadvantage of this formulation is the number of variables required. There are $nC_{\max}$ variables x_{jt}. ||

Often, there is more than one integer programming formulation of the same problem. In the next example a different integer programming formulation is given for $1 \mid\mid \sum w_j C_j$. This second formulation can also be applied to $1 \mid prec \mid \sum w_j C_j$.

Example A.2.2 (An Integer Programming Formulation with Sequencing Variables)

Consider $1 \mid prec \mid \sum w_j C_j$. Let x_{jk} denote a $0 - 1$ decision variable that assumes the value 1 if job j precedes job k in the sequence and 0 otherwise.

The values x_{jj} have to be 0 for all j. The completion time of job j is then equal to $\sum_{k=1}^{n} p_k x_{kj} + p_j$. The integer programming formulation of the problem without precedence constraints thus becomes

$$\text{minimize} \sum_{j=1}^{n} \sum_{k=1}^{n} w_j p_k x_{kj} + \sum_{j=1}^{n} w_j p_j$$

subject to

$$\begin{aligned} x_{kj} + x_{jk} &= 1 && \text{for } j,k = 1,\ldots,n,\ j \neq k, \\ x_{kj} + x_{lk} + x_{jl} &\geq 1 && \text{for } j,k,l = 1,\ldots,n,\ j \neq k, j \neq l, k \neq l, \\ x_{jk} &\in \{0,1\} && \text{for } j,k = 1,\ldots,n, \\ x_{jj} &= 0 && \text{for } j = 1,\ldots,n. \end{aligned}$$

The third set of constraints can be replaced by a combination of (i) a set of linear constraints that require all x_j to be nonnegative, (ii) a set of linear constraints requiring all x_j to be less than or equal to 1 and (iii) a set of constraints requiring all x_j to be integer. Constraints requiring certain precedences between the jobs can be added easily by specifying the corresponding x_{jk} values. ||

There are several ways for dealing with Integer Programs. The best known approaches are:

(i) cutting plane (polyhedral) techniques and

(ii) branch-and-bound techniques.

The first class of techniques focuses on the linear program relaxation of the integer program. They aim at generating *additional* linear constraints that have to be satisfied for the variables to be integer. These additional inequalities constrain the feasible set more than the original set of linear inequalities without cutting off integer solutions. Solving the LP relaxation of the IP with the additional inequalities then yields a different solution, which may be integer. If the solution is integer, the procedure stops as the solution obtained is an optimal solution for the original IP. If the variables are not integer, more inequalities are generated. As this method is not used in the main body of the book it is not further discussed here.

The second approach, branch-and-bound, is basically a sophisticated way of doing complete enumeration that can be applied to many combinatorial problems. The branching refers to a partitioning of the solution space. Each part of the solution space is then considered separately. The bounding refers to the development of lower bounds for parts of the solution space. If a lower bound on the objective in one part of the solution space is larger than a solution already obtained in a different part of the solution space, the corresponding part of the former solution space can be disregarded.

Branch-and-bound can easily be applied to integer programs. Suppose that one solves the LP relaxation of an IP (that is, solves the IP without the integrality constraints). If the solution of the LP relaxation happens to be integer, say $\bar{x}^0$, then this solution is optimal for the original integer program as well. If $\bar{x}^0$ is not integer, then the value of the optimal solution of the LP relaxation, $\bar{c}\bar{x}^0$, still serves as a lower bound for the value of the optimal solution for the original integer program.

If one of the variables in $\bar{x}^0$ is not integer, say $x_j = r$, then the branch-and-bound procedure proceeds as follows. The integer programming problem is split into two subproblems by adding two mutually exclusive and exhaustive constraints. In one subproblem, say *Problem (1)*, the original integer program is modified by adding the additional constraint

$$x_j \leq \lfloor r \rfloor,$$

where $\lfloor r \rfloor$ denotes the largest integer smaller than r, while in the other subproblem, say *Problem (2)*, the original integer program is modified by adding the additional constraint

$$x_j \geq \lceil r \rceil$$

where $\lceil r \rceil$ denotes the smallest integer larger than r. It is clear that the optimal solution of the original integer program has to lie in the feasible region of one of these two subproblems.

The branch-and-bound procedure now considers the LP relaxation of one of the subproblems, say *Problem (1)*, and solves it. If the solution is integer, then this branch of the tree does not have to be explored further, as this solution is the optimal solution of the original integer programming version of *Problem (1)*. If the solution is not integer, *Problem (1)* has to be split into two subproblems, say *Problem (1.1)* and *Problem (1.2)* through the addition of mutually exclusive and exhaustive constraints.

Proceeding in this manner a tree is constructed. From every node that corresponds to a noninteger solution a branching occurs to two other nodes, and so on. The procedure stops when all nodes of the tree correspond to problems of which the linear program relaxations have either an integer solution or a fractional solution that is higher than a feasible integer solution found elsewhere. The node with the best solution is the optimal solution of the original integer program.

An enormous amount of research and experimentation has been done on branch-and-bound techniques. For example, the bounding technique described above based on LP relaxations is relatively simple. There are other bounding techniques that generate lower bounds that are substantially better (higher) than the LP relaxation bounds and better bounds typically cut down the overall computation time substantially. One of the most popular bounding techniques is referred to as Lagrangean relaxation. This strategy, instead of dropping the integrality constraints, relaxes some of the main constraints. However, the relaxed constraints are not totally dropped. Instead, they are dualized or weighted

in the objective function with suitable Lagrange multipliers to discourage violations.

Two ways of applying branch-and-bound have proven to be very useful in practice, namely:

(i) branch-and-cut and
(ii) branch-and-price (also known as column generation).

Branch-and-cut combines branch-and-bound with cutting plane techniques. Branch-and-cut uses in each subproblem of the branching tree a cutting plane algorithm to generate a lower bound. That is, a cutting plane algorithm is applied to the problem formulation that includes the additional constraints introduced at that node.

Branch-and-price, also referred to as column generation, is often used to solve integer programs with a huge number of variables (columns). A branch-and-price algorithm always works with a restricted problem in a sense that only a subset of the variables is taken into account; the variables outside the subset are fixed at 0. From the theory of Linear Programming it is known that after solving this restricted problem to optimality, each variable that is included has a nonnegative so-called *reduced cost.* If each variable that is not included in the restricted problem has a nonnegative reduced cost, then an optimal solution for the original problem is found. However, if there are variables with a negative reduced cost, then one or more of these variables should be included in the restricted problem. The main idea behind column generation is that the occurrence of variables with negative reduced cost is not verified by enumerating all variables, but rather by solving an optimization problem. This optimization problem is called the pricing problem and is defined as the problem of finding the variable with minimum reduced cost. To apply column generation effectively it is important to find a good method for solving the pricing problem. A branch-and-bound algorithm in which the lower bounds are computed by solving LP relaxations through column generation is called a branch-and-price algorithm.

Column generation has been applied successfully to various parallel machine scheduling problems.

A.3 Bounds, Approximations and Heuristics Based on Linear Programming

Since Linear Programming formulations can be solved fairly efficiently, a certain amount of research has focused on the use of LP in the development of bounds, approximations, and heuristics for NP-Hard scheduling problems. One obvious way to obtain a lower bound on the objective function of an NP-hard scheduling problem that can be formulated as a $0-1$ Mixed Integer Program is by relaxing the integer $0-1$ variables and replacing them with linear constraints that force these variables to be nonnegative and also less than or equal to 1.

Consider the following Mixed Integer Programming formulation for $1 \mid r_j \mid \sum w_j C_j$. This MIP is a more general version of the formulation for $1 \mid\mid \sum w_j C_j$ presented in Example A.2.1. Again, let the integer variable x_{jt} be 1 if job j starts at the integer time t and 0 otherwise. Let l denote the planning horizon. An upper bound on the planning horizon can be obtained by adding the sum of the processing times to the last release date.

$$\text{minimize} \quad \sum_{j=1}^{n} \sum_{t=0}^{l} w_j (t + p_j) x_{jt}$$

subject to

$$\sum_{t=0}^{l} x_{jt} = 1 \qquad \text{for } j = 1, \ldots, n,$$

$$\sum_{j=1}^{n} \sum_{s=\max(t-p_j,0)}^{t-1} x_{js} \leq 1 \qquad \text{for } t = 0, \ldots, l,$$

$$x_{jt} \in \{0,1\} \qquad \text{for } j = 1, \ldots, n; \;\; t = 0, \ldots, l,$$

$$x_{jt} = 0 \qquad \text{for } j = 1, \ldots, n; \;\; t = 0, \ldots, \max(r_j - 1, 0).$$

The first set of constraints ensures that a job can start only at one point in time. The second set of constraints ensures that at most one job can be processed at any point in time (since the jobs have different release dates, it may be the case that at certain points in time the machine may have to remain idle). The third set contains the integrality constraints on the variables. The last set of constraints ensures that a job cannot start before its release date.

The LP relaxation of this problem is obtained by relaxing the integrality constraints on x_{jt} and replacing them with the constraint set

$$0 \leq x_{jt} \leq 1 \qquad j = 1, \ldots, n; \; t = 0, \ldots, l.$$

This LP relaxation has been observed to give very strong lower bounds. However, it is somewhat difficult to solve because of its size (there are $n \times (l + 1)$ variables x_{jt}). The relaxation yields solutions in which the jobs are sliced in horizontal pieces. For example, if a job in the original problem requires 6 time units of uninterrupted processing, then a feasible solution in the relaxation may require this job to occupy, say, 0.4 of the machine capacity for one uninterrupted period of 6 time units and 0.6 of the machine capacity for another uninterrupted period of 6 time units. These two periods of 6 time units may partially overlap. A feasible solution of the relaxation may, of course, slice a job in more than two horizontal pieces.

Clearly, the solutions obtained with the relaxation will be useful for any branch-and-bound approach that is applied to the original MIP. However, the solution obtained with the relaxation may also form the basis for a heuristic solution to the original problem. Let

$$\sum_{t=0}^{l-1}(t+p_j)x_{jt}$$

denote the so-called "average completion time" of job j in the relaxed problem. One solution for the original MIP can now be generated by ordering the jobs nonpreemptively in increasing order of their average completion times. It has been shown that, from a theoretical point of view, the objective value using this heuristic can be at most three times larger than the objective value of the optimal solution. Empirical research has shown that in practice the heuristic solution will typically be within 2-4% of optimality.

The heuristic above consists thus of several phases. It requires a MIP formulation, an LP relaxation of the MIP and a transformation of the solution of the LP into a feasible schedule for the original problem. It has already been observed earlier that for any given scheduling problem there may be several MIP formulations and each such formulation may lead to a different type of LP relaxation. Moreover, for any specific LP relaxation there may be a number of ways of translating the LP solution into a feasible nonpreemptive schedule for the original problem (a fair amount of research has been done on this last phase as well).

If it is desirable to have solutions that are guaranteed to be *very* close to optimality, then a Polynomial Time Approximation Scheme (PTAS) may have to be used. These approximation schemes are described in Appendix D.

A.4 Disjunctive Programming Formulations

There is a large class of mathematical programs in which the constraints can be divided into a set of *conjunctive* constraints and one or more sets of *disjunctive* constraints. A set of constraints is called conjunctive if each one of the constraints has to be satisfied. A set of constraints is called disjunctive if at least one of the constraints has to be satisfied but not necessarily all.

In the standard linear program all constraints are conjunctive. The mixed integer program described in Example A.2.1 in essence contains pairs of disjunctive constraints. The fact that the integer variable x_{jk} has to be either 0 or 1 can be enforced by a pair of disjunctive linear constraints: either $x_{jk} = 0$ or $x_{jk} = 1$. This implies that the problem $1 \mid prec \mid \sum w_j C_j$ can be formulated as a *disjunctive program* as well.

Example A.4.1 (A Disjunctive Programming Formulation)

Before formulating $1 \mid prec \mid \sum w_j C_j$ as a disjunctive program it is of interest to represent the problem by a disjunctive graph model. Let N denote the set of nodes that correspond to the n jobs. Between any pair of nodes (jobs) j and k in this graph exactly one of the following three conditions has to hold:

(i) job j precedes job k,

(ii) job k precedes job j and

(iii) jobs j and k are independent with respect to one another.

The set of directed arcs A represent the precedence relationships between the jobs. These arcs are the so-called conjunctive arcs. Let set I contain all the pairs of jobs that are independent of one another. Each pair of jobs $(j,k) \in I$ are now connected with one another by two arcs going in opposite directions. These arcs are referred to as disjunctive arcs. The problem is to select from each pair of disjunctive arcs between two independent jobs j and k one arc that indicates which one of the two jobs goes first. The selection of disjunctive arcs has to be such that these arcs together with the conjunctive arcs do not contain a cycle. The selected disjunctive arcs together with the conjunctive arcs determine a schedule for the n jobs.

Let the variable x_j in the disjunctive program formulation denote the completion time of job j. The set A denotes the set of precedence constraints $j \rightarrow k$ that require job j to be processed before job k.

$$\text{minimize} \quad \sum_{j=1}^{n} w_j x_j$$

subject to

$$\begin{aligned} x_k - x_j &\geq p_k && \text{for all } j \rightarrow k \in A, \\ x_j &\geq p_j && \text{for } j = 1, \ldots, n, \\ x_k - x_j \geq p_k \ \text{ or } \ x_j - x_k &\geq p_j && \text{for all } (j,k) \in I. \end{aligned}$$

The first and second set of constraints are sets of conjunctive constraints. The third set is a set of disjunctive constraints. ||

The same techniques that are applicable to integer programs are also applicable to disjunctive programs. The application of branch-and-bound to a disjunctive program is straightforward. First the LP relaxation of the disjunctive program has to be solved (i.e., the LP obtained after deleting the set of disjunctive constraints). If the optimal solution of the LP by chance satisfies all disjunctive constraints, then the solution is optimal for the disjunctive program as well. However, if one of the disjunctive constraints is violated, say the constraint

$$(x_k - x_j) \geq p_k \ \text{ or } \ (x_j - x_k) \geq p_j,$$

then two additional LP's are generated. One has the additional constraint $(x_k - x_j) \geq p_k$ and the other has the additional constraint $(x_j - x_k) \geq p_j$. The procedure is in all other respects similar to the branch-and-bound procedure for integer programming.

Comments and References

Many books have been written on linear programming, integer programming and combinatorial optimization. Examples of some relatively recent ones are Papadimitriou and Steiglitz (1982), Parker and Rardin (1988), Nemhauser and Wolsey (1988), Du and Pardalos (1998), Schrijver (1998), Wolsey (1998), and Schrijver (2003).

Blazewicz, Dror and Weglarz (1991) give an overview of mathematical programming formulations of machine scheduling problems. The thesis by Van de Velde (1991) contains many examples (and references) of integer programming formulations for scheduling problems. Dauzère-Pérès and Sevaux (1998) present an interesting comparison of four different integer programming formulations for $1 \mid r_j \mid \sum U_j$.

The first example of branch-and-bound with Lagrangean relaxation applied to scheduling is due to Fisher (1976); he develops a solution method for $1 \mid\mid \sum w_j T_j$. Fisher (1981) presents an overview of the Lagrangean Relaxation method for solving integer programming problems in general. Dauzère-Pérès and Sevaux (2002) and Baptiste, Peridy and Pinson (2003) apply Lagrangean relaxation to $1 \mid r_j \mid \sum w_j U_j$. Barnhart, Johnson, Nemhauser, Savelsbergh and Vance (1998) provide an excellent general overview of branch-and-price (column generation) and Van den Akker, Hoogeveen and Van de Velde (1999) as well as Chen and Powell (1999) apply this technique specifically to scheduling.

Savelsbergh, Uma, and Wein (2005) have done a thorough experimental study of LP-based approximation algorithms for scheduling problems.

Appendix B

Deterministic and Stochastic Dynamic Programming

Dynamic programming is one of the more widely used techniques for dealing with combinatorial optimization problems. Dynamic Programming can be applied to problems that are solvable in polynomial time, as well as problems that cannot be solved in polynomial time (see Appendix C). It has proven to be very useful for stochastic problems as well.

B.1 Deterministic Dynamic Programming

Dynamic programming is basically a complete enumeration scheme that attempts, via a divide and conquer approach, to minimize the amount of computation to be done. The approach solves a series of subproblems until it finds the solution of the original problem. It determines the optimal solution for each subproblem and its contribution to the objective function. At each iteration it determines the optimal solution for a subproblem, which is larger than all previously solved subproblems. It finds a solution for the current subproblem by utilizing all the information obtained before in the solutions of all the previous subproblems.

Dynamic programming is characterized by three types of equations, namely

(i) initial conditions;
(ii) a recursive relation and
(iii) an optimal value function.

In scheduling, a choice can be made between forward dynamic programming and backward dynamic programming. The following example illustrates the use of forward dynamic programming.

Example B.1.1 (A Forward Dynamic Programming Formulation)
Consider $1 \mid\mid \sum h_j(C_j)$. This problem is a very important problem in scheduling theory as it comprises many of the objective functions studied in Part I of the book. The problem is, for example, a generalization of $1 \mid\mid \sum w_j T_j$ and is therefore NP-hard in the strong sense. Let J denote a subset of the n jobs and assume the set J is processed first. Let

$$V(J) = \sum_{j \in J} h_j(C_j),$$

provided the set of jobs J is processed first. The dynamic programming formulation of the problem is based on the following initial conditions, recursive relation and optimal value function.

Initial Conditions:

$$V(\{j\}) = h_j(p_j), \qquad j = 1, \ldots, n$$

Recursive Relation:

$$V(J) = \min_{j \in J} \left(V(J - \{j\}) + h_j(\sum_{k \in J} p_k) \right)$$

Optimal Value Function:

$$V(\{1, \ldots, n\})$$

The idea behind this dynamic programming procedure is relatively straightforward. At each iteration the optimal sequence for a subset of the jobs (say a subset J which contains l jobs) is determined, assuming this subset goes first. This is done for *every* subset of size l. There are $n!/(l!(n-l)!)$ subsets. For each subset the contribution of the l scheduled jobs to the objective function is computed. Through the recursive relation this is expanded to every subset which contains $l+1$ jobs. Each one of the $l+1$ jobs is considered as a candidate to go first. When using the recursive relation the actual sequence of the l jobs of the smaller subset does not have to be taken into consideration; only the contribution of the l jobs to the objective has to be known. After the value $V(\{1, \ldots, n\})$ has been determined the optimal sequence is obtained through a simple backtracking procedure.

The computational complexity of this problem can be determined as follows. The value of $V(J)$ has to be determined for all subsets that contain l jobs. There are $n!/l!(n-l)!$ subsets. So the total number of evaluations that have to be done are

$$\sum_{l=1}^{n} \frac{n!}{l!(n-l)!} = O(2^n). \qquad ||$$

Example B.1.2 (An Application of Forward Dynamic Programming)

Consider the problem described in the previous example with the following jobs.

jobs	1	2	3
p_j	4	3	6
$h_j(C_j)$	$C_1 + C_1^2$	$3 + C_2^3$;	$8C_3$

So $V(\{1\}) = 20$, $V(\{2\}) = 30$ and $V(\{3\}) = 48$. The second iteration of the procedure considers all sets containing two jobs. Applying the recursive relation yields

$$\begin{aligned} V(\{1,2\}) &= \min\Big(V(\{1\}) + h_2(p_1+p_2), V(\{2\}) + h_1(p_2+p_1)\Big) \\ &= \min(20+346,\ 30+56) = 86 \end{aligned}$$

So if jobs 1 and 2 precede job 3, then job 2 has to go first and job 1 has to go second. In the same way it can be determined that $V(\{1,3\}) = 100$ with job 1 going first and job 3 going second and that $V(\{2,3\}) = 102$ with job 2 going first and job 3 going second. The last iteration of the procedure considers set $\{1,2,3\}$.

$$V(\{1,2,3\}) = \min\Big(V(\{1,2\}) + h_3(p_1+p_2+p_3), V(\{2,3\}) + h_1(p_1+p_2+p_3),$$

$$V(\{1,3\}) + h_2(p_1+p_2+p_3)\Big).$$

So

$$V(\{1,2,3\}) = \min\Big(86+104, 102+182, 100+2200\Big) = 190.$$

It follows that jobs 1 and 2 have to go first and job 3 last. The optimal sequence is $2,1,3$ with objective value 190. ||

In the following example the same problem is handled through the backward dynamic programming procedure. In scheduling problems the backward version typically can be used only for problems with a makespan that is schedule independent (e.g., single machine problems without sequence dependent setups, multiple machine problems with jobs that have identical processing times).

The use of backward dynamic programming is nevertheless important as it is somewhat similar to the dynamic programming procedure discussed in the next section for stochastic scheduling problems.

Example B.1.3 (A Backward Dynamic Programming Formulation)
Consider again $1 \mid\mid \sum h_j(C_j)$. It is clear that the makespan $C_{\max}$ is schedule independent and that the last job is completed at $C_{\max}$ which is equal to the sum of the n processing times.

Again, J denotes a subset of the n jobs and it is assumed that J is processed first. Let J^C denote the complement of J. So set J^C is processed last. Let $V(J)$ denote the minimum contribution of the set J^C to the objective function. In other words, $V(J)$ represent the minimum additional cost to complete all *remaining* jobs after all jobs in set J already have been completed.

The backward dynamic programming procedure is now characterized by the following initial conditions, recursive relation and optimal value function.

Initial Conditions:

$$V(\{1,\ldots,j-1,j+1,\ldots,n\}) = h_j(C_{\max}) \qquad j = 1,\ldots,n$$

Recursive Relation:

$$V(J) = \min_{j \in J^C} \Big(V(J \cup \{j\}) + h_j(\sum_{k \in J \cup \{j\}} p_k) \Big)$$

Optimal Value Function:

$$V(\emptyset)$$

Again, the procedure is relatively straightforward. At each iteration, the optimal sequence for a subset of the n jobs, say a subset J^C of size l, is determined, assuming this subset goes *last*. This is done for every subset of size l. Through the recursive relation this is expanded for every subset of size $l+1$. The optimal sequence is obtained when the subset comprises all jobs. Note that, as in Example B.1.1, subset J goes first; however, in Example B.1.1 set J denotes the set of jobs already scheduled while in this example set J denotes the set of jobs still to be scheduled. ||

Example B.1.4 (An Application of Backward Dynamic Programming)
Consider the same instance as in Example B.1.2. The makespan $C_{\max}$ is 13. So

$$V(\{1,3\}) = h_2(C_{\max}) = 2200$$
$$V(\{1,2\}) = h_3(C_{\max}) = 104$$
$$V(\{2,3\}) = h_1(C_{\max}) = 182$$

The second iteration of the procedure results in the following recursive relations.

$$V(\{1\}) = \min\Big(V(\{1,2\}) + h_2(p_1+p_2), V(\{1,3\}) + h_3(p_1+p_3)\Big)$$
$$= \min(104+346, 2200+80) = 450.$$

In the same way $V(\{2\})$ and $V(\{3\})$ can be determined: $V(\{2\}) = 160$ and $V(\{3\}) = 914$. The last iteration results in the recursive relation

$$V(\emptyset) = \min\Big(V(\{1\}) + h_1(p_1), V(\{2\}) + h_2(p_2), V(\{3\}) + h_3(p_3)\Big)$$
$$= \min\Big(450+20,\ 160+30,\ 914+48\Big) = 190. \qquad ||$$

Of course, dynamic programming can also be used for problems that are polynomial time solvable. Examples of such dynamic programming algorithms are the $O(n^2)$ procedure for $1 \mid prec \mid h_{\max}$ and the pseudopolynomial time $O(n^4 \sum p_j)$ procedure for $1 \mid\mid \sum T_j$ described in Chapter 3.

Dynamic programming concepts can also be used to prove the optimality of certain rules, e.g., LRPT for $Pm \mid prmp \mid C_{\max}$.

The proofs are then done through a combination of induction and contradiction. The induction argument assumes that the priority rule is optimal for $k-1$ jobs. In order to show that the rule is optimal for k jobs a contradiction argument is used. Assume that at time zero an action is taken that is not prescribed by the priority rule. At the first job completion there is one less job, i.e., $k-1$ jobs, and the scheduler has to revert back to the priority rule because of the induction hypothesis.

It has to be shown now that starting out at time zero following the priority rule results in a lower objective than not acting according to the priority rule at time zero and switching over to the priority rule at the first job completion. This proof technique is usually applied in a preemptive setting. The proof of optimality of the LRPT rule for $Pm \mid prmp \mid C_{\max}$ in Section 5.2 is an example of this technique.

B.2 Stochastic Dynamic Programming

Dynamic programming is often used in stochastic sequential decision processes, especially when the random variables are exponentially distributed. This class of decision processes is usually referred to as *Markovian Decision Processes (MDP's)*. An MDP can be characterized, in the same way as a deterministic dynamic program, by

(i) initial conditions;
(ii) a recursive relation and
(iii) an optimal value function.

The setup of an MDP formulation of a scheduling problem is fairly similar to the setup of a backward dynamic program as described in Example B.1.3.

Example B.2.1 (An MDP Formulation of a Stochastic Scheduling Problem)

Consider the following stochastic counterpart of $Pm \mid prmp \mid C_{\max}$ with m machines in parallel and n jobs. The processing time of job j is exponentially distributed with rate λ_j. Consider a particular time t. Let J denote the set of jobs already completed and let J^C the set of jobs still in the process. Let $V(J)$ denote the expected value of the remaining completion time under the optimal schedule when the set of jobs J already have been completed. In this respect notation $V(J)$ is somewhat similar to notation used in Example B.1.3. The following initial conditions, recursive relation and optimal value function characterize this Markov Decision Process.

Initial Conditions:

$$V(\{1,\ldots,j-1,j+1,\ldots,n\}) = \frac{1}{\lambda_j}$$

Recursive Relation:

$$V(J) = \min_{j,k \in J^C} \Big(\frac{1}{\lambda_j + \lambda_k} + \frac{\lambda_j}{\lambda_j + \lambda_k} V(J \cup \{j\}) + \frac{\lambda_k}{\lambda_j + \lambda_k} V(J \cup \{k\}) \Big)$$

Optimal Value Function:

$$V(\emptyset)$$

The initial conditions are clear. If only job j remains to be completed, then the expected time till all jobs have completed their processing is, because of the memoryless property, $1/\lambda_j$. The recursive relation can be explained as follows. Suppose two or more jobs remain to be completed. If jobs j and k are selected for processing, then the expected remaining time till all jobs have completed their processing can be computed by conditioning on which one of the two jobs finishes first with its processing. The first completion occurs after an expected time $1/(\lambda_j + \lambda_k)$. With probability $\lambda_j/(\lambda_j + \lambda_k)$ it is job j that is completed first; the expected remaining time needed to complete remaining jobs is then $V(J \cup \{j\})$. With probability $\lambda_k/(\lambda_j + \lambda_k)$ it is job k that is completed first; the expected time needed to complete then all remaining jobs is $V(J \cup \{k\})$. ||

Dynamic programming is also used for stochastic models as a basis to verify the optimality of certain priority rules. The proofs are then also done through a combination of induction and contradiction, very much in the same way as they are done for the deterministic models.

Comments and References

Many books have been written on deterministic as well as on stochastic dynamic programming. See, for example, Denardo (1982), Ross (1983) and Bertsekas (1987).

Appendix C

Constraint Programming

In contrast to mathematical programming, which has its roots in the Operations Research community, constraint programming has its origins in the Artificial Intelligence and Computer Science communities. Constraint programming can be traced back to the constraint satisfaction problems studied in the 1970's. A constraint satisfaction problem requires a search for a feasible solution that satisfies all given constraints. To facilitate the search for a solution to such a problem various special purpose languages have been developed, e.g., Prolog. However, during the last decade of the twentieth century, constraint programming has not only been used for solving feasibility problems, but also for solving optimization problems. Several approaches have been developed that facilitate the application of constraint programming to optimization problems. One such approach is via the Optimization Programming Language (OPL), which was designed for modeling and solving optimization problems through both constraint programming techniques and mathematical programming procedures.

C.1 Constraint Satisfaction

To describe the constraint programming framework, it is necessary to first define the constraint satisfaction problem. In order to be consistent with the mathematical programming material presented in Appendix A, it is advantageous to present the constraint satisfaction problem using mathematical programming terminology. Assume n decision variables $x_1, \ldots, x_n$ and let $\mathcal{D}_j$ denote the set of allowable values for decision variable x_j. This set is typically referred to as

the *domain* of the variable x_j. Decision variables can take integer values, real values, as well as set elements.

Formally, a constraint is a mathematical relation that implies a subset $\mathcal{S}$ of the set $\mathcal{D}_1 \times \mathcal{D}_2 \times \cdots \times \mathcal{D}_n$ such that if $(x_1, \ldots, x_n) \in \mathcal{S}$, then the constraint is said to be satisfied. One can also define a mathematical function f such that

$$f(x_1, \ldots, x_n) = 1$$

if and only if the constraint is satisfied. Using this notation, the Constraint Satisfaction Problem (CSP) can be defined as follows:

$$\begin{aligned} f_i(x_1, \ldots, x_n) &= 1, \qquad i = 1, \ldots, m \\ x_j &\in \mathcal{D}_j \qquad j = 1, \ldots, n \end{aligned}$$

Since the problem is only a feasibility problem, no objective function has to be defined. The constraints in the constraint set may be of various different types: they may be linear, nonlinear, logical combinations of other constraints, cardinality constraints, higher order constraints or global constraints.

One basic difference between the constraints in a mathematical programming formulation and the constraints in a constraint satisfaction problem lies in the fact that the constraints in a mathematical programming formulation are typically either linear or nonlinear, whereas the constraints in a constraint programming formulation can be of a more general form.

Constraint Satisfaction is typically solved via a tree search algorithm. Each node in the search tree corresponds to a set of domains $\mathcal{D}'_1 \times \mathcal{D}'_2 \times \cdots \times \mathcal{D}'_n$ such that $\mathcal{D}'_j \subset \mathcal{D}_j$. In other words, a node is nothing more than a contraction of the original domains that has not yet proven infeasible. The tree search algorithm can branch from one node to another by assigning a value to a problem variable. At each node, the following operations have to be performed:

```
WHILE not solved AND not infeasible DO
    consistency checking (domain reduction)
    IF a dead-end is detected THEN
        try to escape from dead-end (backtrack)
    ELSE
        select variable
        assign value to variable
    ENDIF
ENDWHILE
```

The selection of the next variable and the assignment of its value is done by variable selection heuristics and value assignment heuristics. In job shop scheduling a variable typically corresponds to an operation and the value corresponds to its starting time. After a value is assigned to a variable, inconsistent values of unassigned variables are deleted. The process of removing inconsistent values is

often referred to as consistency checking or domain reduction. One well-known technique of consistency checking is *constraint propagation*. For a variable x, the *current domain* $\delta(x)$ is the set of values for which no inconsistency can be found with the available consistency checking techniques. If, after removing inconsistent values from the current domains, a current domain has become empty, a so-called *dead-end* has been reached. A dead-end means that either the original problem is infeasible or that some of the branching decisions made from the root of the tree down to this point has created an infeasibility. In such a case, the algorithm has to backtrack; that is, one or more assignments of variables have to be undone and alternatives have to be tried out. An instance is solved if every variable is assigned a value; an instance is shown to be infeasible if for a variable in the root of the tree no values are remaining to be tried.

If constraint satisfaction is applied to the job shop problem in Chapter 7, then the problem is to verify whether there exists a feasible job shop schedule with a makespan $C_{\max}$ that is less than a given value z^*.

Domain reduction in job shop scheduling boils down to the following: Given the partial schedule already constructed, each operation yet to be scheduled has an earliest possible starting time and a latest possible completion time (which are basically equivalent to a release date and a due date). Whenever the starting time and completion time of an operation are fixed, some form of checking has to be done on how the newly scheduled (fixed) operation affects the earliest possible starting times and latest possible completion times of all the operations that still remain to be scheduled. The earliest possible starting time of a yet to be scheduled operation may now have to be set later while the latest possible completion time of that operation may now have to be set earlier.

Note that constraint satisfaction requires the specification of an actual makespan; this allows the procedure to construct a schedule going forward in time as well as backward in time. A branch-and-bound approach, on the other hand, often constructs a schedule going forward in time whenever the makespan is not known a priori (see, for example, Sections 3.2 and 7.1).

C.2 Constraint Programming

Originally, constraint satisfaction was used only to find feasible solutions for problems. However, the constraint satisfaction structure, when embedded in a more elaborate framework, can be applied to optimization (e.g., minimization) problems as well. An optimization problem may be formulated as follows:

$$\text{minimize} \quad g(x_1, \ldots, x_n)$$

subject to

$$\begin{aligned} f_i(x_1, \ldots, x_n) &= 1, \qquad i = 1, \ldots, m \\ x_j &\in \mathcal{D}_j \qquad j = 1, \ldots, n \end{aligned}$$

The standard search procedure for finding the optimal solution is to first find a feasible solution to the Constraint Satisfaction Problem, while ignoring the objective function. Let $y_1, \ldots, y_n$ represent such a feasible solution. Let $z^* = g(y_1, \ldots, y_n)$ and add the constraint

$$g(x_1, \ldots, x_n) < z^*$$

to the constraint set and solve this modified Constraint Satisfaction Problem. The additional constraint forces the new feasible solution to have a better objective value than the current one. Constraint propagation may cause the domains of the decision variables to be narrowed, thus reducing the size of the search space. As the search goes on, new solutions must have progressively better objective values. The algorithm terminates when no feasible solution is found; when this happens, the last feasible solution found is optimal.

A more sophisticated and efficient search procedure, often referred to as *dichotomic* search, requires at the outset a good lower bound $\mathcal{L}$ on the objective $g(x_1, \ldots, x_n)$. The procedure must also find an initial feasible solution that represents an upper bound $\mathcal{U}$ on the objective function. The dichotomic search procedure essentially performs a binary search on the objective function. The procedure computes the midpoint

$$\mathcal{M} = \frac{(\mathcal{U} + \mathcal{L})}{2}$$

of the two bounds and then solves the constraint satisfaction problem with the added constraint

$$g(x_1, \ldots, x_n) < \mathcal{M}$$

If it finds a feasible solution, then the new upper bound is set equal to $\mathcal{M}$ and the lower bound is kept the same; a new midpoint is computed and the search continues. If it does not find a feasible solution, then the lower bound is set equal to $\mathcal{M}$ and the upper bound is kept the same; a new midpoint is computed and the search continues.

Dichotomic search is effective when the lower bound is strong, because the computational time that is needed to show that a constraint satisfaction problem does not have a feasible solution may be very long.

Constraint programming can be described as a two-level architecture that at one level has a list of constraints (sometimes referred to as a constraint store) and at the other level a programming language component that specifies the organization and the sequence in which the basic operations are executed. One fundamental feature of the programming language component is its ability to specify a search procedure.

C.3 An Example of a Constraint Programming Language

The Optimization Programming Language (OPL) was developed during the last decade of the twentieth century to deal with optimization problems through a combination of constraint programming techniques and mathematical programming techniques.

This section describes a constraint program designed for the job shop problem $Jm \mid\mid C_{\max}$ that is considered in Chapter 7.

Example C.3.1 (An OPL Program for a Job Shop)

An OPL program for the job shop problem $Jm \mid\mid C_{\max}$ can be written as follows:

```
01. int nbMachines = ...;
02. range Machines 1..nbMachines;
03. int nbJobs = ...;
04. range Jobs 1..nbJobs;
05. int nbOperations = ...;
06. range Operations 1..nbOperations;
07. Machines resource[Jobs,Operations]= ...;
08. int+ duration[Jobs,Operations] = ... ;
09. int totalDuration=sum(j in Jobs, o in Operations)duration[j,o];
10. scheduleHorizon=totalDuration;
11. Activity operation[j in Jobs, o in Operations](duration[j,o]);
12. Activity makespan(0);
13. UnaryResource tool[Machines];
14. minimize
15.       Makespan.end
16. subject to {
17.       forall(j in Jobs)
18.            Operation[j,nbOperations] precedes makespan;
19.       forall(j in Jobs)
20.            forall(o in 1,..nbOperations-1)
21.                 Operation[j,o] precedes Operation[j,o+1];
22.       forall(j in Jobs)
23.            forall(o in Operations)
24.                 Operation[j,o] requires tool[resource[j,o]];
25. };
```

The first six lines define the input data of the problem, consisting of the number of machines, the number of jobs, and the number of operations. The array resource contains input data that consists of the identity and the characteristics of the machine needed for processing a particular operation of a job. The array duration specifies the time required for processing each operation of a job. The keyword Activity implicitly indicates a decision variable. An activity consists of three elements: a start time, an end time

and a duration. In the declaration given here, the durations of each activity are given as data. When an activity is declared, the constraint

```
OPERATION[J,O].START + OPERATION[J,O].DURATION =
                       OPERATION[J,O].END
```

is automatically included in the program. The makespan activity is a dummy activity with zero processing time that will be the very last operation in the schedule. UNARYRESOURCE implies also a decision variable; decisions have to be made which activities have to be assigned to which resources at any given time.

The remaining part of the program states the problem: first the objective and then all the constraints. The statement PRECEDES is a keyword of the language. So one of the constraints is internally immediately translated into

```
OPERATION[J,O].END <= OPERATION[J,O+1].START
```

The word REQUIRES is also a keyword of the language. Declaring a set of requirements causes the creation of a set of so-called *disjunctive* constraints (see Appendix A). For example, let the resource TOOL[I] denote a given machine I and let OPERATION[J1,O1] and OPERATION[J2,O2] denote two operations which both require machine I, i.e., the data include

```
RESOURCE[J1,O1]=RESOURCE[J2,O2]=I
```

The following disjunctive constraint is now created automatically by the system, ensuring that the two operations cannot occupy machine I at the same time:

```
OPERATION[J1,O1].END<= OPERATION[J2,O2].START
                    or
OPERATION[J2,O2].END<= OPERATION[J1,O1].START
```

These disjunctive constraints imply that OPERATION[J1,O1] either precedes or follows OPERATION[J2,O2]. ||

C.4 Constraint Programming vs. Mathematical Programming

Constraint programming, as a modelling tool, is more flexible than mathematical programming. The objective functions as well as the constraints can be of a more general nature, since the decision variables may represent besides integer values and real values also set elements and even subsets of sets.

The ways in which searches for optimal solutions can be conducted in mathematical programming and in constraint programming have similarities as well as differences. One similarity between constraint programming and the branch-and-bound method applied to integer programming is based on the fact that

both approaches typically search through a tree of which each one of the nodes represent a partial solution of a schedule. Both approaches have to deal with issues such as how to examine the search space effectively and how to evaluate all the alternatives as efficiently as possible (i.e., how to prune the tree).

One of the differences between the dichotomic search used in constraint programming and a branch-and-bound procedure for a mixed-integer programming problem lies in the fact that the dichotomic search stresses a search for feasible solutions, whereas a branch-and-bound procedure emphasizes improvements in lower bounds.

Constraint programming techniques and mathematical programming techniques can be embedded within one framework. Such an integrated approach has already proven useful for a variety of scheduling problems.

Consider, for example, the application of a branch-and-price approach to the parallel machine problem $Pm \mid\mid \sum w_j C_j$. Such a problem is usually of the set partitioning type and the solution techniques are typically based on column generation. A column in this parallel machine problem represents a schedule for one machine, which is a feasible combination of a subset of the jobs. Since the potential number of columns is rather huge, it is important to be able to start the optimization with an initial subset of columns that is appropriate.

The solution strategy uses constraint programming in two ways. First, it is used when generating the initial subset of columns. Second, it is used as a subproblem algorithm for generating new columns during the optimization process. Thus, the master problem must find a set of appropriate single machine scheduling problems. The main program alternates between solving the linear programming relaxation of the set covering problem and solving a column generation problem that generates new columns for the master problem. Once the columns are satisfactory, the columns are fixed and the set covering problem is solved to find the integer optimal solution.

The constraint programming subproblem is interesting. Constraint programming is used to enumerate the potential single machine schedules. In the initialization stage a set of single machine schedules are generated that are guaranteed to cover every job. In the column generation phase the constraint program is used twice. First, the optimal single machine schedules with the current set of dual multipliers are determined. After this has been determined, a search is done for all single machine schedules with a reduced cost of at least half of the optimal schedule. This provides a large set of entering columns and eliminates one of the major weaknesses of column generation, namely the large number of iterations needed to improve the objective value in the master problem.

Constraint programming turns out to be also suitable for this preprocessing stage. A constraint programming routine can detect columns that are infeasible and reduce this way the number of columns to start out with.

Comments and References

There is an enormous literature on constraint satisfaction and on constraint programming. Even the part of the literature that focuses on scheduling is extensive. For general treatises on constraint programming, see Van Hentenryck and Michel (2005), Van Hentenryck and Lustig (1999), Hooker (2000), and Lustig and Puget (2001). The presentation in this appendix is mainly based on the paper by Lustig and Puget (2001).

A number of papers have focused on the application of constraint programming to scheduling; see, for example, Nuijten and Aarts (1996), Jain and Grossmann (2001), and Lustig and Puget (2001). Various papers considered the integration of constraint programming techniques with mathematical programming; see, for example, Grönkvist (2002) and the conference proceedings edited by Régin and Rueher (2004).

Appendix D

Complexity Theory

Complexity theory is based on a mathematical framework developed by logicians and computer scientists. This theory was developed to study the intrinsic difficulty of algorithmic problems and has proven very useful for combinatorial optimization. This appendix presents a brief overview of this theory and its ramifications for scheduling.

The applications discussed again concern scheduling problems. In order to understand these examples the reader should be familiar with the notation introduced in Chapter 2.

D.1 Preliminaries

In complexity theory the term "problem" refers to the generic description of a problem. For example, the scheduling problem $Pm \mid\mid C_{\max}$ is a problem, as is the linear programming problem described in the first section of Appendix A. The term "instance" refers to a problem with a given set of numerical data. For example, the setting with two machines, five jobs with processing times 2, 3, 5, 5, 8, and the makespan as objective is an instance of the $Pm \mid\mid C_{\max}$ problem.

With each instance there is a "size" associated. The size of an instance refers to the length of the data string necessary to specify the instance. It is also referred to as the length of the encoding. For example, consider instances of the problem $Pm \mid\mid C_{\max}$ and assume that a convention is made to encode every instance in a certain way. First the number of machines is specified, followed by the number of jobs and then the processing times of each one of n jobs. The different pieces of data may be separated from one another by comma's which

perform the functions of separators. The instance of the $Pm \mid\mid C_{\max}$ problem described above is then encoded as

$$2, 5, 2, 3, 5, 5, 8.$$

One could say that the size of the instance under this encoding scheme is 7 (not counting the separators). Of course, the length of the encoding depends heavily on the encoding conventions. In the example above the processing times were encoded using the decimal system. If all the data were presented in binary form the length of the encoding changes. The example then becomes

$$10, 101, 10, 11, 101, 101, 1000.$$

The length of the encoding is clearly larger. Another form of encoding is the unary encoding. Under this encoding the integer number n is represented by n ones. According to this encoding the example above becomes

$$11, 11111, 11, 111, 11111, 11111, 11111111.$$

Clearly, the size of an instance under unary encoding is larger than that under binary encoding. The size of an instance depends not only on the number of jobs but also on the length of the processing times of the jobs. One instance is larger than another instance with an equal number of jobs if all the processing times in the first instance are larger than all the processing times in the second instance.

However, in practice, the size of an instance is often simply characterized by the number of jobs (n). Although this may appear at first sight somewhat crude, it is sufficiently accurate for making distinctions between the complexities of different problems. In this book the size of an instance is measured by the number of jobs.

The efficiency of an algorithm for a given problem is measured by the maximum (worst-case) number of computational steps needed to obtain an optimal solution as a function of the size of the instance. This, in turn, requires a definition of a computational step. In order to define a computational step, a standard model of computing is used, the *Turing* machine. Any standard text on computational complexity contains the assumptions of the Turing machine; however, these assumptions are beyond the scope of this appendix. In practice however, a computational step in an algorithm is either a comparison, a multiplication or any data manipulation step concerning one job. The efficiency of an algorithm is then measured by the maximum number of computational steps needed to obtain an optimal solution (as a function of the size of the instance, i.e., the number of jobs). The number of computational steps may often be just the maximum number of iterations the algorithm has to go through. Even this number of iterations is typically approximated.

For example, if careful analysis of an algorithm establishes that the maximum number of iterations needed to obtain an optimal solution is $1500 + 100n^2 +$

$5n^3$, then only the term which, as a function of n, increases the fastest is of importance. This algorithm is then referred to as an $O(n^3)$ algorithm. In spite of the fact that for small n the first two terms have a larger impact on the number of iterations, these first two terms are not of interest for large scale problems. For large numbers of n the third term has the largest impact on the maximum number of iterations required. Even the coefficient of this term (the 5) is not that important. An $O(n^3)$ algorithm is usually referred to as a polynomial time algorithm; the number of iterations is polynomial in the size (n) of the problem. Such a polynomial time algorithm is in contrast to an algorithm that is either $O(4^n)$ or $O(n!)$. The number of iterations in an $O(4^n)$ or an $O(n!)$ algorithm is, in the worst case, exponential in the size of the problem.

Some of the easiest scheduling problems can be solved through a simple priority rule, e.g., WSPT, EDD, LPT and so on. To determine the optimal order then requires a simple sorting of the jobs based on one or two parameters. In the following example it is shown that a simple sort can be done in $O(n\ log(n))$ time.

Example D.1.1 (MERGESORT)

The input of the algorithm is a sequence of numbers $x_1, \ldots, x_n$ and the desired output is the sequence $y_1, \ldots, y_n$, a permutation of the input, satisfying $y_1 \leq y_2 \leq \cdots \leq y_n$. One procedure for this problem is in the literature referred to as *MERGESORT*. This method takes two sorted sequences $\mathcal{S}_1$ and $\mathcal{S}_2$ of equal size as its input and produces a single sequence $\mathcal{S}$ containing all elements of $\mathcal{S}_1$ and $\mathcal{S}_2$ in sorted order. This algorithm repeatedly selects the larger of the largest elements remaining in $\mathcal{S}_1$ and $\mathcal{S}_2$ and then deletes the element selected. This recursive procedure requires for n elements the following number of comparisons:

$$T(1) = 0$$

$$T(n) = 2T\left(\frac{n}{2}\right) + (n-1)$$

It can be shown easily that if n is a power of 2, the total number of comparisons is $O(n\ log(n))$. Examples of problems that can be solved this way in polynomial time are $1 \mid\mid \sum w_j C_j$ and $1 \mid\mid L_{\max}$. ||

Example D.1.2 (Complexity of the Assignment Problem)

The assignment problem described in the first part of Appendix A can be solved in polynomial time; actually in $O(n^3)$. ||

Actually, any LP can be solved in polynomial time. However, in contrast to the linear program, there exists no polynomial time algorithm for the integer program.

D.2 Polynomial Time Solutions versus NP-Hardness

In complexity theory a distinction is made between *optimization* problems and *decision* problems. The question raised in a decision problem requires either a yes or a no answer. These decision problems are therefore often also referred to as yes-no problems. With every optimization problem one can associate a decision problem. For example, in the problem $Fm \mid\mid C_{\max}$ the makespan has to be minimized. In the associated decision problem the question is raised whether there exists a schedule with a makespan less than a given value z. It is clear that the optimization problem and the related decision problem are strongly connected. Actually, if there exists a polynomial time algorithm for the optimization problem, then there exists a polynomial time algorithm for the decision problem and vice versa. A fundamental concept in complexity theory is the concept of *problem reduction.* Very often it occurs that one combinatorial problem is a special case of another problem, or equivalent to another problem, or more general than another problem. Often, an algorithm that works well for one combinatorial problem works as well for another after only minor modifications. Specifically, it is said that problem P *reduces* to problem P′ if for any instance of P an equivalent instance of P′ can be constructed. In complexity theory usually a more stringent notion is used. Problem P *polynomially* reduces to problem P′ if a polynomial time algorithm for P′ implies a polynomial time algorithm for P. Polynomial reducibility of P to P′ is denoted by $\mathrm{P} \propto \mathrm{P}'$. If it is known that if there does not exist a polynomial time algorithm for problem P, then there does not exist a polynomial time algorithm for problem P′ either.

Some formal definitions of problem classes are now in order.

Definition D.2.1 (Class $\mathcal{P}$). *The class $\mathcal{P}$ contains all decision problems for which there exists a Turing machine algorithm that leads to the right yes-no answer in a number of steps bounded by a polynomial in the length of the encoding.*

The definition of the class $\mathcal{P}$ is based on the time it takes a Turing machine to *solve* a decision problem. There exists a larger class of problems that is based on the time it takes a Turing machine to *verify* whether a *given* solution of a decision problem is correct or not. This solution may be in the form of a clue, e.g., for a scheduling problem a clue may be a sequence or a schedule.

Definition D.2.2 (Class $\mathcal{NP}$). *The class $\mathcal{NP}$ contains all decision problems for which the correct answer, given a proper clue, can be verified by a Turing machine in a number of steps bounded by a polynomial in the length of the encoding.*

For example, if the decision problem associated with the $F3 \mid\mid C_{\max}$ problem is considered, then a proper clue could be an actual schedule or permutation of jobs which results in a certain makespan, which is less than the given fixed value z. In order to verify whether the correct answer is yes, the algorithm takes the given permutation and computes the makespan under this permutation in

order to verify that it is less than the given constant z. Verifying that a sequence satisfies such a condition is, of course, simpler than *finding* a sequence that satisfies such a condition. The class $\mathcal{P}$ is, clearly, a subclass of the class $\mathcal{NP}$.

One of the most important open issues in mathematical logic and combinatorial optimization is the question whether or not $\mathcal{P} = \mathcal{NP}$. If $\mathcal{P}$ were equal to $\mathcal{NP}$, then there would exist polynomial time algorithms for a very large class of problems for which up to now no polynomial time algorithm has been found.

Definition D.2.3 (NP-hardness). *A problem* P*, either a decision problem or an optimization problem, is called NP-hard if the entire class of* $\mathcal{NP}$ *problems polynomially reduces to* P*.*

Actually, not all problems within the NP-hard class are equally difficult. Some problems are more difficult than others. For example, it may be that a problem can be solved in polynomial time as a function of the size of the problem in unary encoding, while it cannot be solved in polynomial time as a function of the size of the problem in binary encoding. For other problems there may not exist polynomial time algorithms under either unary or binary encoding. The first class of problems are not as hard as the second class of problems. The problems in this first class are usually referred to as *NP-hard in the ordinary sense* or simply *NP-hard.* The algorithms for this class of problems are called *pseudo-polynomial.* The second class of problems are usually referred to as *strongly NP-hard.*

A great variety of decision and optimization problems have been shown to be either NP-hard or strongly NP-hard. Some of the more important decision problems in this class are listed below.

The most important NP-hard decision problem is a problem in Boolean logic, the so-called SATISFIABILITY problem. In this problem there are n Boolean variables $x_1, \ldots, x_n$. These variables may assume either the value 0 (false) or the value 1 (true). A so-called *clause* is a function of a subset of these variables. A variable x_j may appear in a clause as x_j or as its *negation* $\neg x_j$. If $x_j = 0$, then its negation $\neg x_j = 1$ and vice versa. The clause $(x_2 + \neg x_5 + x_1)$ is 1 (true) if at least one of the elements in the clause gives rise to a true value, i.e., $x_2 = 1$ and/or $x_5 = 0$ and/or $x_1 = 1$. An expression can consist of a number of clauses. For the entire expression to be true, there must exist an assignment of 0's and 1's to the n variables which makes *each* clause true. More formally, the SATISFIABILITY problem is defined as follows.

Definition D.2.4 (SATISFIABILITY). *Given a set of variables and a collection of clauses defined over the variables, is there an assignment of values to the variables for which each one of the clauses is true?*

Example D.2.5 (SATISFIABILITY)

Consider the expression

$$(x_1 + \neg x_4 + x_3 + \neg x_2)(\neg x_1 + \neg x_2 + x_4 + \neg x_3)(\neg x_2 + \neg x_3 + x_1 + x_5)(\neg x_5 + \neg x_1 + x_4 + x_2)$$

It can be verified easily that the assignment $x_1 = 0, x_2 = 0, x_3 = 0, x_4 = 0$ and $x_5 = 0$ gives a truth assignment to each one of the four clauses. ||

The following four NP-hard problems are very important from the scheduling point of view.

Definition D.2.6 (PARTITION). *Given positive integers $a_1, \ldots, a_t$ and*

$$b = (1/2) \sum_{j=1}^{t} a_j,$$

do there exist two disjoint subsets S_1 and S_2 such that

$$\sum_{j \in S_i} a_j = b$$

for $i = 1, 2$?

Definition D.2.7 (3-PARTITION). *Given positive integers $a_1, \ldots, a_{3t}, b$ with*

$$\frac{b}{4} < a_j < \frac{b}{2}, \qquad j = 1, \ldots, 3t,$$

and

$$\sum_{j=1}^{3t} a_j = tb,$$

do there exist t pairwise disjoint three element subsets $S_i \subset \{1, \ldots, 3t\}$ such that

$$\sum_{j \in S_i} a_j = b$$

for $i = 1, \ldots, t$?

Definition D.2.8 (HAMILTONIAN CIRCUIT). *For a graph $G = (N, A)$ with node set N and arc set A, does there exist a circuit (or tour) that connects all nodes in N exactly once?*

Definition D.2.9 (CLIQUE). *For a graph $G = (N, A)$ with node set N and arc set A, does there exist a clique of size c? That is, does there exist a set $N^* \subset N$, consisting of c nodes, such that for each distinct pair of nodes $u, v \in N^*$, the arc $\{u, v\}$ is an element of A?*

Problems of which the complexity is established through a reduction from PARTITION typically allow for pseudo-polynomial time algorithms and are therefore NP-hard in the ordinary sense.

D.3 Examples

This section contains a number of examples illustrating several simple problem reductions.

Example D.3.1 (The Knapsack Problem)

Consider the so-called *knapsack* problem, which is equivalent to the scheduling problem $1 \mid d_j = d \mid \sum w_j U_j$. It is clear why this problem is usually referred to as the knapsack problem. The value d refers to the size of the knapsack and the jobs are the items that have to be put into the knapsack. The size of item j is p_j and the benefit obtained by putting item j into the knapsack is w_j. It can be shown that PARTITION reduces to the knapsack problem by taking

$$\begin{aligned} n &= t, \\ p_j &= a_j, \\ w_j &= a_j, \\ d &= \frac{1}{2}\sum_{j=1}^{t} a_j = b, \\ z &= \frac{1}{2}\sum_{j=1}^{t} a_j = b. \end{aligned}$$

It can be verified that there exists a schedule with an objective value less than or equal to $1/2\sum_{j=1}^{n} w_j$ if and only if there exists a solution for the PARTITION problem. ||

Example D.3.2 (Minimizing Makespan on Parallel Machines)

Consider $P2 \mid\mid C_{\max}$. It can be shown that PARTITION reduces to this problem by taking

$$\begin{aligned} n &= t, \\ p_j &= a_j, \\ z &= \frac{1}{2}\sum_{j=1}^{t} a_j = b. \end{aligned}$$

It is trivial to verify that there exists a schedule with an objective value less than or equal to $1/2\sum_{j=1}^{n} p_j$ if and only if there exists a solution for the PARTITION problem. ||

The following example illustrates how $P2 \mid\mid C_{\max}$ can be solved in pseudo-polynomial time via a simple dynamic programming algorithm.

Example D.3.3 (Application of a Pseudo-Polynomial Time Algorithm)

Consider $P2 \mid\mid C_{\max}$ with 5 jobs.

jobs	1	2	3	4	5
p_j	7	8	2	4	1

The question is: does there exist a partition with a makespan equal to 11 (half of the sum of the processing times)? The following dynamic program (see Appendix B) results in an optimal partition. Let the indicator variable $I(j,z)$ be 1 if there is a subset of jobs $\{1,2,\ldots,j-1,j\}$ for which the sum of the processing times is exactly z and 0 otherwise. The values of $I(j,z)$ have to be determined for $j=1,\ldots,5$ and $z=0,\ldots,11$. The procedure basically fills in 0-1 entries in the $I(j,z)$ matrix row by row.

z	0	1	2	3	4	5	6	7	8	9	10	11
$j=1$	1	0	0	0	0	0	0	1	0	0	0	0
$j=2$	1	0	0	0	0	0	0	1	1	0	0	0
$j=3$	1	0	1	0	0	0	0	1	1	1	1	0
$j=4$	1	0	1	0	1	0	1	1	1	1	1	1
$j=5$	1	1	1	1	1	1	1	1	1	1	1	1

From the table entries it follows that there is a partition of the jobs which results in a makespan of 11. From simple backtracking it follows that jobs 4 and 1 have a total processing time of 11. Clearly, this table entry algorithm is polynomial in $n\sum p_j/2$. If the instance is encoded according to the *unary* format the length of the encoding is $O(\sum p_j)$. The algorithm is polynomial in the size of the problem. However, if the instance is encoded according to the *binary* format, the length of the encoding is $O(log(\sum p_j))$ and the algorithm, being of $O(n\sum p_j)$, is *not* bounded by any polynomial function of $O(log(\sum p_j))$. The algorithm is therefore pseudo-polynomial. ||

NP-hard problems of which the complexity is established via a reduction from SATISFIABILITY, 3-PARTITION, HAMILTONIAN CIRCUIT or CLIQUE are strongly NP-hard.

Example D.3.4 (Minimizing Makespan in a Job Shop)

Consider $J2 \mid rcrc, prmp \mid C_{\max}$. It can be shown that 3-PARTITION reduces to $J2 \mid rcrc, prmp \mid C_{\max}$. The reduction to the scheduling problem is based

on the following transformation. The number of jobs, n, is chosen to be equal to $3t + 1$. Let

$$p_{1j} = p_{2j} = a_j \qquad j = 1, \ldots, 3t.$$

Each one of these $3t$ jobs has to be processed first on machine 1 and then on machine 2. These $3t$ jobs do *not* recirculate. The last job, job $3t + 1$, has to start its processing on machine 2 and then has to alternate between machines 1 and 2. It has to be processed this way t times on machine 2 and t times on machine 1 and each one of these $2t$ processing times is equal to b. For a schedule to have a makespan

$$C_{\max} = 2tb,$$

this last job has to be scheduled without interruption. The remaining time slots can be filled without idle times by jobs $1, \ldots, 3t$ if and only if 3-PARTITION has a solution. ||

Example D.3.5 (Sequence Dependent Setup Times)

Consider the Travelling Salesman Problem or, in scheduling terms, the $1 \mid s_{jk} \mid C_{\max}$ problem. That HAMILTONIAN CIRCUIT can be reduced to $1 \mid s_{jk} \mid C_{\max}$ can be shown as follows. Let each node in the HAMILTONIAN CIRCUIT correspond to a city in the travelling salesman problem. Let the distance between two cities equal to 1 if there exists an arc between the two corresponding nodes in the HAMILTONIAN CIRCUIT. Let the distance between two cities equal to 2 if there does *not* exist an arc between the two corresponding nodes. The bound on the objective is equal to the number of nodes in the HAMILTONIAN CIRCUIT. It is easy to see that the two problems are equivalent. ||

Example D.3.6 (Scheduling with Precedence Constraints)

Consider $1 \mid p_j = 1, prec \mid \sum U_j$. That CLIQUE can be reduced to $1 \mid p_j = 1, prec \mid \sum U_j$ can be shown as follows. Given a graph $G = (N, A)$ and an integer c. Assume there are l nodes, i.e., $N = \{1, \ldots, l\}$. Let a denote the number of arcs, i.e., $a =\mid A \mid$, and let

$$\ell = \frac{c(c-1)}{2}.$$

So a clique of size c has exactly ℓ arcs.

The reduction involves two types of jobs: a node-job j correspond to node $j \in N$ and an arc-job (j, k) corresponds to arc $\{j, k\} \in A$. Each arc-job (j, k) is subject to the constraint that it must follow the two corresponding node-jobs j and k.

Consider the following instance of $1 \mid p_j = 1, prec \mid \sum U_j$. Let the number of jobs be $n = l + a$. All node-jobs have the same due date $l + a$, i.e., due date $d_j = l + a$ for $j = 1, \ldots, l$. All arc-jobs (j, k) have the same due date $c + \ell$,

i.e., due date $d_{(j,k)} = c + \ell$ for all $\{j,k\} \in A$. The precedence constraints are such that $j \rightarrow (j,k)$ and $k \rightarrow (j,k)$ for all $\{j,k\} \in A$. Let $z = a - \ell$.

A schedule with at least $l + \ell$ early jobs and at most z late jobs exists if and only if CLIQUE has a solution. If G has a clique of size c on a subset N^*, the corresponding node-jobs $j \in N^*$ have to be scheduled first. The ℓ corresponding arc-jobs can then be completed before their due dates by time $c + \ell$. The remaining $a - \ell$ arc-jobs are late and $\sum U_j = a - \ell = z$.

If G has no clique of size c, then at most $\ell - 1$ arc jobs can be on time and the treshold cannot be met. So the two problems are equivalent. ||

D.4 Approximation Algorithms and Schemes

Many scheduling problems are either NP-hard in the ordinary sense or strongly NP-hard. For these problems it will be clearly very hard to find an optimal solution in a time effective manner. It is of interest to develop for these problems polynomial time algorithms that can deliver, with some form of a guarantee, solutions close to optimal. This need has led to a significant amount of research in an area that is referred to as Approximation Algorithms or as Approximation Schemes. In order to state some necessary definitions, let ϵ be a small positive number and let $\rho = 1 + \epsilon$.

Definition D.4.1 (Approximation Algorithm). *An algorithm A is called a ρ-approximation algorithm for a problem, if for any instance I of that problem the algorithm A yields a feasible solution with objective value $A(I)$ such that*

$$| A(I) - OPT(I) | \leq \epsilon \cdot OPT(I).$$

The ρ value is usually referred to as the performance guarantee or the worst case ratio of the approximation algorithm A.

The definition above includes, of course, many approximation algorithms, including algorithms that are not very effective. The following definition focuses on more special classes of approximation algorithms, taking also their effectiveness into consideration.

Definition D.4.2 (Approximation Scheme).

(i) An Approximation Scheme for a given problem is a family of $(1 + \epsilon)$-approximation algorithms A_ϵ for that problem.

(ii) A Polynomial Time Approximation Scheme (PTAS) for a problem is an approximation scheme with a time complexity that is polynomial in the input size (e.g., the number of jobs n).

(iii) A Fully Polynomial Time Approximation Scheme (FPTAS) for a problem is an approximation scheme with a time complexity that is polynomial in the input size as well as in $1/\epsilon$.

So, it would be acceptable for a PTAS to have a time complexity $O(n^{2/\epsilon})$, even though this time complexity is exponential in $1/\epsilon$ (it is polynomial in

the size of the input which is exactly what is required in the definition of a PTAS). An FPTAS cannot have a time complexity that grows exponentially in $1/\epsilon$, but a time complexity $O(n^8/\epsilon^3)$ would be fine. With regard to worst case approximations, an FPTAS is the strongest possible result that one can obtain for an NP-hard problem.

A common approach for the design of approximation schemes is a technique that adds more structure to the input data. The main idea is to turn a difficult instance into a more simplified instance that is easier to handle. The optimal solution for the simplified instance can then be used to get a handle on the original instance. The approach basically consists of three steps: the first step involves a simplification of the instance at hand. The level of simplification depends on the desired precision ϵ. The second step involves the solution of the simplified instance. Solving the simplified instance typically can be done in polynomial time. In the third step the optimal solution of the simplified instance is translated back into an approximate solution for the original instance. This translation exploits the similarity between the original instance and the simplified instance.

Of course, finding the right simplification in the first step is more an art than a science. The following approaches for simplifying the input tend to work well, namely

(i) Rounding: the simplest way of adding structure to the input is to round some of the numbers of the input. For example, we may round non-integral due dates up to the closest integers.

(ii) Merging: a second way of adding structure is to merge small pieces into larger pieces of a primitive shape. For example, a huge number of tiny jobs can be merged into a single job with processing time equal to the total processing time of the tiny jobs.

(iii) Aligning: a third way of adding structure is to align the processing times of similar jobs. For example, we can replace 48 different jobs of roughly equal length with 48 identical jobs of the average length.

In order to illustrate the design of a PTAS following this approach consider the $P2 \mid\mid C_{\max}$ problem. In the previous section it was already shown that $P2 \mid\mid C_{\max}$ is NP-hard in the ordinary sense since it is equivalent to PARTITION. This problem turns out to be a nice candidate to illustrate the design of a PTAS.

In order to describe the PTAS some terminology and notation needs to be introduced. It is clear that

$$C_{\max}(OPT) \geq \max\left(p_{\max}, \frac{\sum_{j=1}^{n} p_j}{2}\right)$$

Denote this lower bound by $\mathcal{L}$. The jobs in an instance of $P2 \mid\mid C_{\max}$ can now be classified as being either *big* or *small*. This classification depends on a precision parameter $0 < \epsilon < 1$. A job is referred to as big if its processing time is larger than $\epsilon\,\mathcal{L}$ and as small if its processing time is less than or equal to $\epsilon\mathcal{L}$. Let

P^s denote the sum of the processing times of all small jobs and let P_1^s (P_2^s) denote the total processing time of the small jobs that in the optimal schedule are assigned to machine 1 (2).

The PTAS consists of three steps. In the first step the instance of the scheduling problem is transformed in a certain way into a new, more simplified, instance. The new instance will contain all the big jobs of the original instance with exactly the same processing times. However, it does not contain the original small jobs. Instead, the new instance contains $\lfloor P^s/(\epsilon\mathcal{L})\rfloor$ small jobs with all having the same length $\epsilon\mathcal{L}$. Note that the number of small jobs in the new instance is smaller than the number of small jobs in the original instance; the total processing of all the small jobs in the new instance may be slightly less than P^s.

It can be argued that the optimal makespan in the new instance, $C'_{\max}(OPT)$ is fairly close to the optimal makespan of the original instance $C_{\max}(OPT)$. The schedule for the new instance can be created as follows. Assign all the big jobs to the same machine as in the original instance. However, replace the original small jobs on machine i, $i = 1, 2$, by $\lceil P_i^s/(\epsilon\mathcal{L})\rceil$ small pieces of length $\epsilon\mathcal{L}$. Since

$$\lceil P_1^s/(\epsilon\mathcal{L})\rceil + \lceil P_2^s/(\epsilon\mathcal{L})\rceil \;\geq\; \lfloor P_1^s/(\epsilon\mathcal{L}) + P_2^s/(\epsilon\mathcal{L})\rfloor \;=\; \lfloor P^s/(\epsilon\mathcal{L})\rfloor,$$

this process assigns all small pieces of length $\epsilon\mathcal{L}$. By assigning the small pieces the load of machine i is increased by at most

$$\lceil P_i^s/(\epsilon\mathcal{L})\rceil\epsilon\mathcal{L} - P_i^s \;\leq\; (P_i^s/(\epsilon\mathcal{L}) + 1)\epsilon\mathcal{L} - P_i^s \;=\; \epsilon\mathcal{L}.$$

The resulting schedule is feasible for the new instance and it can be concluded that

$$C'_{\max}(OPT) \;\leq\; C_{\max}(OPT) + \epsilon\mathcal{L} \;\leq\; (1+\epsilon)C_{\max}(OPT).$$

Note that the stronger inequality $C'_{\max}(OPT) \leq C_{\max}(OPT)$ does not hold in general. Consider for example an instance that consists of six jobs of length 1 with $\epsilon = 2/3$. Then $C_{\max}(OPT) = 3$ and all the jobs are small. In the modified instance they are replaced by 3 small pieces of length 2, resulting in a $C'_{\max}(OPT) = 4$.

The second step involves the solution of the modified instance. When the small jobs in the original instance were replaced by small pieces of length $\epsilon\mathcal{L}$, the total processing time was not increased. Hence the total processing time of the new instance is at most $2\mathcal{L}$. Since each job in the new instance has a length of at least $\epsilon\mathcal{L}$, there can be at most $2\mathcal{L}/(\epsilon\mathcal{L}) = 2/\epsilon$ small jobs in the new instance. The number of jobs in the new instance is bounded by a finite constant that only depends on ϵ and thus is completely independent of the number of jobs in the original instance (n). The new instance can now be solved very easily. One simply can try all possible schedules! Since each of the $2/\epsilon$ jobs is assigned to one of the two machines, there are at most $2^{2/\epsilon}$ possible schedules and the makespan of each one of these schedules can be determined in $O(2/\epsilon)$ time. So

the new instance can be solved in constant time. (Clearly, this constant is huge and grows exponentially in $1/\epsilon$, but is still a constant.)

A third and last step remains to be done. The solution for the new instance has to be translated back into a solution for the original instance. Consider an optimal schedule σ' for the new instance. Let P_i', $i = 1, 2$, denote the total processing on machine i under this optimal schedule. Let $P_i^{b'}$ denote the total processing time of the big jobs and $P_i^{s'}$ the total processing of the small pieces on machine i under schedule σ'. Clearly, $P_i' = P_i^{b'} + P_i^{s'}$ and

$$P_1^{s'} + P_2^{s'} = \epsilon\mathcal{L} \cdot \lfloor P^s/(\epsilon\mathcal{L}) \rfloor > P^s - \epsilon\mathcal{L}.$$

The following schedule σ can now be constructed for the original instance. Every big job is put onto the same machine as in schedule σ'. In order to assign the small pieces to the two machines, reserve an interval of length $P_1^{s'} + 2\epsilon\mathcal{L}$ on machine 1 and an interval of length $P_2^{s'}$ on machine 2. The small jobs are now inserted greedily into these reserved intervals. First, start packing small jobs into the reserved interval on machine 1 until some job is encountered that does not fit any more. Since the size of a small job is at most $\epsilon\mathcal{L}$, the total size of the packed small jobs on machine 1 is at least $P_1^{s'} + \epsilon\mathcal{L}$. Then the total size of the unpacked jobs is at most $P^s - P_1^{s'} - \epsilon\mathcal{L}$, which is bounded from above by $P_2^{s'}$. So all remaining unpacked small jobs will fit together into the reserved interval on machine 2. This completes the description of schedule σ for the original instance.

Compare the loads P_1 and P_2 of the two machines in σ to the machine completion times P_1' and P_2' in schedule σ'. Since the total size of the small jobs on machine i is at most $P_i^{s'} + 2\epsilon\mathcal{L}$, it follows that

$$\begin{aligned} P_i &\leq P_i^{b'} + (P_i^{s'} + 2\,\epsilon\,\mathcal{L}) \\ &= P_i' + 2\,\epsilon\,\mathcal{L} \\ &\leq (1+\epsilon)\,C_{\max}(OPT) + 2\,\epsilon\,C_{\max}(OPT) \\ &= (1+3\,\epsilon)\,C_{\max}(OPT). \end{aligned}$$

Hence the makespan of the schedule σ is at most a factor $1+3\epsilon$ larger than the minimum makespan. Since 3ϵ can be made arbitrarily close to 0, the procedure serves as a PTAS for $P2 \,||\, C_{\max}$.

Summarizing, the algorithm can be described more formally as follows.

Algorithm D.4.3 (PTAS for Minimizing Makespan without Preemptions)

Step 0. (Initial Conditions)

Set $\mathcal{L} = \max(p_{\max}, \sum_{j=1}^{n} p_j)$. *Choose* ϵ *between 0 and 1.*
Partition the original job set into a subset of big jobs with $p_j > \epsilon\mathcal{L}$ *and a subset of small jobs with* $p_j \leq \epsilon\mathcal{L}$.
Set P^s *equal to the total processing of all small jobs.*

Step 1. (Problem Modification)

Construct a new instance and include each big job in the new instance. Add $\lfloor P^s/\epsilon\mathcal{L} \rfloor$ *small jobs of length* $\epsilon\mathcal{L}$.

Step 2. (Solution of Modified Problem)

Find an optimal schedule of the new instance via complete enumeration. Let $P_1^{s'}$ *(*$P_2^{s'}$*) denote the total amount of processing of the small pieces that are assigned to machine 1 (2).*

Step 3. (Solution of Original Problem)

Construct the schedule of the original instance by assigning the big jobs to the same machine as in the schedule obtained in Step 2. Assign the small jobs in the original instance one after another on machine 1 in a reserved space of size $P_1^{s'} + 2\epsilon\mathcal{L}$. *As soon as one of the small jobs does not fit in the remaining reserved space on machine 1, assign the remaining small jobs to machine 2.* ||

The PTAS described above is based on the approach of adding more structure to the input. There are two other approaches that are also very popular in the development of PTASs. The second approach is based on adding more structure to the output. The main idea here is to partition the output space, i.e., the set of all feasible solutions, into many smaller regions over which the optimization problem is easy to approximate. Tackling the problem for each such small region and taking the best approximate solution over all regions will then yield a globally good approximate solution.

The third approach for constructing approximation schemes is based on adding more structure to the execution of an algorithm. The main idea is to take an exact but slow algorithm, and interact with it while it is working. If the algorithm accumulates a lot of auxiliary data during its execution, then part of this data may be removed and the algorithm's memory may be cleaned. As a result the algorithm becomes faster (since there is less data to process) and generates an incorrect output (since the removal of data introduces errors). In the ideal case, the time complexity of the algorithm becomes polynomial and the incorrect output constitutes a good approximation of the true optimum.

Comments and References

The classic on computational complexity is Garey and Johnson (1979). A number of books have chapters on this topic; see, for example, Papadimitriou and Steiglitz (1982), Parker and Rardin (1988), Papadimitriou (1994), Schrijver (1998), and Wolsey (1998). The section on approximation algorithms and schemes is based on the tutorial by Schuurman and Woeginger (2007).

Appendix E

Complexity Classification of Deterministic Scheduling Problems

In scheduling theory it is often of interest to determine the borderline between polynomial time problems and NP-hard problems. In order to determine the exact boundaries it is necessary to find the "hardest" or the "most general" problems that still can be solved in polynomial time. These problems are characterized by the fact that any generalization, e.g., the inclusion of precedence constraints, results in NP-hardness, either in the ordinary sense or strongly. In the same vein it is of interest to determine the "simplest" or "least general" problems that are NP-hard, either in the ordinary sense or strongly. Making such a strongly NP-hard problem easier in any respect, e.g., setting all w_j equal to 1, results in a problem that is either solvable in polynomial time or NP-hard in the ordinary sense. In addition, it is also of interest to determine the most general problems that are NP-hard in the ordinary sense, but not strongly NP-hard.

A significant amount of research has focused on these boundaries. However, the computational complexity of a number of scheduling problems has not yet been determined and the borderlines are therefore still somewhat fuzzy.

In the following problem classification it is assumed throughout that the number of machines (m) is fixed. If an algorithm is said to be polynomial, then the algorithm is polynomial in the number of jobs n but not necessarily polynomial in the number of machines m. If a problem is said to be NP-Hard, either in the ordinary sense or strongly, then the assumption is made that the number of machines is fixed.

Table E.1 presents a sample of fairly general problems that are solvable in polynomial time. The table is organized according to machine environments. Some of the problems in this table are, however, not the *most* general problems solvable in polynomial time, e.g., $1 \mid\mid \sum U_j$ is a special case of the proportionate flow shop problem $Fm \mid p_{ij} = p_j \mid \sum U_j$. Also, the fact that $Fm \mid p_{ij} = p_j \mid$

SINGLE MACHINE	PARALLEL MACHINES	SHOPS
$1 \mid r_j, p_j = 1, prec \mid \sum C_j$	$P2 \mid p_j = 1, prec \mid L_{\max}$	$O2 \mid\mid C_{\max}$
$1 \mid r_j, prmp \mid \sum C_j$	$P2 \mid p_j = 1, prec \mid \sum C_j$	
$1 \mid tree \mid \sum w_j C_j$		$Om \mid r_j, prmp \mid L_{\max}$
	$Pm \mid p_j = 1, tree \mid C_{\max}$	
$1 \mid prec \mid L_{\max}$	$Pm \mid prmp, tree \mid C_{\max}$	$F2 \mid block \mid C_{\max}$
$1 \mid r_j, prmp, prec \mid L_{\max}$	$Pm \mid p_j = 1, outtree \mid \sum C_j$	$F2 \mid nwt \mid C_{\max}$
	$Pm \mid p_j = 1, intree \mid L_{\max}$	
$1 \mid\mid \sum U_j$	$Pm \mid prmp, intree \mid L_{\max}$	$Fm \mid p_{ij} = p_j \mid \sum C_j$
$1 \mid r_j, prmp \mid \sum U_j$		$Fm \mid p_{ij} = p_j \mid L_{\max}$
$1 \mid r_j, p_j = 1 \mid \sum w_j U_j$	$Q2 \mid prmp, prec \mid C_{\max}$	$Fm \mid p_{ij} = p_j \mid \sum U_j$
	$Q2 \mid r_j, prmp, prec \mid L_{\max}$	
$1 \mid r_j, p_j = 1 \mid \sum w_j T_j$		$J2 \mid\mid C_{\max}$
	$Qm \mid r_j, p_j = 1 \mid C_{\max}$	
	$Qm \mid p_j = 1, M_j \mid C_{\max}$	
	$Qm \mid r_j, p_j = 1 \mid \sum C_j$	
	$Qm \mid prmp \mid \sum C_j$	
	$Qm \mid p_j = 1 \mid \sum w_j C_j$	
	$Qm \mid p_j = 1 \mid L_{\max}$	
	$Qm \mid prmp \mid \sum U_j$	
	$Qm \mid p_j = 1 \mid \sum w_j U_j$	
	$Qm \mid p_j = 1 \mid \sum w_j T_j$	
	$Rm \mid\mid \sum C_j$	
	$Rm \mid r_j, prmp \mid L_{\max}$	

Table E.1 Polynomial Time Solvable Problems

$\sum U_j$ can be solved in polynomial time implies that $Fm \mid p_{ij} = p_j \mid L_{\max}$ can be solved in polynomial time as well.

Table E.2 presents a number of problems that are NP-hard in the ordinary sense. This table contains some of the simplest as well as some of the most general problems that fall in this class. The complexity of these problems is determined through a reduction from PARTITION. However, not for every one of these problems a pseudo-polynomial algorithm is known. For the problems followed by a $(*)$ there exists a pseudopolynomial time algorithm. For example, as no pseudo-polynomial time algorithm is known for $O2 \mid prmp \mid \sum C_j$, this problem may still turn out to be strongly NP-hard, even though there is a reduction from PARTITION.

Table E.3 contains problems that are strongly NP-hard. The problems tend to be the simplest problems that are strongly NP-hard. However, in this table

SINGLE MACHINE	PARALLEL MACHINES	SHOPS
$1 \mid\mid \sum w_j U_j$ (*) $1 \mid r_j, prmp \mid \sum w_j U_j$ (*) $1 \mid\mid \sum T_j$ (*)	$P2 \mid\mid C_{\max}$ (*) $P2 \mid r_j, prmp \mid \sum C_j$ $P2 \mid\mid \sum w_j C_j$ (*) $P2 \mid r_j, prmp \mid \sum U_j$ $Pm \mid prmp \mid \sum w_j C_j$ $Qm \mid\mid \sum w_j C_j$ (*) $Rm \mid r_j \mid C_{\max}$ (*) $Rm \mid\mid \sum w_j U_j$ (*) $Rm \mid prmp \mid \sum w_j U_j$	$O2 \mid prmp \mid \sum C_j$ $O3 \mid\mid C_{\max}$ $O3 \mid prmp \mid \sum w_j U_j$

Table E.2 NP-Hard Problems in the Ordinary Sense

SINGLE MACHINE	PARALLEL MACHINES	SHOPS
$1 \mid s_{jk} \mid C_{\max}$ $1 \mid r_j \mid \sum C_j$ $1 \mid prec \mid \sum C_j$ $1 \mid r_j, prmp, tree \mid \sum C_j$ $1 \mid r_j, prmp \mid \sum w_j C_j$ $1 \mid r_j, p_j = 1, tree \mid \sum w_j C_j$ $1 \mid p_j = 1, prec \mid \sum w_j C_j$ $1 \mid r_j \mid L_{\max}$ $1 \mid r_j \mid \sum U_j$ $1 \mid p_j = 1, chains \mid \sum U_j$ $1 \mid r_j \mid \sum T_j$ $1 \mid p_j = 1, chains \mid \sum T_j$ $1 \mid\mid \sum w_j T_j$	$P2 \mid chains \mid C_{\max}$ $P2 \mid chains \mid \sum C_j$ $P2 \mid prmp, chains \mid \sum C_j$ $P2 \mid p_j = 1, tree \mid \sum w_j C_j$ $R2 \mid prmp, chains \mid C_{\max}$	$F2 \mid r_j \mid C_{\max}$ $F2 \mid r_j, prmp \mid C_{\max}$ $F2 \mid\mid \sum C_j$ $F2 \mid prmp \mid \sum C_j$ $F2 \mid\mid L_{\max}$ $F2 \mid prmp \mid L_{\max}$ $F3 \mid\mid C_{\max}$ $F3 \mid prmp \mid C_{\max}$ $F3 \mid nwt \mid C_{\max}$ $O2 \mid r_j \mid C_{\max}$ $O2 \mid\mid \sum C_j$ $O2 \mid prmp \mid \sum w_j C_j$ $O2 \mid\mid L_{\max}$ $O3 \mid prmp \mid \sum C_j$ $J2 \mid rcrc \mid C_{\max}$ $J3 \mid p_{ij} = 1, rcrc \mid C_{\max}$

Table E.3 Strongly NP-Hard Problems

also there are some exceptions. For example, the fact that $1 \mid r_j \mid L_{\max}$ is strongly NP-hard implies that $1 \mid r_j \mid \sum U_j$ and $1 \mid r_j \mid \sum T_j$ are strongly NP-hard as well.

These tables have to be used in conjunction with the figures presented in Chapter 2. When attempting to determine the status of a problem that does not appear in any one of the three tables, it is necessary to search for related problems that are either easier or harder in order to determine the complexity status of the given problem.

Comments and References

The complexity classification of scheduling problems has its base in the work of Lenstra and Rinnooy Kan (1979), and Lageweg, Lawler, Lenstra and Rinnooy Kan (1981, 1982). Timkovsky (2000) discusses reducibility among scheduling problems and Brucker (2004) presents a very thorough and up-to-date complexity classification of scheduling problems.

A significant amount of research attention has focused on the complexity statuses of scheduling problems that are close to the boundaries; see, for example, Du and Leung (1989, 1990, 1993a, 1993b), Du, Leung and Wong (1992), Du, Leung and Young (1990, 1991), Leung and Young (1990), Timkovski (1998) and Baptiste (1999).

Appendix F

Overview of Stochastic Scheduling Problems

No framework or classification scheme has ever been introduced for stochastic scheduling problems. It is more difficult to develop such a scheme for stochastic scheduling problems than for deterministic scheduling problems. In order to characterize a stochastic scheduling problem more information is required. For example, the distributions of the processing times have to be specified as well as the distributions of the due dates (which may be different). It has to be specified whether the processing times of the n jobs are independent or correlated (e.g., equal to the same random variable) and also which class of policies is considered. For these reasons no framework has been introduced in this book either.

Table F.1 outlines a number of scheduling problems of which stochastic versions are tractable. This list refers to most of the problems discussed in Part 2 of the book. In the distribution column the distribution of the processing times is specified. If the entry in this column specifies a form of stochastic dominance, then the n processing times are arbitrarily distributed and ordered according to the form of stochastic dominance specified. The due dates in this table are considered fixed (deterministic).

However, the list in Table F.1 is far from complete. For example, it is mentioned that the stochastic counterpart of $Pm \mid\mid \sum C_j$ leads to the SEPT rule when the processing times are exponentially distributed. However, as stated in Chapter 12, a much more general result holds: if the processing times (from distributions $F_1, \ldots, F_n$) are independent, then SEPT is optimal provided the n distributions can be ordered stochastically.

Comparing Table F.1 with the tables in Appendix E reveals that there are a number of stochastic scheduling problems that are tractable while their deterministic counterparts are NP-Hard. The four NP-Hard deterministic problems are:

(i) $1 \mid r_j, prmp \mid \sum w_j C_j$,
(ii) $1 \mid d_j = d \mid \sum w_j U_j$,
(iii) $1 \mid d_j = d \mid \sum w_j T_j$,

DETERMINISTIC COUNTERPART	DISTRIBUTIONS	OPTIMAL POLICY	SECTION
$1 \mid\mid \sum w_j C_j$	arbitrary	WSEPT	10.1
$1 \mid r_j, prmp \mid \sum w_j C_j$	exponential	WSEPT (preemptive)	10.4
$1 \mid\mid \sum w_j (1 - e^{-rC_j})$	arbitrary	DWSEPT	10.1
$1 \mid prmp \mid \sum w_j (1 - e^{-rC_j})$	arbitrary	Gittins Index	10.2
$1 \mid\mid L_{\max}$	arbitrary	EDD	10.1
$1 \mid d_j = d \mid \sum w_j U_j$	exponential	WSEPT	10.4
$1 \mid d_j = d \mid \sum w_j T_j$	exponential	WSEPT	10.4
$Pm \mid\mid C_{\max}$	exponential	LEPT	12.1, 12.2
$Pm \mid prmp \mid C_{\max}$	exponential	LEPT	12.1, 12.2
$Pm \mid\mid \sum C_j$	exponential	SEPT	12.2
$Pm \mid prmp \mid \sum C_j$	exponential	SEPT	12.2
$P2 \mid p_j = 1, intree \mid C_{\max}$	exponential	CP	12.2
$P2 \mid p_j = 1, intree \mid \sum C_j$	exponential	CP	12.2
$F2 \mid\mid C_{\max}$	exponential	$(\lambda_j - \mu_j) \downarrow$	13.1
$Fm \mid p_{ij} = p_j \mid C_{\max}$	$\geq_{as}$	SEPT-LEPT	13.1
$Fm \mid p_{ij} = p_j \mid \sum C_j$	$\geq_{as}$	SEPT	13.1
$F2 \mid block \mid C_{\max}$	arbitrary	TSP	13.2
$F2 \mid p_{ij} = p_j, block \mid C_{\max}$	$\geq_{st}$	SEPT-LEPT	13.2
$F2 \mid p_{ij} = p_j, block \mid C_{\max}$	$\geq_{sv}$	LV-SV	13.2
$Fm \mid p_{ij} = p_j, block \mid C_{\max}$	$\geq_{as}$	SEPT-LEPT	13.2
$Fm \mid p_{ij} = p_j, block \mid \sum C_j$	$\geq_{as}$	SEPT	13.2
$J2 \mid\mid C_{\max}$	exponential	Theorem 13.3.1	13.3
$O2 \mid p_{ij} = p_j \mid C_{\max}$	exponential	Theorem 13.4.1	13.4
$O2 \mid p_{ij} = 1, prmp \mid \sum C_j$	exponential	SERPT	13.4

Table F.1 Tractable Stochastic Scheduling Problems

(iv) $Pm \mid\mid C_{\max}$.

The first problem allows for a nice solution when the processing times are exponential and the release dates are arbitrarily distributed. The optimal policy is then the preemptive WSEPT rule. When the processing time distributions are anything but exponential it appears that the preemptive WSEPT rule is not necessarily optimal. The stochastic counterparts of the second and third problem also lead to the WSEPT rule when the processing time distributions

are exponential and the jobs have a common due date which is arbitrarily distributed. Also here, if the processing times are anything but exponential the optimal rule is not necessarily WSEPT.

The stochastic counterparts of $Pm \mid\mid C_{\max}$ are slightly different. When the processing times are exponential the LEPT rule minimizes the expected makespan in all classes of policies. However, this holds for other distributions also. If the processing times are DCR (e.g., hyperexponentially distributed) and satisfy a fairly strong form of stochastic dominance, the LEPT rule remains optimal. Note that when preemptions are allowed, and the processing times are DCR, the nonpreemptive LEPT rule remains optimal. Note also, that if the n processing times have the same mean and are hyperexponentially distributed as in Example 12.1.7, then the LV rule minimizes the expected makespan.

There are many problems of which the stochastic versions exhibit very strong similarities to their deterministic counterparts. Examples of such problems are

(i) $1 \mid r_j, prmp \mid L_{\max}$,
(ii) $1 \mid prec \mid h_{\max}$,
(iii) $F2 \mid\mid C_{\max}$,
(iv) $J2 \mid\mid C_{\max}$.

It can be shown that the preemptive EDD rule is optimal for the deterministic problem $1 \mid r_j, prmp \mid L_{\max}$ and that it remains optimal when the processing times are random variables that are arbitrarily distributed. In Chapter 10 it is shown that the algorithm for the stochastic counterpart of $1 \mid prec \mid h_{\max}$ is very similar to the algorithm for the deterministic version. The same can be said with regard to $F2 \mid\mid C_{\max}$ and $J2 \mid\mid C_{\max}$ when the processing times are exponential.

Of course, there are also problems of which the deterministic version is easy and the version with exponential processing times is hard. Examples of such problems are

(i) $Pm \mid p_j = 1, tree \mid C_{\max}$,
(ii) $F2 \mid block \mid C_{\max}$,
(iii) $O2 \mid\mid C_{\max}$.

For the deterministic problem $Pm \mid p_j = 1, tree \mid C_{\max}$ the CP rule is optimal. For the version of the same problem with all processing times i.i.d. exponential the optimal policy is not known and may depend on the structure of the tree. The $F2 \mid block \mid C_{\max}$ problem with deterministic processing times is equivalent to a TSP with a special structure that allows for a polynomial time algorithm (see Section 6.2). However, when the processing time of job j on the first (second) machine is exponentially distributed with rate λ_j (μ_j), then the problem also reduces to a TSP (see Example 13.2.1); however, the structure of this TSP does *not* allow for a polynomial time solution. For the $O2 \mid\mid C_{\max}$ problem the LAPT rule is optimal; when the processing times are exponential the problem appears to be very hard.

Comments and References

For an early overview of stochastic scheduling on parallel machines, see Weiss (1982). For a discussion of tractable stochastic scheduling problems of which the deterministic counterparts are NP-hard, see Pinedo (1983). For a more recent and comprehensive overview of stochastic scheduling, see Righter (1994).

Appendix G

Selected Scheduling Systems

Over the last two decades hundreds of scheduling systems have been developed. These developments have taken place in industry and academia in various countries. An up-to-date list or annotated bibliography of all systems most likely does not exist. However, several survey papers have been written describing a number of systems.

In this appendix a distinction is made between commercial generic systems, industrial systems that are application-specific, and academic prototypes (research systems). The considerations in the design and the development of the various classes of systems are usually quite different. In this appendix systems are *not* categorized according to the approach used for generating schedules, i.e., whether it is knowledge-based or based on algorithms (since most knowledge-based systems also have algorithmic components).

Commercial generic systems are designed for implementation in a wide variety of settings with only minor customization. The software houses that develop generic systems are usually not associated with a single company. However, they may focus on a specific industry. Examples of such systems are presented in Table G.1.

SYSTEM	COMPANY	WEBSITE
Cyberplan	Cybertec	www.cybertec.it
SKEP	DynaSys Group	www.adaptasolutions.com
Production Scheduler	i2 Technologies, Inc.	www.i2.com
Quintiq Scheduler	Quintiq	www.quintiq.com
APO	SAP AG	www.sap.com
SSA Manufacturing Scheduling	SSA Global	www.ssaglobal.com

Table G.1 Commercial Generic Systems

Application-specific systems are designed for either a single installation or a single type of installation. The algorithms embedded in these systems are typically quite elaborate. A number of the application-specific systems in industry have been developed in collaboration with an academic partner. Examples of application-specific systems are listed in Table G.2.

SYSTEM	COMPANY	REFERENCE
BPSS	International Paper	Adler et al. (1993)
GATES	Trans World Airways	Brazile and Swigger (1988)
Jobplan	Siemens	Kanet and Sridharan (1990)
LMS	IBM	Sullivan and Fordyce (1990)
MacMerl	Pittsburgh Plate & Glass	Hsu et al. (1993)
SAIGA	Aeroports de Paris	Ilog (1999)

Table G.2 Application-Specific Systems

Academic prototypes are usually developed for research and teaching. The programmers are typically graduate students who work part-time on the system. The design of these systems is often completely different from the design of commercial systems. Some attempts have been made to commercialize academic prototypes. Examples of such academic systems are presented in Table G.3.

SYSTEM	INSTITUTION	REFERENCE
LEKIN	New York University	Feldman and Pinedo (1998)
OPIS	Carnegie-Mellon University	Smith (1994)
TORSCHE	Czech Technical University	Stibor and Kutil (2006)
TOSCA	University of Edinburgh	Beck (1993)
TTA	Universidad Catolica de Chile	Nussbaum and Parra (1993)

Table G.3 Academic Systems

Comments and References

Several reviews and survey papers have been written on scheduling systems, see Steffen (1986), Adelsberger and Kanet (1991), Smith (1992), Arguello (1994), and Yen and Pinedo (1994).

Appendix H

The Lekin System

This appendix provides examples of the formats of files that contain workstation information and job information. It also gives an example that illustrates how input data are read and how output data are written by a program that is linked to the LEKIN system.

H.1 Formatting of Input and Output Files

This section focuses on the formats of the input and output files.

Example H.1.1 (File Containing Workstation Information)

The following file contains the information pertaining to various workstations. The first workstation consists of two machines in parallel.

Actual line		Comments
Flexible:		Defines "flexible" environment. If the word is "single", then this is an Uni-m/c environment.
Workstation:	Wks000	Defines the first workstation. Wks000 is the name of the workstation.
Setup:	A;B;1 B;A;2	Setup matrix for this workstation. The setup time between a status A operation and a status B operation is 1, whereas if you switch the order to B,A the setup time becomes 1. All other setup times are 0.
Machine:	Wks000.000	Defines a machine with name Wks000.000

Machine:	Wks000.001	Defines another machine with name Wks000.001
Workstation:	Wks001	Defines the next workstation named Wks001
Setup:		Even when there are no setups the system requires a line for setups.
Machine:	Wks001.000	
.....		(more machines in Wks001)

||

Example H.1.2 (File Containing Information Pertaining to Jobs)
The following file contains the data pertaining to a job with multiple operations and arbitrary routes.

Actual line		Comments
Shop:	Job	Defines the job shop environment. "Flow" would indicate a flow shop, and "single" would indicate a single workstation or machine.
Job:	Job000	Defines the first job, named Job000.
Release:	0	Release date of Job000
Due:	8	Due date of Job000
Weight:	4	Weight of Job000
Oper:	Wks000;2;B	The first operation on Job000 route. It requires 2 time units at Wks000 and a machine setting B.
Oper:	Wks004;1;A	The second operation on Job000 route.
.....		(More operations of Job000)
Job:	Job001	Defines the second job, named Job001.
Release:	0	Release date of Job001
Due:	10	Due date of Job001
Weight:	2	Weight of Job001
Oper:	Wks001;3;C	The first operation on Job001 route. It requires 3 time units at Wks001 and a machine setting C.
Oper:	Wks003;1;A	The second operation on Job001 route.
.....		(More operations of Job001)

||

The following example describes the format of the output file.

Example H.1.3 (Format of Output File)

The output file has a very easy format. Only the sequences of the operations on the various machines have to be specified. It is not necessary to provide starting times and completion times in the output.

Actual line		Comments
Schedule:	SB for Cmax	Provides the name "SB for Cmax" for the first schedule in the file.
Machine:	Wks000.000	This is the proper name for the machine Wks000.000 at the workstation Wks000.
Oper:	Job01	This is the first operation scheduled on the machine above. Since recirculation is not allowed the pair (Job01, Wks000) uniquely defines the operation.
Oper:	Job06	Second operation on this machine.
.....		More operations on this machine.
Machine:	Wks000.001	This is the second machine of workstation Wks000.
Oper:	Job06	This is the first operation scheduled on machine Wks000.001
.....		More operations on this machine.
Schedule:	SB for Lmax	provides the name "SB for Lmax" for the second schedule in the file.
.....		Data with regard to the second schedule.

||

H.2 Linking Scheduling Programs

The following example contains a program written in C++ that schedules a number of jobs on a single machine. This example illustrates how input data are read and how output data are written.

Example H.2.1 (Scheduling Jobs on a Single Machine)

This example illustrates how input files and output files are used in an actual program. In this program a set of jobs are scheduled on a single machine according to the WSPT rule.

```
#include <iostream.h>
#include <fstream.h>
#include <string.h>
#include <stdio.h>
```

```
#include <stdlib.h>
// We need a data structure to store information about job.
struct Tjob
{
int id;
int release;
int due;
int weight;
int proc;
double wp; // weight divided by processing time
};
Tjob jobArray[100];
int jobCount;
// a string buffer.
char buffer[1024];
// ----------------------------------------
// For the single machine setting, we do not have to read
// the machine file. But we will do it here anyhow just to
// verify that it actually represents a single machine.
void ReadMch()
{
// No need to use the qualified path. Just "_user.mch".
ifstream Fmch("_user.mch", ios::nocreate);
// Check the first line in the file.
// If it is not "Single:", too bad.
Fmch.getline(buffer, 1024);
if (strcmp(buffer, "Single:"))
{
cout << "we do not support flexible workstations!\n";
exit(1);
}
// Now we skip several lines. There are two ways to skip:
// Getline or ignore. Getline allows you to check what you are skipping.
Fmch.getline(buffer, 1024);
// buffer = "Workstation: Wks000",
// but we do not care.
Fmch.ignore(1024, '\n');  //skip "Setup:"
Fmch.ignore(1024, '\n');  //skip "Machine:"
// We do not need the availability time or the starting status for the
// machine, but we will read it just to show how it is done.
Fmch.ignore(20);  // skip "Release:"
int avail: Fmch >> avail;
Fmch.ignore(20)  // skip "Status:"
// Counting spaces is not a good idea, so just read till the first character.
Fmch.eatwhite();
```

```
char status=Fmch.get();
// Now the rest of the file must contain only white-space characters.
Fmch.eatwhite();
if (!Fmch.eof())
{
cout << "The file must contain at least two workstations!\n";
exit(1);
}
//————————————————————————————
// With the job file it is less easy; a stream of jobs have to be read.
void readJob()
{
ifstream Fjob("_user.job", ios::nocreate);
Fjob >> buffer;  // buffer = "Shop:", ignore
Fjob >> buffer;  // check if single machine
if (strcmp(buffer, "Single"))
{
cout << "This is not a single machine file!\n";
exit(1);
}
while(1)
{
Fjob >> buffer; // buffer = "Job:"
if (strcmp(buffer, "Job:")) // if not, must be the end of the file
break;
Fjob >> buffer; // buffer = "Job###", ignore
jobarray[jobCount].id=jobCount;
Fjob >> buffer;  // buffer = "release:'
Fjob >> jobArray[jobCount].release;
Fjob >> buffer;  // buffer = "due:'
Fjob >> jobArray[jobCount].due;
Fjob >> buffer;  // buffer = "weight:'
Fjob >> jobArray[jobCount].weight;
Fjob >> buffer;  // buffer = "Oper:'
Fjob >> buffer;  // buffer = "Wks000;#;A" and we need the #
char* ss = strchr(buffer, ';');
if (!ss) break;
if (sscanf(ss+1, "%d", & jobArray[jobCount].proc) < 1) break;
jobArray[jobCount].wp=
double(jobArray[jobCount].weight/jobArray[jobCount].proc);
jobcount++;
}
if (jobCount == 0 )
{
cout << "No jobs defined!\n";
```

```
exit(1);
}
}
//————————————————————————————
// Compare function for sorting
int compare(const void* j1, const void* j2)
{
TJob* jb1= (TJob*)j1;
TJob* jb2= (TJob*)j2;
double a = jb1 − >wp - jb2− >wp;
return a<0 ? -1 : a>0 ? 1: 0;
}
// Since this is just a single machine,
// we can implement any rule by sorting on the job array.
// We use that C standard qsort function.
void SortJobs()
{ qsort(jobArray, jobCount, sizeof(TJob), compare); }
// Output the schedule file.
void WriteSeq()
{
ofstream Fsch("_user.seq");
Fsch << "Schedule: WSPT rule\n";  // schedule name
Fsch << "Machine: Wks000.000\n";  // Name of the first and last
machine
// Now enumerate the operations.
for (int i=0; i<jobCount; i++)
Fsch << "Oper: Job" << JobArray[i].id << "\n";
}
//————————————————————————————
int main (int argc. char* argv[])
{
 // We have to have exactly 2 command line segments:
// objective function and time limit.
if (argc !=3)
{
cout << "illegal call!\n";
exit(1)
}
// Check the objective function.
// The WSPT rule is for the total weighted completion time.
// Do not bother to use sscanf
if (strcmp(argv[1], "3"))
{
cout << "The only objective supported is
total weighted completion time.\n';
```

```
    exit(1);
    }
    ReadMch();
    ReadJob();
    SortJobs();
    WriteSeq();
    cout << "Success\n";
    return 0;
    }
```

||

Comments and References

The LEKIN system is due to Asadathorn (1997) and Feldman and Pinedo (1998). The general purpose routine of the shifting bottleneck type that is embedded in the system is due to Asadathorn (1997). The local search routines that are applicable to the flow shop and job shop are due to Kreipl (2000). The more specialized SB-LS routine for the flexible flow shop is due to Yang, Kreipl and Pinedo (2000).

References

E.H.L. Aarts and J.K. Lenstra (eds.) (1997) *Local Search and Combinatorial Optimization*, J. Wiley, New York.

J.O. Achugbue and F.Y. Chin (1982) "Scheduling the Open Shop to Minimize Mean Flow Time", *SIAM Journal of Computing*, Vol. 11, pp. 709–720.

J. Adams, E. Balas and D. Zawack (1988) "The Shifting Bottleneck Procedure for Job Shop Scheduling", *Management Science*, Vol. 34, pp. 391–401.

H.H. Adelsberger and J.J. Kanet (1991) "The LEITSTAND - A New Tool for Computer-Integrated-Manufacturing", *Production and Inventory Management Journal*, Vol. 32, pp. 43–48.

L. Adler, N.M. Fraiman, E. Kobacker, M. Pinedo, J.C. Plotnicoff and T.-P. Wu (1993) "BPSS: A Scheduling System for the Packaging Industry", *Operations Research*, Vol. 41, pp. 641–648.

A. Agnetis, P. B. Mirchandani, D. Pacciarelli and A. Pacifici (2004) "Scheduling Problems with Two Competing Agents", *Operations Research*, Vol. 52, pp. 229–242.

A.K. Agrawala, E.G. Coffman, Jr., M.R. Garey and S.K. Tripathi (1984) "A Stochastic Optimization Algorithm Minimizing Exponential Flow Times on Uniform Processors", *IEEE Transactions on Computers*, C-33, pp. 351–356.

C. Akkan and S. Karabati (2004) "The Two-Machine Flowshop Total Completion Time Problem: Improved Lower Bounds and a Branch-and-Bound Algorithm", *European Journal of Operational Research*, Vol. 159, pp. 420–429.

R. Akkiraju, P. Keskinocak, S. Murthy, F. Wu (1998) "A New Decision Support System for Paper Manufacturing", in *Proceedings of the Sixth International Workshop on Project Management and Scheduling (1998)*, pp. 147–150, Bogazici University Printing Office, Istanbul, Turkey.

R. Akkiraju, P. Keskinocak, S. Murthy, F. Wu (2001) "An Agent based Approach to Multi Machine Scheduling", *Journal of Applied Intelligence*, Vol. 14, pp. 135–144.

M.S. Akturk and E. Gorgulu (1999) "Match-Up Scheduling under a Machine Breakdown", *European Journal of Operational Research*, Vol. 112, pp. 81–97.

N. Alon, Y. Azar, G.J. Woeginger and T. Yadid (1998) "Approximation Schemes for Scheduling on Parallel Machines", *Journal of Scheduling*, Vol. 1, pp. 55–66.

D. Applegate and W. Cook (1991) "A Computational Study of the Job-Shop Scheduling Problem", *ORSA Journal on Computing*, Vol. 3, pp. 149–156.

M. Arguello (1994) "Review of Scheduling Software", Technology Transfer 93091822A-XFR, SEMATECH, Austin, Texas.

N. Asadathorn (1997) *Scheduling of Assembly Type of Manufacturing Systems: Algorithms and Systems Development*, Ph.D Thesis, Department of Industrial Engineering, New Jersey Institute of Technology, Newark, New Jersey.

H. Atabakhsh (1991) "A Survey of Constraint Based Scheduling Systems Using an Artificial Intelligence Approach", *Artificial Intelligence in Engineering*, Vol. 6, No. 2, pp. 58–73.

H. Aytug, S. Bhattacharyya, G.J. Koehler and J.L. Snowdon (1994) "A Review of Machine Learning in Scheduling", *IEEE Transactions on Engineering Management*, Vol. 41, pp. 165–171.

H. Aytug, M. Khouja and F.E. Vergara (2003) "Use of Genetic Algorithms to Solve Production and Operations Management Problems: a Review", *International Journal of Production Research*, Vol. 41, pp. 3955–4009.

P.C. Bagga (1970) "n-Job, 2-Machine Sequencing Problem with Stochastic Service", *Opsearch*, Vol. 7, pp. 184–197.

K.R. Baker (1974) *Introduction to Sequencing and Scheduling*, John Wiley, NY.

K.R. Baker (1975) "A Comparative Survey of Flowshop Algorithms", *Operations Research*, Vol. 23, pp. 62–73.

K.R. Baker (1995) *Elements of Sequencing and Scheduling*, K. Baker, Amos Tuck School of Business Administration, Dartmouth College, Hanover, NH 03755.

K.R. Baker and G.D. Scudder (1990) "Sequencing with Earliness and Tardiness Penalties: A Review", *Operations Research*, Vol. 38, pp. 22–36.

K.R. Baker and J. C. Smith (2003) "A Multiple-Criterion Model for Machine Scheduling". *Journal of Scheduling*, Vol. 6, pp. 7–16.

E. Balas, J.K. Lenstra and A. Vazacopoulos (1995) "The One-Machine Scheduling Problem with Delayed Precedence Constraints and its Use in Job Shop Scheduling", *Management Science*, Vol. 41, pp. 94–109.

P. Baptiste (1999) "Polynomial Time Algorithms for Minimizing the Weighted Number of Late Jobs on a Single Machine when Processing Times are Equal", *Journal of Scheduling*, Vol. 2, pp. 245–252.

P. Baptiste, C. Le Pape, and W. Nuijten (1995) "Constraint-Based Optimization and Approximation for Job-Shop Scheduling", in *Proceedings of the AAAI-SIGMAN Workshop on Intelligent Manufacturing Systems IJCAI-95*, Montreal, Canada.

P. Baptiste, C. Le Pape, and W. Nuijten (2001) *Constraint-Based Scheduling*, Kluwer Academic Publishers, Boston.

P. Baptiste, L. Peridy and E. Pinson (2003) "A Branch and Bound to Minimize the Number of Late Jobs on a Single Machine with Release Time Constraints", *European Journal of Operational Research*, Vol. 144, pp. 1–11.

J.R. Barker and G.B. McMahon (1985) "Scheduling the General Job-shop", *Management Science*, Vol. 31, pp. 594–598.

R.E. Barlow and F. Proschan (1975) *Statistical Theory of Reliability and Life Testing: Probability Models*, Holt, Rinehart and Winston, Inc., New York.

C. Barnhart, E.L. Johnson, G.L. Nemhauser, M.W.P. Savelsbergh and P.H. Vance (1998) "Branch and Price: Column Generation for Solving Huge Integer Programs", *Operations Research*, Vol. 46, pp. 316–329.

K.M. Baumgartner and B.W. Wah (1991) "Computer Scheduling Algorithms: Past, Present and Future", *Information Sciences*, Vol. 57–58, pp. 319–345.

J. Bean (1994) "Genetics and Random Keys for Sequencing and Optimization", *ORSA Journal of Computing*, Vol. 6, pp. 154–160.

J. Bean, J. Birge, J. Mittenthal and C. Noon (1991) "Matchup Scheduling with Multiple Resources, Release Dates and Disruptions", *Operations Research*, Vol. 39, pp. 470–483.

H. Beck (1993) "The Management of Job Shop Scheduling Constraints in TOSCA", in *Intelligent Dynamic Scheduling for Manufacturing Systems*, L. Interrante (ed.), Proceedings of a Workshop Sponsored by the National Science Foundation, the University of Alabama in Huntsville and Carnegie Mellon University, held at Cocoa Beach, January, 1993.

M. Bell (2000) "i2's Rhythm Architecture", Technical Report, i2 Technologies, Dallas.

D.P. Bertsekas (1987) *Dynamic Programming: Deterministic and Stochastic Models*, Prentice Hall, New Jersey.

K. Bhaskaran and M. Pinedo (1992) "Dispatching", Chapter 83 in *Handbook of Industrial Engineering*, G. Salvendy (ed.), pp. 2184–2198, John Wiley, New York.

L. Bianco, S. Ricciardelli, G. Rinaldi and A. Sassano (1988) "Scheduling Tasks with Sequence-Dependent Processing Times", *Naval Research Logistics Quarterly*, Vol. 35, pp. 177–184.

C. Bierwirth and D.C. Mattfeld (1999) "Production Scheduling and Rescheduling with Genetic Algorithms", *Evolutionary Computation*, Vol. 7, pp. 1–17.

J. Birge, J.B.G. Frenk, J. Mittenthal and A.H.G. Rinnooy Kan (1990) "Single Machine Scheduling Subject to Stochastic Breakdowns", *Naval Research Logistics Quarterly*, Vol. 37, pp. 661–677.

G.R. Bitran and D. Tirupati (1988) "Planning and Scheduling for Epitaxial Wafer Production", *Operations Research*, Vol. 36, pp. 34–49.

J. Blazewicz, W. Cellary, R. Slowinski and J. Weglarz (1986), *Scheduling under Resource Constraints - Deterministic Models*, *Annals of Operations Research*, Vol. 7, Baltzer, Basel.

J. Blazewicz, M. Dror and J. Weglarz (1991) "Mathematical Programming Formulations for Machine Scheduling: A Survey", *European Journal of Operational Research*, Vol. 51, pp. 283–300.

J. Blazewicz, K. Ecker, G. Schmidt and J. Weglarz (1993) *Scheduling in Computer and Manufacturing Systems*, Springer Verlag, Berlin.

J. Blazewicz, K. Ecker, E. Pesch, G. Schmidt and J. Weglarz (1996) *Scheduling Computer and Manufacturing Processes*, Springer Verlag, Berlin.

G. Booch (1994) *Object-Oriented Analysis and Design with Applications (Second Edition)*, Benjamin/Cummings Scientific, Menlo Park, California.

O.J. Boxma and F.G. Forst (1986) "Minimizing the Expected Weighted Number of Tardy Jobs in Stochastic Flow Shops", *Operations Research Letters*, Vol. 5, pp. 119–126.

H. Braun (2000) "Optimizing the Supply Chain - Challenges and Opportunities", *SAP Inside*, SAP AG, Walldorf, Germany.

R.P. Brazile and K.M. Swigger (1988) "GATES: An Airline Assignment and Tracking System", *IEEE Expert*, Vol. 3, pp. 33–39.

R.P. Brazile and K.M. Swigger (1991) "Generalized Heuristics for the Gate Assignment Problem", *Control and Computers*, Vol. 19, pp. 27–32.

A. Brown and Z.A. Lomnicki (1966) "Some Applications of the Branch and Bound Algorithm to the Machine Sequencing Problem", *Operational Research Quarterly*, Vol. 17, pp. 173–186.

D.E. Brown and W.T. Scherer (eds.) (1995) *Intelligent Scheduling Systems*, Kluwer Academic Publishers, Boston.

M. Brown and H. Solomon (1973) "Optimal Issuing Policies under Stochastic Field Lives", *Journal of Applied Probability*, Vol. 10, pp. 761–768.

S. Browne and U. Yechiali (1990) "Scheduling Deteriorating Jobs on a Single Processor", *Operations Research*, Vol. 38, pp. 495–498.

P. Brucker (1995) *Scheduling Algorithms (First Edition)*, Springer Verlag, Berlin.

P. Brucker (2004) *Scheduling Algorithms (Fourth Edition)*, Springer Verlag, Berlin.

P. Brucker, A. Gladky, H. Hoogeveen, M.Y. Kovalyov, C.N. Potts, T. Tautenhahn and S.L. van de Velde (1998) "Scheduling a Batching Machine", *Journal of Scheduling*, Vol. 1, pp. 31–54.

P. Brucker, B. Jurisch, and M. Jurisch (1993) "Open Shop Problems with Unit Time Operations", *Zeitschrift für Operations Research*, Vol. 37, pp. 59–73.

P. Brucker, B. Jurisch, and A. Krämer (1994) "The Job Shop Problem and Immediate Selection", *Annals of Operations Research*, Vol. 50, pp. 73–114.

P. Brucker, B. Jurisch, and B. Sievers (1994) "A Branch and Bound Algorithm for the Job Shop Problem", *Discrete Applied Mathematics*, Vol. 49, pp. 107–127.

J. Bruno, P. Downey and G. Frederickson (1981) "Sequencing Tasks with Exponential Service Times to Minimize the Expected Flow Time or Makespan", *Journal of the Association of Computing Machinery*, Vol. 28, pp. 100–113.

J. Bruno and T. Gonzalez (1976) "Scheduling Independent Tasks with Release Dates and Due Dates on Parallel Machines", *Technical Report 213*, Computer Science Department, Pennsylvania State University.

L. Burns and C.F. Daganzo (1987) "Assembly Line Job Sequencing Principles", *International Journal of Production Research*, Vol. 25, pp. 71–99.

G. Buxey (1989) "Production Scheduling: Practice and Theory", *European Journal of Operational Research*, Vol. 39, pp. 17–31.

C. Buyukkoc, P. Varaiya and J. Walrand (1985) "The $c\mu$ Rule Revisited", *Advances in Applied Probability*, Vol. 17, pp. 237–238.

H.G. Campbell, R.A. Dudek and M.L. Smith (1970) "A Heuristic Algorithm for the n Job m Machine Sequencing Problem", *Management Science*, Vol. 16, pp. B630–B637.

J. Carlier (1982) "The One-Machine Sequencing Problem", *European Journal of Operational Research*, Vol. 11, pp. 42–47.

J. Carlier and E. Pinson (1989) "An Algorithm for Solving the Job Shop Problem", *Management Science*, Vol. 35, pp. 164–176.

D.C. Carroll (1965) *Heuristic Sequencing of Single and Multiple Component Jobs*, Ph.D Thesis, Sloan School of Management, M.I.T., Cambridge, MA.

S. Chand, R. Traub and R. Uzsoy (1996) "Single Machine Scheduling with Dynamic Arrivals: Decomposition Results and an Improved Algorithm", *Naval Research Logistics*, Vol. 31, pp. 709–719.

S. Chand, R. Traub and R. Uzsoy (1997) "Rolling Horizon Procedures for the Single Machine Deterministic Total Completion Time Scheduling Problem with Release Dates", *Annals of Operations Research*, Vol. 70, C.-Y. Lee and L. Lei (eds.), pp. 115–125.

K.M. Chandy and P.F. Reynolds (1975) "Scheduling Partially Ordered Tasks with Probabilistic Execution Times", in *Proceedings of the fifth Symposium on Operating Systems Principles*, *Operating Systems Review*, Vol. 9, pp. 169–177.

C.-S. Chang, R. Nelson and M. Pinedo (1992) "Scheduling Two Classes of Exponential Jobs on Parallel Processors: Structural Results and Worst Case Analysis", *Advances in Applied Probability*, Vol. 23, pp. 925–944.

C.-S. Chang, X.L. Chao, M. Pinedo and R.R. Weber (1992) "On the Optimality of *LEPT* and $c\mu$ Rules for Machines in Parallel", *Journal of Applied Probability*, Vol. 29, pp. 667–681.

C.-S. Chang and D.D. Yao (1993) "Rearrangement, Majorization and Stochastic Scheduling", *Mathematics of Operations Research*, Vol. 18, pp. 658–684.

X. Chao and M. Pinedo (1992) "A Parametric Adjustment Method for Dispatching", Technical Report, Department of Industrial Engineering and Operations Research, Columbia University, New York.

C. Chekuri, R. Motwani, B. Natarajan, and C. Stein (1997) "Approximation Techniques for Average Completion Time Scheduling", in *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 609–617.

B. Chen, C.N. Potts and G.J. Woeginger (1998) "A Review of Machine Scheduling: Complexity, Algorithms, and Approximability", in *Handbook of Combinatorial Optimization*, D.-Z. Du and P. Pardalos (eds.), pp. 21–169, Kluwer Academic Press, Boston.

C.-L. Chen and R.L. Bulfin (1993) "Complexity of Single Machine, Multi-Criteria Scheduling Problems", *European Journal of Operational Research*, Vol. 70, pp. 115–125.

C.-L. Chen and R.L. Bulfin (1994) "Scheduling a Single Machine to Minimize Two Criteria: Maximum Tardiness and Number of Tardy Jobs", *IIE Transactions*, Vol. 26, pp. 76–84.

N.-F. Chen and C.L. Liu (1975) "On a Class of Scheduling Algorithms for Multiprocessors Computing Systems", in *Parallel Processing*, T.-Y. Feng (ed.), Lecture Notes in Computer Science, No. 24, pp. 1–16, Springer Verlag , Berlin.

Y.R. Chen and M.N. Katehakis (1986) "Linear Programming for Finite State Multi-Armed Bandit Problems", *Mathematics of Operations Research*, Vol. 11, pp. 180–183.

Z.-L. Chen and W.B. Powell (1999) "Solving Parallel Machine Scheduling Problems by Column Generation", *INFORMS Journal of Computing*, Vol. 11, pp. 78–94.

Z.-L. Chen and G. Pundoor (2006) "Order Assignment and Scheduling in a Supply Chain", *Operations Resarch*, Vol. 54, pp. 555–572.

Z.-L. Chen and G. Vairaktarakis (2005) "Integrated Scheduling of Production and Distribution Operations", *Management Science*, Vol. 51, pp. 614–628.

C.-C. Cheng and S.F. Smith (1997) "Applying Constraint Satisfaction Techniques to Job Shop Scheduling", *Annals of Operations Research*, Vol. 70, C.-Y. Lee and L. Lei (eds.), pp. 327–357.

T.C.E. Cheng and M.C. Gupta (1989) "Survey of Scheduling Research Involving Due Date Determination Decisions", *European Journal of Operational Research*, Vol. 38, pp. 156–166.

T.C.E. Cheng, C.T. Ng and J.J. Yuan (2006) "Multi-Agent Scheduling on a Single Machine to Minimize Total Weighted Number of Tardy Jobs", *Theoretical Computer Science*, Vol. 362, pp. 273–281.

M. Chimani, N. Lesh, M. Mitzenmacher, C. Sidner, and H. Tanaka (2005) "A Case Study in Large-Scale Interactive Optimization", in *Proceedings of the International Conference on Artificial Intelligence and Applications 2005 (AIA05)*, IASTED-Proc, pp. 24–29, Acta Press, Anaheim, CA.

Y. Cho and S. Sahni (1981) "Preemptive Scheduling of Independent Jobs with Release and Due Times on Open, Flow and Job Shops", *Operations Research*, Vol. 29, pp. 511–522.

P. Chrétienne, E.G. Coffman, Jr., J.K. Lenstra, and Z. Liu (eds.) (1995) *Scheduling Theory and Applications*, John Wiley, New York.

A. Cobham (1954) "Priority Assignment in Waiting Line Problems", *Operations Research*, Vol. 2, pp. 70–76.

E.G. Coffman, Jr. (ed.) (1976) *Computer and Job Shop Scheduling Theory*, John Wiley, New York.

E.G. Coffman, Jr., L. Flatto, M.R. Garey and R.R. Weber (1987) "Minimizing Expected Makespans on Uniform Processor Systems", *Advances in Applied Probability*, Vol. 19, pp. 177–201.

E.G. Coffman, Jr., M.R. Garey and D.S. Johnson (1978) "An Application of Bin-Packing to Multiprocessor Scheduling", *SIAM Journal of Computing*, Vol. 7, pp. 1–17.

A. Collinot, C. LePape and G. Pinoteau (1988) "SONIA: A Knowledge-Based Scheduling System", *Artificial Intelligence in Engineering*, Vol. 2, pp. 86–94.

R.K. Congram, C.N. Potts, and S.L. van de Velde (2002) "An Iterated Dynasearch Algorithm for the Single-Machine Total Weighted Tardiness Scheduling Problem", *INFORMS Journal on Computing*, Vol. 14, pp. 52–67.

R.W. Conway (1965a) "Priority Dispatching and Work-In-Process Inventory in a Job Shop", *Journal of Industrial Engineering*, Vol. 16, pp. 123–130.

R.W. Conway (1965b) "Priority Dispatching and Job Lateness in a Job Shop", *Journal of Industrial Engineering*, Vol. 16, pp. 228–237.

R.W. Conway, W.L. Maxwell, L.W. Miller (1967) *Theory of Scheduling*, Addison-Wesley, Reading, MA.

D.R. Cox and W.L. Smith (1961) *Queues*, John Wiley, New York.

T.B. Crabill and W.L. Maxwell (1969) "Single Machine Sequencing with Random Processing Times and Random Due Dates", *Naval Research Logistics Quarterly*, Vol. 16, pp. 549–554.

A.A. Cunningham and S.K. Dutta (1973) "Scheduling Jobs with Exponentially Distributed Processing Times on Two Machines of a Flow Shop", *Naval Research Logistics Quarterly*, Vol. 20, pp. 69–81.

I. Curiel, H. Hamers and F. Klijn (2002) "Sequencing Games: A Survey", in *Chapters in Game Theory (Theory and Decision Library, 31)*, P.E.M. Borm and H.J.M. Peeters (eds.), Kluwer Academic Publishers, Dordrecht, the Netherlands.

D.G. Dannenbring (1977) "An Evaluation of Flowshop Sequencing Heuristics", *Management Science*, Vol. 23, pp. 1174–1182.

S. Dauzère-Pérès and J.-B. Lasserre (1993) "A Modified Shifting Bottleneck Procedure for Job Shop Scheduling", *International Journal of Production Research*, Vol. 31, pp. 923–932.

S. Dauzère-Pérès and J.-B. Lasserre (1994) *An Integrated Approach in Production Planning and Scheduling*, Lecture Notes in Economics and Mathematical Systems, No. 411, Springer Verlag, Berlin.

S. Dauzère-Pérès and M. Sevaux (1998) "Various Mathematical Programming Formulations for a General One Machine Sequencing Problem", *JNPC'98*, $4^{ième}$ *Journées Nationales pour la Résolution Pratique de Problèmes NP-Complets*, Ecole des Mines, Nantes, France, pp. 63-68, Research Report 98/3/AUTO.

S. Dauzère-Pérès and M. Sevaux (2002) "Using Lagrangean Relaxation to Minimize the Weighted Number of Late Jobs on a Single Machine", *Naval Research Logistics*, Vol. 50, pp. 273–288.

E. Davis and J.M. Jaffe (1981) "Algorithms for Scheduling Tasks on Unrelated Processors", *Journal of the Association of Computing Machinery*, Vol. 28, pp. 721–736.

M.W. Dawande, H.N. Geismar, S.P. Sethi, and C. Sriskandarajah (2007) *Throughput Optimization in Robotic Cells*, International Series in Operations Research and Management Science, Springer.

K. Deb (2001) *Multi-objective Optimization Using Evolutionary Algorithms*, John Wiley, Chichester, England.

M. Dell'Amico and M. Trubian (1991) "Applying Tabu-Search to the Job Shop Scheduling Problem", *Annals of Operations Research*, Vol. 41, pp. 231–252.

F. Della Croce, M. Ghirardi, and R. Tadei (2002) "An Improved Branch-and-Bound Algorithm for the Two Machine Total Completion Time Flow Shop Problem", *European Journal of Operational Research*, Vol. 139, pp. 293–301.

F. Della Croce, V. Narayan, and R. Tadei (1996) "The Two-Machine Total Completion Time Flow Shop Problem", *European Journal of Operational Research*, Vol. 90, pp. 227–237.

F. Della Croce, R. Tadei and G. Volta (1992) "A Genetic Algorithm for the Job Shop Problem", *Computers and Operations Research*, Vol. 22, pp. 15–24.

M.A.H. Dempster, J.K. Lenstra and A.H.G. Rinnooy Kan (eds.) (1982) *Deterministic and Stochastic Scheduling*, Reidel, Dordrecht.

E.V. Denardo (1982) *Dynamic Programming: Models and Applications*, Prentice-Hall, New Jersey.

M.L. den Besten, T. Stützle and M. Dorigo (2000) "Ant Colony Optimization for the Total Weighted Tardiness Problem", in *Proceedings of PPSN-VI, Sixth International Conference on Parallel Problem Solving from Nature*, M. Schoenauer, K. Deb, G. Rudolph, X. Yao, E. Lutton, J.J. Merelo and H.-P. Schwefel (eds.), Lecture Notes in Computer Science, No. 1917, pp. 611–620, Springer Verlag, Berlin.

H. Deng, L. Chen, C. Wang and Q. Deng (2006) "A Grid-Based Scheduling System of Manufacturing Resources for a Virtual Enterprise", *International Journal of Advanced Manufacturing Technology*, Vol. 28, pp. 137–141.

C. Derman, G. Lieberman and S.M. Ross (1978) "A Renewal Decision Problem", *Management Science*, Vol. 24, pp. 554–561.

M. Derthick and S.F. Smith (2007) "An Interactive 3D Visualization for Requirements Analysis", *Journal of Scheduling*, to appear.

G. Dobson (1984) "Scheduling Independent Tasks on Uniform Processors", *SIAM Journal of Computing*, Vol. 13, pp. 705–716.

M. Dorigo and T. Stützle (2004) *Ant Colony Optimization*, MIT Press, Cambridge, MA, USA.

D.-Z. Du and P. Pardalos (eds.) (1998) *Handbook of Combinatorial Optimization*, Kluwer Academic Press, Boston.

J. Du and J.Y.-T. Leung (1989) "Scheduling Tree-Structured Tasks on Two Processors to Minimize Schedule Length", *SIAM Journal of Discrete Mathematics*, Vol. 2, pp. 176–196.

J. Du and J.Y.-T. Leung (1990) "Minimizing Total Tardiness on One Machine is NP-Hard", *Mathematics of Operations Research*, Vol. 15, pp. 483–495.

J. Du and J.Y.-T. Leung (1993a) "Minimizing Mean Flow Time in Two-Machine Open Shops and Flow Shops", *Journal of Algorithms*, Vol. 14, pp. 24–44.

J. Du and J.Y.-T. Leung (1993b) "Minimizing Mean Flow Time with Release Times and Deadline Constraints", *Journal of Algorithms*, Vol. 14, pp. 45–68.

J. Du, J.Y.-T. Leung and C.S. Wong (1992) "Minimizing the Number of Late Jobs with Release Time Constraint", *Journal of Combinatorial Mathematics and Combinatorial Computing*, Vol. 11, pp. 97–107.

J. Du, J.Y.-T. Leung and G.H. Young (1990) "Minimizing Mean Flow Time with Release Time Constraint", *Theoretical Computer Science*, Vol. 75, pp. 347–355.

J. Du, J.Y.-T. Leung and G.H. Young (1991) "Scheduling Chain-Structured Tasks to Minimize Makespan and Mean Flow Time", *Information and Computation*, Vol. 92, pp. 219–236.

B. Eck and M. Pinedo (1988) "On the Minimization of the Flow Time in Flexible Flow Shops", Technical Report, Department of Industrial Engineering and Operations Research, Columbia University, New York.

B. Eck and M. Pinedo (1993) "On the Minimization of the Makespan Subject to Flow Time Optimality", *Operations Research*, Vol. 41, pp. 797–800.

A. Elkamel and A. Mohindra (1999) "A Rolling Horizon Heuristic for Reactive Scheduling of Batch Process Operations", *Engineering Optimization*, Vol. 31, pp. 763–792.

S.E. Elmaghraby and S.H. Park (1974) "Scheduling Jobs on a Number of Identical Machines", *AIIE Transactions*, Vol. 6, pp. 1–12.

H. Emmons (1969) "One-Machine Sequencing to Minimize Certain Functions of Job Tardiness", *Operations Research*, Vol. 17, pp. 701–715.

H. Emmons (1975) "A Note on a Scheduling Problem with Dual Criteria", *Naval Research Logistics Quarterly*, Vol. 22, pp. 615–616.

H. Emmons and M. Pinedo (1990) "Scheduling Stochastic Jobs with Due Dates on Parallel Machines", *European Journal of Operational Research*, Vol. 47, pp. 49–55.

A. Federgruen and H. Groenevelt (1986) "Preemptive Scheduling of Uniform Machines by Ordinary Network Flow Techniques", *Management Science*, Vol. 32, pp. 341–349.

A. Feldman (1999) *Scheduling Algorithms and Systems*, PhD Thesis, Department of Industrial Engineering and Operations Research, Columbia University, New York.

A. Feldman and M. Pinedo (1998) "The Design and Implementation of an Educational Scheduling System", Technical Report, Department of Operations Management, Stern School of Business, New York University, New York.

M.L. Fisher (1976) "A Dual Algorithm for the One-Machine Scheduling Problem", *Mathematical Programming*, Vol. 11, pp. 229–251.

M.L. Fisher (1981) "The Lagrangean Relaxation Method for Solving Integer Programming Problems", *Management Science*, Vol. 27, pp. 1–18.

R. Fleischer and M. Wahl (2000) "Online Scheduling Revisited", *Journal of Scheduling*, Vol. 3, pp. 343–355.

R.D. Foley and S. Suresh (1984a) "Minimizing the Expected Flow Time in Stochastic Flow Shops", *IIE Transactions*, Vol. 16, pp. 391–395.

R.D. Foley and S. Suresh (1984b) "Stochastically Minimizing the Makespan in Flow Shops", *Naval Research Logistics Quarterly*, Vol. 31, pp. 551–557.

R.D. Foley and S. Suresh (1986) "Scheduling n Nonoverlapping Jobs and Two Stochastic Jobs in a Flow Shop", *Naval Research Logistics Quarterly*, Vol. 33, pp. 123–128.

F.G. Forst (1984) "A Review of the Static Stochastic Job Sequencing Literature", *Opsearch*, Vol. 21, pp. 127–144.

M.S. Fox (1987) *Constraint Directed Search: A Case Study of Job-Shop Scheduling*, Morgan Kaufmann Publishers, San Francisco, California.

M.S. Fox and S.F. Smith (1984) "ISIS – A Knowledge-Based System for Factory Scheduling", *Expert Systems*, Vol. 1, pp. 25–49.

N.M. Fraiman, M. Pinedo and P.-C. Yen (1993) "On the Architecture of a Prototype Scheduling System", in *Proceedings of the 1993 NSF Design and Manufacturing Systems Conference*, Vol. 1, pp. 835–838, Society of Manufacturing Engineers, Dearborn, MI.

S. French (1982) *Sequencing and Scheduling: an Introduction to the Mathematics of the Job Shop*, Horwood, Chichester, England.

J.B.G. Frenk (1991) "A General Framework for Stochastic One Machine Scheduling Problems with Zero Release Times and no Partial Ordering", *Probability in the Engineering and Informational Sciences*, Vol. 5, pp. 297–315.

D.K. Friesen (1984a) "Tighter Bounds for the Multifit Processor Scheduling Algorithm", *SIAM Journal of Computing*, Vol. 13, pp. 170–181.

D.K. Friesen (1984b) "Tighter Bounds for LPT Scheduling on Uniform Processors", *SIAM Journal of Computing*, Vol. 16, pp. 554–560.

D.K. Friesen and M.A. Langston (1983) "Bounds for Multifit Scheduling on Uniform Processors", *SIAM Journal of Computing*, Vol. 12, pp. 60–70.

E. Frostig (1988) "A Stochastic Scheduling Problem with Intree Precedence Constraints", *Operations Research*, Vol. 36, pp. 937–943.

E. Frostig and I. Adiri (1985) "Three-Machine Flow Shop Stochastic Scheduling to Minimize Distribution of Schedule Length", *Naval Research Logistics Quarterly*, Vol. 32, pp. 179–183.

G. Galambos and G.J. Woeginger (1995) "Minimizing the Weighted Number of Late Jobs in UET Open Shops", *ZOR-Mathematical Methods of Operations Research*, Vol. 41, pp. 109–114.

J. Gaylord (1987) *Factory Information Systems*, Marcel Dekker, NY.

M.R. Garey and D.S. Johnson (1979) *Computers and Intractability - A Guide to the Theory of NP-Completeness*, W.H. Freeman and Company, San Francisco.

M.R. Garey, D.S. Johnson and R. Sethi (1976) "The Complexity of Flowshop and Jobshop Scheduling", *Mathematics of Operations Research*, Vol. 1, pp. 117–129.

M.R. Garey, D.S. Johnson, B.B. Simons and R.E. Tarjan (1981) "Scheduling Unit-Time Tasks with Arbitrary Release Times and Deadlines", *SIAM Journal of Computing*, Vol. 10, pp. 256–269.

L. Gelders and P.R. Kleindorfer (1974) "Coordinating Aggregate and Detailed Scheduling Decisions in the One-Machine Job Shop: Part I. Theory", *Operations Research*, Vol. 22, pp. 46–60.

L. Gelders and P.R. Kleindorfer (1975) "Coordinating Aggregate and Detailed Scheduling Decisions in the One-Machine Job Shop: Part II. Computation and Structure", *Operations Research*, Vol. 23, pp. 312–324.

G.V. Gens and E.V. Levner (1981) "Fast Approximation Algorithm for Job sequencing with Deadlines", *Discrete Applied Mathematics*, Vol. 3, pp. 313–318.

B. Giffler and G.L. Thompson (1960) "Algorithms for Solving Production Scheduling Problems", *Operations Research*, Vol. 8, pp. 487–503.

P.C. Gilmore and R.E. Gomory (1964) "Sequencing a One-State Variable Machine: a Solvable Case of the Travelling Salesman Problem", *Operations Research*, Vol. 12, pp. 655–679.

J.C. Gittins (1979) "Bandit Processes and Dynamic Allocation Indices", *Journal of the Royal Statistical Society Series B*, Vol. 14, pp. 148–177.

J.C. Gittins (1981) "Multiserver Scheduling of Jobs with Increasing Completion Rates", *Journal of Applied Probability*, Vol. 18, pp. 321–324.

K.D. Glazebrook (1981a) "On Nonpreemptive Strategies for Stochastic Scheduling Problems in Continuous Time", *International Journal of System Sciences*, Vol. 12, pp. 771–782.

K.D. Glazebrook (1981b) "On Nonpreemptive Strategies in Stochastic Scheduling", *Naval Research Logistics Quarterly*, Vol. 28, pp. 289–300.

K.D. Glazebrook (1982) "On the Evaluation of of Fixed Permutations as Strategies in Stochastic Scheduling", *Stochastic Processes and Applications*, Vol. 13, pp. 171–187.

K.D. Glazebrook (1984) "Scheduling Stochastic jobs on a Single Machine Subject to Breakdowns", *Naval Research Logistics Quarterly*, Vol. 31, pp. 251–264.

K.D. Glazebrook (1987) "Evaluating the Effects of Machine Breakdwons in Stochastic Scheduling Problems", *Naval Research Logistics*, Vol. 34, pp. 319–335.

F. Glover (1990) "Tabu Search: A Tutorial", *Interfaces*, Vol. 20, Issue 4, pp. 74–94.

T. Gonzalez (1979) "A Note on Open Shop Preemptive Schedules", *IEEE Transactions on Computers*, Vol. C-28, pp. 782–786.

T. Gonzalez, J.Y.-T. Leung and M. Pinedo (2006) "Minimizing Total Completion Time on Uniform Machines with Deadline Constraints", *ACM Transactions on Algorithms*, Vol. 2, pp. 95–115.

T. Gonzalez and S. Sahni (1976) "Open Shop Scheduling to Minimize Finish Time", *Journal of the Association of Computing Machinery*, Vol. 23, pp. 665–679.

T. Gonzalez and S. Sahni (1978a) "Preemptive Scheduling of Uniform Processor Systems", *Journal of the Association of Computing Machinery*, Vol. 25, pp. 92–101.

T. Gonzalez and S. Sahni (1978b) "Flowshop and Jobshop Schedules: Complexity and Approximation", *Operations Research*, Vol. 26, pp. 36–52.

S.K. Goyal and C. Sriskandarajah (1988) "No-Wait Shop Scheduling: Computational Complexity and Approximate Algorithms", *Opsearch*, Vol. 25, pp. 220–244.

R.L. Graham (1966) "Bounds for Certain Multiprocessing Anomalies", *Bell System Technical Journal*, Vol. 45, pp. 1563–1581.

R.L. Graham (1969) "Bounds on Multiprocessing Timing Anomalies", *SIAM Journal of Applied Mathematics*, Vol. 17, pp. 263–269.

S.C. Graves (1981) "A Review of Production Scheduling", *Operations Research*, Vol.29, pp. 646–676.

S.C. Graves, H. C. Meal, D. Stefek and A. H. Zeghmi (1983) "Scheduling of Reentrant Flow Shops", *Journal of Operations Management*, Vol. 3, pp. 197–207.

M. Grönkvist (2004) "A Constraint Programming Model for Tail Assignment", in *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems - Proceedings of CPAIOR 2004*, J.-C. Régin and M. Rueher (eds.), Lecture Notes in Computer Science, No. 3011, pp. 142–156, Springer.

J.N.D. Gupta (1972) "Heuristic Algorithms for the Multistage Flow Shop Problem", *AIIE Transactions*, Vol. 4, pp. 11–18.

N.G. Hall, W. Kubiak and S.P. Sethi (1991) "Earliness-Tardiness Scheduling Problems, II: Weighted Deviation of Completion Times about a Common Due Date", *Operations Research*, Vol. 39, pp. 847–856.

N.G. Hall and M.E. Posner (1991) "Earliness-Tardiness Scheduling Problems, I: Weighted Deviation of Completion Times about a Common Due Date", *Operations Research*, Vol. 39, pp. 836–846.

N.G. Hall and C.N. Potts (2003) "Supply Chain Scheduling: Batching and Delivery", *Operations Research*, Vol. 51, pp. 566–584.

N.G. Hall and C. N. Potts (2004) "Rescheduling for New Orders", *Operations Research*, Vol. 52, pp. 440–453.

N.G. Hall and C. Sriskandarajah (1996) "A Survey of Machine Scheduling Problems with Blocking and No-Wait in Process", *Operations Research*, Vol. 44, pp. 421–439.

J.M. Harrison (1975a) "A Priority Queue with Discounted Linear Costs", *Operations Research*, Vol. 23, pp. 260–269.

J.M. Harrison (1975b) "Dynamic Scheduling of a Multiclass Queue: Discount Optimality", *Operations Research*, Vol. 23, pp. 270–282.

J. Herrmann, C.-Y. Lee and J. Snowdon (1993) "A Classification of Static Scheduling Problems", in *Complexity in Numerical Optimization*, P.M. Pardalos (ed.), World Scientific, pp. 203–253.

D.P. Heyman and M.J. Sobel (1982) *Stochastic Models in Operations Research, Volume I (Stochastic Processes and Operating Characteristics)*, McGraw-Hill, New York.

D.S. Hochbaum and D.B. Shmoys (1987) "Using Dual Approximation Algorithms for Scheduling Problems: Theoretical and Practical Results", *Journal of the ACM*, Vol. 34, pp. 144–162.

T.J. Hodgson (1977) "A Note on Single Machine Sequencing with Random Processing Times", *Management Science*, Vol. 23, pp. 1144–1146.

T.J. Hodgson and G.W. McDonald (1981a) "Interactive Scheduling of a Generalized Flow Shop. Part I: Success Through Evolutionary Development", *Interfaces*, Vol. 11, No. 2, pp. 42–47.

T.J. Hodgson and G.W. McDonald (1981b) "Interactive Scheduling of a Generalized Flow Shop. Part II: Development of Computer Programs and Files", *Interfaces*, Vol. 11, No. 3, pp. 83–88.

T.J. Hodgson and G.W. McDonald (1981c) "Interactive Scheduling of a Generalized Flow Shop. Part III: Quantifying User Objectives to Create Better Schedules", *Interfaces*, Vol. 11, No. 4, pp. 35–41.

D.J. Hoitomt, P.B. Luh and K.R. Pattipati (1993) "A Practical Approach to Job Shop Scheduling Problems", *IEEE Transactions on Robotics and Automation*, Vol. 9, pp. 1–13.

J.A. Hoogeveen and S.L. Van de Velde (1995) "Minimizing Total Completion Time and Maximum Cost Simultaneously is Solvable in Polynomial Time", *Operations Research Letters*, Vol. 17, pp. 205–208.

J.A. Hoogeveen, L. van Norden, and S.L. van de Velde (2006) "Lower Bounds for Minimizing Total Completion Time in a Two-Machine Flow Shop", *Journal of Scheduling*, Vol. 9, pp. 559–568.

J.N. Hooker (2000) *Logic-Based Methods for Optimization: Combining Optimization and Constraint Satisfaction*, John Wiley, New York.

H. Hoos and T. Stützle (2005) *Stochastic Local Search - Foundations and Applications*, Morgan Kaufmann Publishers, Elsevier, Amsterdam.

E. Horowitz and S. Sahni (1976) "Exact and Approximate Algorithms for Scheduling Nonidentical Processors", *Journal of the Association of Computing Machinery*, Vol. 23, pp. 317–327.

E.C. Horvath, S. Lam and R. Sethi (1977) "A Level Algorithm for Preemptive Scheduling", *Journal of the Association of Computing Machinery*, Vol. 24, pp. 32–43.

W.-L. Hsu, M. Prietula, G. Thompson and P.S. Ow (1993) "A Mixed-Initiative Scheduling Workbench: Integrating AI, OR and HCI", *Decision Support Systems*, Vol. 9, pp. 245–257.

T.C. Hu (1961) "Parallel Sequencing and Assembly Line Problems", *Operations Research*, Vol. 9, pp. 841–848.

H.-C. Hwang, K. Lee, and S.Y. Chang (2005) "The Effect of Machine Availability on the Worst-Case Performance of LPT", *Discrete Applied Mathematics*, Vol. 148, pp. 49–61.

O.H. Ibarra and C.E. Kim (1978) "Approximation Algorithms for Certain Scheduling Problems", *Mathematics of Operations Research*, Vol. 3, pp. 179–204.

Ilog (1997) "Reducing Congestion at Paris Airports - SAIGA", Ilog ADP Success Story, Ilog, Paris, France.

L. Interrante (ed.) (1993) *Intelligent Dynamic Scheduling for Manufacturing Systems*, Proceedings of a Workshop Sponsored by National Science Foundation, the University of Alabama in Huntsville and Carnegie Mellon University, held in Cocoa Beach, January, 1993.

J.R. Jackson (1955) "Scheduling a Production Line to Minimize Maximum Tardiness", Research Report 43, Management Science Research Project, University of California, Los Angeles.

J.R. Jackson (1956) "An Extension of Johnson's Results on Job Lot Scheduling", *Naval Research Logistics Quarterly*, Vol. 3, pp. 201–203.

V. Jain and I.E. Grossmann (2001) "Algorithms for Hybrid MILP/CP Models for a Class of Optimization Problems", *Informs Journal on Computing*, Vol. 13, pp. 258–276.

S.M. Johnson (1954) "Optimal Two and Three-Stage Production Schedules with Setup Times Included", *Naval Research Logistics Quarterly*, Vol. 1, pp. 61–67.

P.J. Kalczynski and J. Kamburowski (2005) "Two-Machine Stochastic Flow Shops with Blocking and the Travelling Salesman Problem", *Journal of Scheduling*, Vol. 8, pp. 529–536.

T. Kämpke (1987a) "On the Optimality of Static Priority Policies in Stochastic Scheduling on Parallel Machines", *Journal of Applied Probability*, Vol. 24, pp. 430–448.

T. Kämpke (1987b) "Necessary Optimality Conditions for Priority Policies in Stochastic Weighted Flow Time Scheduling Problems", *Advances in Applied Probability*, Vol. 19, pp. 749–750.

T. Kämpke (1989) "Optimal Scheduling of Jobs with Exponential Service Times on Identical Parallel Processors", *Operations Research*, Vol. 37, pp. 126–133.

J.J. Kanet and H.H. Adelsberger (1987) "Expert Systems in Production Scheduling", *European Journal of Operational Research*, Vol. 29, pp. 51–59.

J.J. Kanet and V. Sridharan (1990) "The Electronic Leitstand: A New Tool for Shop Scheduling", *Manufacturing Review*, Vol. 3, pp. 161–170.

R.M. Karp (1972) "Reducibility among Combinatorial Problems", in *Complexity of Computer Computations*, R.E. Miller and J.W. Thatcher (eds.), Plenum Press, New York.

K.R. Karwan and J.R. Sweigart (eds.) (1989) *Proceedings of the Third International Conference on Expert Systems and the Leading Edge in Production and Operations Management*, Conference held on Hilton Head Island, South Carolina, 1989, sponsored by Management Science Department, College of Business Administration, University of South Carolina.

M.N. Katehakis and A.F. Veinott Jr. (1987) "The Multi-Armed Bandit Problem: Decomposition and Computation", *Mathematics of Operations Research*, Vol. 12, pp. 262–268.

T. Kawaguchi and S. Kyan (1986) "Worst Case Bound of an LRF Schedule for the Mean Weighted Flow Time Problem", *SIAM Journal of Computing*, Vol. 15, pp. 1119–1129.

K.G. Kempf (1989) "Intelligent Interfaces for Computer Integrated Manufacturing", in *Proceedings of the Third International Conference on Expert Systems and the Leading Edge in Production and Operations Management*, K.R. Karwan and J.R. Sweigert (eds.), pp. 269–280, Management Science Department, College of Business Administration, University of South Carolina.

K.G. Kempf (1994) "Intelligently Scheduling Semiconductor Wafer Fabrication", in *Intelligent Scheduling*, M. Zweben and M. Fox (eds.), Morgan and Kaufmann, San Francisco, California.

S. Kerpedjiev and S.F. Roth (2000) "Mapping Communicative Goals into Conceptual Tasks to Generate Graphics in Discourse", in *Proceedings of Intelligent User Interfaces (IUI '00)*, January 2000.

M. Kijima, N. Makimoto and H. Shirakawa (1990) "Stochastic Minimization of the Makespan in Flow Shops with Identical Machines and Buffers of Arbitrary Size", *Operations Research*, Vol. 38, pp. 924–928.

Y.-D. Kim and C.A. Yano (1994) "Minimizing Mean Tardiness and Earliness in Single-Machine Scheduling Problems with Unequal Due Dates", *Naval Research Logistics*, Vol. 41, pp. 913–933.

S. Kirkpatrick, C.D. Gelatt and M. P. Vecchi (1983) "Optimization by Simulated Annealing", *Science*, Vol. 220, pp. 671–680.

H. Kise, T. Ibaraki and H. Mine (1978) "A Solvable Case of the One-Machine Scheduling Problem with Ready and Due Times", *Operations Research*, Vol. 26, pp. 121–126.

L. Kleinrock (1976) *Queueing Systems, Vol. II: Computer Applications*, John Wiley, New York.

G. Koole and R. Righter (2001) "A Stochastic Batching and Scheduling Problem", *Probability in the Engineering and Informational Sciences*, Vol. 15, pp. 65–79.

S.A. Kravchenko (2000) "On the Complexity of Minimizing the Number of Late Jobs in Unit Time Open Shops", *Discrete Applied Mathematics*, Vol. 100, pp. 127–132.

S. Kreipl (2000) "A Large Step Random Walk for Minimizing Total Weighted Tardiness in a Job Shop", *Journal of Scheduling*, Vol. 3, pp. 125–138.

S. Krishnaswamy and S. Nettles (2005) "Scheduling Challenges for 300mm Manufacturing", in *Proceedings of the the International Conference on Modeling and Analysis of Semiconductor Manufacturing (MASM2005)*, Singapore, October 2005.

P. Ku and S.-C. Niu (1986) "On Johnson's Two-Machine Flow Shop with Random Processing Times", *Operations Research*, Vol. 34, pp. 130–136.

M. Kunde (1976) "Beste Schranken Beim LP-Scheduling", *Bericht 7603, Institut für Informatik and und Praktische Mathematik*, Universität Kiel.

K. Kurowski, J. Nabrzyski, A. Oleksiak and J. Weglarz (2006) "Scheduling Jobs on the Grid - Multicriteria Approach", *Computational Methods in Science and Technology*, Vol. 12, pp. 123–138.

A. Kusiak and M. Chen (1988) "Expert Systems for Planning and Scheduling Manufacturing Systems", *European Journal of Operational Research*, Vol. 34, pp. 113–130.

E. Kutanoglu and S.D. Wu (1999) "On Combinatorial Auction and Lagrangean Relaxation for Distributed Resource Scheduling", *IIE Transactions*, Vol. 31, pp. 813–826.

J. Labetoulle, E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan (1984) "Preemptive Scheduling of Uniform Machines subject to Release Dates", in *Progress in Combinatorial Optimization*, W.R. Pulleyblank (ed.), pp. 245–261, Academic Press, NY.

B.J. Lageweg, E.L. Lawler, J.K. Lenstra and A.H.G. Rinnooy Kan (1981) "Computer-Aided Complexity Classification of Deterministic Scheduling Problems", Technical Report BW 138/81, the Mathematical Centre, Amsterdam, the Netherlands.

B.J. Lageweg, E.L. Lawler, J.K. Lenstra and A.H.G. Rinnooy Kan (1982) "Computer-Aided Complexity Classification of Combinatorial Problems", *Communications of the ACM*, Vol. 25, pp 817–822.

E.L. Lawler (1973) "Optimal Sequencing of a Single Machine subject to Precedence Constraints", *Management Science*, Vol. 19, pp. 544–546.

E.L. Lawler (1977) "A 'Pseudopolynomial' Time Algorithm for Sequencing Jobs to Minimize Total Tardiness", *Annals of Discrete Mathematics*, Vol. 1, pp. 331–342.

E.L. Lawler (1978) "Sequencing Jobs to Minimize Total Weighted Completion Time subject to Precedence Constraints", *Annals of Discrete Mathematics*, Vol. 2, pp. 75–90.

E.L. Lawler (1982) "A Fully Polynomial Approximation Scheme for the Total Tardiness Problem", *Operations Research Letters*, Vol. 1, pp. 207–208.

E.L. Lawler and J. Labetoulle (1978) "On Preemptive Scheduling of Unrelated Parallel Processors by Linear Programming", *Journal of the Association of Computing Machinery*, Vol. 25, pp. 612–619.

E.L. Lawler, J.K. Lenstra and A.H.G. Rinnooy Kan (1981) "Minimizing Maximum Lateness in a Two-Machine Open Shop", *Mathematics of Operations Research*, Vol. 6, pp. 153–158; Erratum Vol. 7, pp. 635.

E.L. Lawler, J.K. Lenstra and A.H.G. Rinnooy Kan (1982) "Recent Developments in Deterministic Sequencing and Scheduling: A Survey", in *Deterministic and Stochastic Scheduling*, Dempster, Lenstra and Rinnooy Kan (eds.), pp. 35–74, Reidel, Dordrecht.

E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan and D. Shmoys (1993) "Sequencing and Scheduling: Algorithms and Complexity", in *Handbooks in Operations Research and Management Science, Vol. 4: Logistics of Production and Inventory*, S. S. Graves, A. H. G. Rinnooy Kan and P. Zipkin, (eds.), pp. 445–522, North-Holland, New York.

E.L. Lawler and C.U. Martel (1989) "Preemptive Scheduling of Two Uniform Machines to Minimize the Number of Late Jobs", *Operations Research*, Vol. 37, pp. 314–318.

G. Lawton (1992) "Genetic Algorithms for Schedule Optimization", *AI Expert*, May Issue, pp. 23–27.

C.-Y. Lee (2004) "Machine Scheduling with Availability Constraints", Chapter 22 in *Handbook of Scheduling*, J. Y.-T. Leung (ed.), Chapman and Hall/CRC, Boca Raton, Florida.

C.-Y. Lee and L. Lei (eds.) (1997) "Scheduling: Theory and Applications", *Annals of Operations Research*, Vol. 70, Baltzer, Basel.

C.-Y. Lee and C.S. Lin (1991) "Stochastic Flow Shops with Lateness-Related Performance Measures", *Probability in the Engineering and Informational Sciences*, Vol. 5, pp. 245–254.

C.-Y. Lee, L.A. Martin-Vega, R. Uzsoy and J. Hinchman (1993) "Implementation of a Decision Support System for Scheduling Semiconductor Test Operations", *Journal of Electronics Manufacturing*, Vol. 3, pp. 121–131.

C.-Y. Lee and J.D. Massey (1988) "Multi-Processor Scheduling: Combining *LPT* and *MULTIFIT*", *Discrete Applied Mathematics*, Vol. 20, pp. 233–242.

C.-Y. Lee, R. Uzsoy, and L.A. Martin-Vega (1992) "Efficient Algorithms for Scheduling Semiconductor Burn-In Operations", *Operations Research*, Vol. 40, pp. 764–995.

Y.H. Lee, K. Bhaskaran and M. Pinedo (1997) "A Heuristic to Minimize the Total Weighted Tardiness with Sequence Dependent Setups", *IIE Transactions*, Vol. 29, pp. 45–52.

P. Lefrancois, M.H. Jobin and B. Montreuil "An Object-Oriented Knowledge Representation in Real-Time Scheduling", in *New Directions for Operations Research in Manufacturing*, G. Fandel, Th. Gulledge and A. Jones (eds.), Springer Verlag, pp. 262–279.

J.K. Lenstra (1977) "Sequencing by Enumerative Methods", Mathematical Centre Tracts 69, Centre for Mathematics and Computer Science, Amsterdam.

J.K. Lenstra and A.H.G. Rinnooy Kan (1978) "Computational Complexity of Scheduling under Precedence Constraints", *Operations Research*, Vol. 26, pp. 22–35.

J.K. Lenstra and A.H.G. Rinnooy Kan (1979) "Computational Complexity of Discrete Optimization Problems", *Annals of Discrete Mathematics*, Vol. 4, pp. 121–140.

J.K. Lenstra, A.H.G. Rinnooy Kan, P. Brucker (1977) "Complexity of machine scheduling problems", *Annals of Discrete Mathematics*, Vol. 1, pp. 343–362.

V.J. Leon and S.D. Wu (1994) "Robustness Measures and Robust Scheduling for Job Shops", *IIE Transactions*, Vol. 26, pp. 32–43.

V.J. Leon, S.D. Wu and R. Storer (1994) "A Game Theoretic Approach for Job Shops in the Presence of Random Disruptions", *International Journal of Production Research*, Vol. 32, pp. 1451–1476.

J.Y.-T. Leung (ed.) (2004) *Handbook of Scheduling*, Chapman and Hall/CRC, Boca Raton, Florida.

J.Y.-T. Leung, H. Li and M. Pinedo (2005a) "Order Scheduling in an Environment with Dedicated Machines in Parallel", *Journal of Scheduling*, Vol. 8, pp. 355–386.

J.Y.-T. Leung, H. Li and M. Pinedo (2005b) "Order Scheduling Models: An Overview", in *Multidisciplinary Scheduling - Theory and Applications*, G. Kendall, E. Burke, S. Petrovic, and M. Gendreau (eds.), (Selected Papers from the 1st MISTA Conference held in Nottingham, UK, August 2003), Springer, pp. 37–53.

J.Y.-T. Leung, H. Li and M. Pinedo (2006) "Approximation Algorithms for Minimizing Total Weighted Completion Time of Orders on Identical Machines in Parallel", *Naval Research Logistics*, Vol. 53, pp. 243–260.

J.Y.-T. Leung, H. Li, M. Pinedo and C. Sriskandarajah (2005) "Open Shops with Jobs Overlap - Revisited", *European Journal of Operational Research*, Vol. 163, pp. 569–571.

J.Y.-T. Leung and M. Pinedo (2003) "Minimizing Total Completion Time on Parallel Machines with Deadline Constraints", *SIAM Journal of Computing*, Vol. 32, pp. 1370–1388.

J.Y.-T. Leung and M. Pinedo (2004) "Scheduling Jobs on Parallel Machines that are subject to Breakdowns", *Naval Research Logistics*, Vol. 51, pp. 60–71.

J.Y.-T. Leung and G.H. Young (1990) "Minimizing Total Tardiness on a Single Machine with Precedence Constraint", *ORSA Journal on Computing*, Vol. 2, pp. 346–352.

E.M. Levner (1969) "Optimal Planning of Parts Machining on a Number of Machines", *Automation and Remote Control*, Vol. 12, pp. 1972–1981.

C.-F. Liaw (2000) "A Hybrid Genetic ALgorithm for the Open Shop Scheduling Problem", *European Journal of Operational Research*, Vol. 124, pp. 28–42.

C.-L. Li (2006) "Scheduling Unit-Length Jobs with Machine Eligibility Restrictions", *European Journal of Operational Research*, Vol. 174, pp. 1325–1328.

Y. Lin and W. Li (2004) "Parallel Machine Scheduling of Machine-Dependent Jobs with Unit-Length", *European Journal of Operational Research*, Vol. 156, pp. 261–266.

C.Y. Liu and R.L. Bulfin (1985) "On the Complexity of Preemptive Open-Shop Scheduling Problems", *Operations Research Letters*, Vol. 4, pp. 71–74.

Z.A. Lomnicki (1965) "A Branch and Bound Algorithm for the Exact Solution of the Three-Machine Scheduling Problem", *Operational Research Quarterly*, Vol. 16, pp. 89–100.

P.B. Luh and D.J. Hoitomt (1993) "Scheduling of Manufacturing Systems Using the Lagrangean Relaxation Technique", *IEEE Transactions on Automatic Control, Special Issue: Meeting the Challenge of Computer Science in the Industrial Applications of Control*, Vol. 38, pp. 1066–1080.

P.B. Luh, D.J. Hoitomt, E. Max and K.R. Pattipati (1990) "Schedule Generation and Reconfiguration for Parallel Machines", *IEEE Transactions on Robotics and Automation*, Vol. 6, pp. 687–696.

I.J. Lustig and J.-F. Puget (2001) "Program does not Equal Program: Constraint Programming and Its Relationship to Mathematical Programming", *Interfaces*, Vol. 31, No. 6, pp. 29–53.

M. Marchesi, E. Rusconi, and F. Tiozzo (1999) "Yogurt Process Production: A Flow Shop Scheduling Solution", Cybertec Technical Report, Cybertec, Trieste, Italy.

A.W. Marshall and I. Olkin (1979) *Inequalities: Theory of Majorization and its Applications*, Academic Press, New York.

C.U. Martel (1982) "Preemptive Scheduling with Release Times, Deadlines and Due Times", *Journal of the Association of Computing Machinery*, Vol. 29, pp. 812–829.

J. Martin (1993) *Principles of Object-Oriented Analysis and Design*, Prentice-Hall, Englewood Cliffs, New Jersey.

H. Matsuo (1990) "Cyclic Sequencing Problems in the Two-machine Permutation Flow Shop: Complexity, Worst-Case and Average Case Analysis", *Naval Research Logistics*, Vol. 37, pp. 679–694.

H. Matsuo, C.J. Suh and R.S. Sullivan (1988) "A Controlled Search Simulated Annealing Method for the General Job Shop Scheduling Problem", Working Paper 03–44–88, Graduate School of Business, University of Texas, Austin.

S.T. McCormick and M. Pinedo (1995) "Scheduling n Independent Jobs on m Uniform Machines with both Flow Time and Makespan Objectives: a Parametric Analysis," *ORSA Journal of Computing*, Vol. 7, pp. 63–77.

S.T. McCormick, M. Pinedo, S. Shenker and B. Wolf (1989) "Sequencing in an Assembly Line with Blocking to Minimize Cycle Time", *Operations Research*, Vol. 37, pp. 925–936.

S.T. McCormick, M. Pinedo, S. Shenker and B. Wolf (1990) "Transient Behavior in a Flexible Assembly System", *The International Journal of Flexible Manufacturing Systems*, Vol. 3, pp. 27–44.

K. McKay, M. Pinedo and S. Webster (2002) "A Practice-Focused Agenda for Production Scheduling Research", *Production and Operations Management*, Vol. 11, p. 249–258.

K. McKay, F. Safayeni and J. Buzacott (1988) "Job Shop Scheduling Theory: What is Relevant?", *Interfaces*, Vol. 18, No. 4, pp. 84–90.

G.B. McMahon and M. Florian (1975) "On Scheduling with Ready Times and Due dates to Minimize Maximum Lateness", *Operations Research*, Vol. 23, pp. 475–482.

R. McNaughton (1959) "Scheduling with Deadlines and Loss Functions", *Management Science*, Vol. 6, pp. 1–12.

S.V. Mehta and R. Uzsoy (1999) "Predictable Scheduling of a Single Machine subject to Breakdowns", *International Journal of Computer Integrated Manufacturing*, Vol. 12, pp. 15–38.

D. Merkle and M. Middendorf (2003) "Ant Colony Optimization with Global Pheromone Evaluation for Scheduling a Single Machine", *Applied Intelligence*, Vol. 18, pp. 105–111.

R.H. Möhring and F.J. Radermacher(1985a) "Generalized Results on the Polynomiality of Certain Weighted Sum Scheduling Problems", *Methods of Operations Research*, Vol. 49, pp. 405–417.

R.H. Möhring and F.J. Radermacher(1985b) "An Introduction to Stochastic Scheduling Problems", in *Contributions to Operations Research*, K. Neumann and D. Pallaschke (eds.), Lecture Notes in Economics and Mathematical Systems, No. 240, pp. 72–130, Springer Verlag, Berlin.

R.H. Möhring, F.J. Radermacher and G. Weiss (1984) "Stochastic Scheduling Problems I: General Strategies", *Zeitschrift für Operations Research*, Vol. 28, pp. 193–260.

R.H. Möhring, F.J. Radermacher and G. Weiss (1985) "Stochastic Scheduling Problems II: Set Strategies", *Zeitschrift für Operations Research*, Vol. 29, pp. 65–104.

R.H. Möhring, A.S. Schulz and M. Uetz (1999) "Approximation in Stochastic Scheduling: The Power of LP-Based Priority Policies", *Journal of the ACM*, Vol. 46, pp. 924–942.

C.L. Monma and C.N. Potts (1989) "On the Complexity of Scheduling with Batch Setup Times", *Operations Research*, Vol. 37, pp. 798–804.

C.L. Monma and A.H.G. Rinnooy Kan (1983) "A Concise Survey of Efficiently Solvable Special Cases of the Permutation Flow-Shop Problem", *RAIRO Recherche Operationelle*, Vol. 17, pp. 105–119.

C.L. Monma and J.B. Sidney (1979) "Sequencing with Series-Parallel Precedence Constraints", *Mathematics of Operations Research*, Vol. 4, pp. 215–224.

C.L. Monma and J.B. Sidney (1987) "Optimal Sequencing via Modular Decomposition: Characterizations of sequencing Functions", *Mathematics of Operations Research*, Vol. 12, pp. 22–31.

J.M. Moore (1968) "An n Job, One Machine Sequencing Algorithm for Minimizing the Number of Late Jobs", *Management Science*, Vol. 15, pp. 102–109.

T.E. Morton and D. Pentico (1993) *Heuristic Scheduling Systems*, John Wiley, NY.

R. Motwani, S. Phillips, and E. Torng (1994) "Nonclairvoyant Scheduling", *Theoretical Computer Science*, Vol. 130, pp. 17–47.

E.J. Muth (1979) "The Reversibility Property of a Production Line", *Management Science*, Vol. 25, pp. 152–158.

J.F. Muth and G.L. Thompson (eds.) (1963) *Industrial Scheduling*, Prentice-Hall, NJ.

P. Nain, P. Tsoucas and J. Walrand (1989) "Interchange Arguments in Stochastic Scheduling", *Journal of Applied Probability*, Vol. 27, pp. 815–826.

A. Nareyek (ed.) (2001) *Local Search for Planning and Scheduling*, Revised Papers of ECAI 2000 Workshop in Berlin, Germany, Lecture Notes in Computer Science, No. 2148, Springer Verlag, Berlin.

R.T. Nelson, R.K. Sarin, and R.L. Daniels (1986) "Scheduling with Multiple Performance Measures: The One Machine Case", *Management Science*, Vol. 32, pp. 464–479.

G.L. Nemhauser and L.A. Wolsey (1988) *Integer and Combinatorial Optimization*, John Wiley, New York.

C.T. Ng, T.C.E. Cheng and J.J. Yuan (2003) "Concurrent Open Shop Scheduling to Minimize the Weighted Number of Tardy Jobs", *Journal of Scheduling*, Vol. 6, pp. 405–412.

S.J. Noronha and V.V.S. Sarma (1991) "Knowledge-Based Approaches for Scheduling Problems: A Survey", *IEEE Transactions on Knowledge and Data Engineering*, Vol. 3, pp. 160–171.

E. Nowicki and C. Smutnicki (1996) "A Fast Taboo Search algorithm for the Job Shop Problem", *Management Science*, Vol. 42, pp. 797–813.

E. Nowicki and S. Zdrzalka (1986) "A Note on Minimizing Maximum Lateness in a One-Machine Sequencing Problem with Release Dates", *European Journal of Operational Research*, Vol. 23, pp. 266–267.

W.P.M. Nuijten (1994) *Time and Resource Constrained Scheduling; A Constraint Satisfaction Approach*, Ph.D Thesis, Eindhoven University of Technology, Eindhoven, the Netherlands.

W.P.M. Nuijten and E.H.L. Aarts (1996) "A Computational Study of Constraint Satisfaction for Multiple Capacitated Job Shop Scheduling", *European Journal of Operational Research*, Vol. 90, pp. 269–284.

M. Nussbaum and E.A. Parra (1993) "A Production Scheduling System", *ORSA Journal on Computing*, Vol. 5, pp. 168–181.

A. Oddi and N. Policella (2007) "Improving Robustness of Spacecraft Downlink Schedules", *IEEE Transactions on Systems, Man, and Cybernetics – Part C: Applications and Reviews*, Vol. 37, pp. 887–896.

M.D. Oliff (ed.) (1988) *Expert Systems and Intelligent Manufacturing*, *Proceedings of the Second International Conference on Expert Systems and the Leading Edge in Production Planning and Control*, held May 1988 in Charleston, South Carolina, Elsevier, New York.

I.M. Ovacik and R. Uzsoy (1997) *Decomposition Methods for Complex Factory Scheduling Problems*, Kluwer Academic Publishers, Boston.

P.S. Ow (1985) "Focused Scheduling in Proportionate Flowshops", *Management Science*, Vol. 31, pp. 852–869.

P.S. Ow and T.E. Morton (1988) "Filtered Beam Search in Scheduling", *International Journal of Production Research*, Vol. 26, pp. 297–307.

P.S. Ow and T.E. Morton (1989) "The Single Machine Early/Tardy Problem", *Management Science*, Vol. 35, pp. 177–191.

P.S. Ow, S.F. Smith and R. Howie (1988) "A Cooperative Scheduling System", in *Expert Systems and Intelligent Manufacturing*, M. D. Oliff, (ed.), Elsevier, Amsterdam, pp. 43–56.

D.S. Palmer (1965) "Sequencing Jobs Through a Multi-Stage Process in the Minimum Total Time - A Quick Method of Obtaining a Near Optimum", *Operational Research Quarterly*, Vol. 16, pp. 101–107.

S.S. Panwalkar and W. Iskander (1977) "A Survey of Scheduling Rules", *Operations Research*, Vol. 25, pp. 45–61.

S.S. Panwalkar, M.L. Smith and A. Seidmann (1982) "Common Due Date Assignment to Minimize Total Penalty for the One-Machine Scheduling Problem", *Operations Research*, Vol. 30, pp. 391–399.

C.H. Papadimitriou (1994) *Computational Complexity*, Addison-Wesley, Reading, MA.

C.H. Papadimitriou and P.C. Kannelakis (1980) "Flowshop Scheduling with Limited Temporary Storage", *Journal of the Association of Computing Machinery*, Vol. 27, pp. 533–549.

C.H. Papadimitriou and K. Steiglitz (1982) *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, NJ.

R.G. Parker (1995) *Deterministic Scheduling Theory*, Chapman & Hall, London.

R.G. Parker and R.L. Rardin (1988) *Discrete Optimization*, Academic Press, San Diego.

S. Park, N. Raman and M.J. Shaw (1997) "Adaptive Scheduling in Dynamic Flexible Manufacturing Systems: A Dynamic Rule Selection Approach", *IEEE Transactions on Robotics and Automation*, Vol. 13, pp. 486–502.

E. Pesch (1994) *Learning in Automated Manufacturing - A Local Search Approach*, Physica-Verlag (A Springer Company), Heidelberg, Germany.

J.R. Pimentel (1990) *Communication Networks for Manufacturing*, Prentice-Hall, NJ.

M. Pinedo (1981a) "Minimizing Makespan with Bimodal Processing Time Distributions", *Management Science*, Vol. 27, pp. 582–586.

M. Pinedo (1981b) "A Note on the Two-Machine Job Shop with Exponential Processing Times", *Naval Research Logistics Quarterly*, Vol. 28, pp. 693–696.

M. Pinedo (1982) "Minimizing the Expected Makespan in Stochastic Flow Shops", *Operations Research*, Vol. 30, pp. 148–162.

M. Pinedo (1983) "Stochastic Scheduling with Release Dates and Due Dates", *Operations Research*, Vol. 31, pp. 559–572.

M. Pinedo (1984) "A Note on the Flow Time and the Number of Tardy Jobs in Stochastic Open Shops", *European Journal of Operational Research*, Vol. 18, pp. 81–85.

M. Pinedo (1985) "A Note on Stochastic Shop Models in which Jobs have the Same Processing Requirements on each Machine", *Management Science*, Vol. 31, pp. 840–845.

M. Pinedo (2005) *Planning and Scheduling in Manufacturing and Services*, Springer Series in Operations Research, Springer, New York.

M. Pinedo (2007) "Stochastic Batch Scheduling and the "Smallest Variance First" Rule", *Probability in the Engineering and Informational Sciences*, Vol. 21, pp. pp. 579–595.

M. Pinedo and X. Chao (1999) *Operations Scheduling with Applications in Manufacturing and Services*, Irwin/McGraw-Hill, Burr Ridge, IL.

M. Pinedo and E. Rammouz (1988) "A Note on Stochastic Scheduling on a Single Machine Subject to Breakdown and Repair", *Probability in the Engineering and Informatiional Sciences*, Vol. 2, pp.41–49.

M. Pinedo and S.M. Ross (1980) "Scheduling Jobs under Nonhomogeneous Poisson Shocks", *Management Science*, Vol. 26, pp. 1250–1258.

M. Pinedo and S.M. Ross (1982) "Minimizing Expected Makespan in Stochastic Open Shops", *Advances in Applied Probability*, Vol. 14, pp. 898–911.

M. Pinedo, R. Samroengraja and P.-C. Yen (1994) "Design Issues with Regard to Scheduling Systems in Manufacturing", in *Control and Dynamic Systems*, C. Leondes (ed.), Vol. 60, pp. 203–238, Academic Press, San Diego, California.

M. Pinedo and Z. Schechner (1985) "Inequalities and Bounds for the Scheduling of Stochastic Jobs on Parallel Machines", *Journal of Applied Probability*, Vol. 22, pp. 739–744.

M. Pinedo and M. Singer (1999) "A Shifting Bottleneck Heuristic for Minimizing the Total Weighted Tardiness in a Job Shop", *Naval Research Logistics*, Vol. 46, pp. 1–12.

M. Pinedo and R.R. Weber (1984) "Inequalities and Bounds in Stochastic Shop Scheduling", *SIAM Journal of Applied Mathematics*, Vol. 44, pp. 869–879.

M. Pinedo and G. Weiss (1979) "Scheduling of Stochastic Tasks on Two Parallel Processors", *Naval Research Logistics Quarterly*, Vol. 26, pp. 527–535.

M. Pinedo and G. Weiss (1984) "Scheduling Jobs with Exponentially Distributed Processing Times and Intree Precedence Constraints on Two Parallel Machines", *Operations Research*, Vol. 33, pp. 1381–1388.

M. Pinedo and G. Weiss (1987) "The "Largest Variance First" Policy in some Stochastic Scheduling Problems", *Operations Research*, Vol. 35, pp. 884–891.

M. Pinedo and S.-H. Wie (1986) "Inequalities for Stochastic Flow Shops and Job Shops", *Applied Stochastic Models and Data Analysis*, Vol. 2, pp. 61–69.

M. Pinedo, B. Wolf and S.T. McCormick (1986) "Sequencing in a Flexible Assembly Line with Blocking to Minimize Cycle Time", in *Proceedings of the Second ORSA/TIMS Conference on Flexible Manufacturing Systems*, K. Stecke and R. Suri (eds.), Elsevier, Amsterdam, pp. 499–508.

M. Pinedo and B. P.-C. Yen (1997) "On the Design and Development of Object-Oriented Scheduling Systems", *Annals of Operations Research*, Vol. 70, C.-Y. Lee and L. Lei (eds.), pp. 359–378.

E. Pinson (1995) "The Job Shop Scheduling Problem: A Concise Survey and Some Recent Developments", in *Scheduling Theory and Applications*, P. Chrétienne, E.G. Coffman, Jr., J.K. Lenstra, and Z. Liu (eds.), John Wiley, New York, pp. 177–293.

N. Policella, A. Cesta, A. Oddi and S.F. Smith (2005) "Schedule Robustness through Broader Solve and Robustify Search for Partial Order Schedules", in *AI*IA-05 Advances in Artificial Intelligence - Proceedings of the 9th Congress of the Italian Association for Artificial Intelligence, (held in Milan, Italy, September 2005)*, S. Bandini and S. Manzoni (eds.), Springer, pp. 160–172.

N. Policella, A. Cesta, A. Oddi and S.F. Smith (2007) "From Precedence Constraint Posting to Partial Order Schedules: A CSP Approach to Robust Scheduling", *AI Communications*, Vol. 20, pp. 163–180.

M.E. Posner (1985) "Minimizing Weighted Completion Times with Deadlines", *Operations Research*, Vol. 33, pp. 562–574.

C.N. Potts (1980) "Analysis of a Heuristic for One Machine Sequencing with Release Dates and Delivery Times", *Operations Research*, Vol. 28, pp. 1436–1441.

C.N. Potts and M.Y. Kovalyov (2000) "Scheduling with Batching: A Review", *European Journal of Operational Research*, Vol. 120, pp. 228–249.

C.N. Potts and L.N. van Wassenhove (1982) "A Decomposition Algorithm for the Single Machine Total Tardiness Problem", *Operations Research Letters*, Vol. 1, pp. 177–181.

C.N. Potts and L.N. van Wassenhove (1983) "An Algorithm for Single Machine Sequencing with Deadlines to Minimize Total Weighted Completion Time", *European Journal of Operational Research*, Vol. 12, pp. 379–387.

C.N. Potts and L.N. van Wassenhove (1985) "A Branch and Bound Algorithm for the Total Weighted Tardiness Problem", *Operations Research*, Vol. 33, pp. 363–377.

C.N. Potts and L.N. van Wassenhove (1987) "Dynamic Programming and Decomposition Approaches for the Single Machine Total Tardiness Problem", *European Journal of Operational Research*, Vol. 32, pp. 405–414.

C.N. Potts and L.N. van Wassenhove (1988) "Algorithms for Scheduling a Single Machine to Minimize the Weighted Number of Late Jobs", *Management Science*, Vol. 34, pp. 843–858.

P. Priore, D. de la Fuente, A. Gomez and J. Puente (2001) "A Review of Machine Learning in Dynamic Scheduling of Flexible Manufacturing Systems", *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, Vol. 15, pp. 251–263.

K. Pruhs, J. Sgall, and E. Torng (2004) "Online Scheduling", Chapter 15 in *Handbook of Scheduling*, J. Y-T. Leung (ed.), Chapman and Hall/CRC, Boca Raton, Florida.

M. Queyranne (1993) "Structure of a Simple Scheduling Polyhedron", *Mathematical Programming*, Vol. 58, pp. 263–286.

M. Queyranne and A.S. Schulz (1994) "Polyhedral Approaches to Machine Scheduling", Preprint No. 408/1994, Fachbereich 3 Mathematik, Technische Universität Berlin, Berlin.

M. Queyranne and Y. Wang (1991) "Single Machine Scheduling Polyhedra with Precedence Constraints", *Mathematics of Operations Research*, Vol. 16, pp. 1–20.

R.M.V. Rachamadugu (1987) "A Note on the Weighted Tardiness Problem", *Operations Research*, Vol. 35, pp. 450–452.

M. Raghavachari (1988) "Scheduling Problems with Non-Regular Penalty Functions: A Review", *Opsearch*, Vol. 25, pp. 144–164.

S.S. Reddi and C.V. Ramamoorthy (1972) "On the Flowshop Sequencing Problem with No Wait in Process", *Operational Research Quarterly*, Vol. 23, pp.323–330.

J.-C. Régin and M. Rueher (eds.) (2004) *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, Proceedings of the First International Conference, CPAIOR 2004, held in Nice, France, April 2004, Lecture Notes in Computer Science, No. 3011, Springer.

J. Rickel (1988) "Issues in the Design of Scheduling Systems", in *Expert systems and intelligent manufacturing*, M.D. Oliff (ed.), pp. 70–89, Elsevier, NY.

R. Righter (1988) "Job Scheduling to Minimize Expected Weighted Flow Time on Uniform Processors", *System and Control Letters*, Vol. 10, pp 211–216.

R. Righter (1992) "Loading and Sequencing on Parallel Machines", *Probability in the Engineering and Informational Sciences*, Vol. 6, pp. 193–201.

R. Righter (1994) "Stochastic Scheduling", Chapter 13 in *Stochastic Orders*, M. Shaked and G. Shanthikumar (eds.), Academic Press, San Diego.

R. Righter and S. Xu (1991a) "Scheduling Jobs on Heterogeneous Processors", *Annals of Operations Research*, Vol. 28, pp. 587–602.

R. Righter and S. Xu (1991b) "Scheduling Jobs on Nonidentical IFR Processors to Minimize General Cost Functions", *Advances in Applied Probability*, Vol. 23, pp. 909–924.

A.H.G. Rinnooy Kan (1976) *Machine Scheduling Problems: Classification, Complexity and Computations*, Nijhoff, The Hague.

H. Röck (1984) "The Three-Machine No-Wait Flow Shop Problem is NP-Complete", *Journal of the Association of Computing Machinery*, Vol. 31, pp. 336–345.

F.A. Rodammer and K.P. White (1988) "A Recent Survey of Production Scheduling", *IEEE Transactions on Systems, Man and Cybernetics*, Vol. 18, pp. 841–851.

T.A. Roemer (2006) "A Note on the Complexity of the Concurrent Open Shop Problem", *Journal of Scheduling*, Vol. 9, pp. 389–396.

S.M. Ross (1983) *Introduction to Stochastic Dynamic Programming*, Academic Press, New York.

S.M. Ross (1995) *Stochastic Processes (Second Edition)*, John Wiley, New York.

S.M. Ross (2006) *Introduction to Probability Models (Ninth Edition)*, Academic Press, New York.

M.H. Rothkopf (1966a) "Scheduling Independent Tasks on Parallel Processors", *Management Science*, Vol. 12, pp. 437–447.

M.H. Rothkopf (1966b) "Scheduling with Random Service Times", *Management Science*, Vol. 12, pp. 707–713.

M.H. Rothkopf and S.A. Smith (1984) "There are no Undiscovered Priority Index Sequencing Rules for Minimizing Total Delay Costs", *Operations Research*, Vol. 32, pp. 451–456.

R. Roundy (1992) "Cyclic Schedules for Job Shops with Identical Jobs", *Mathematics of Operations Research,* Vol. 17, pp. 842–865.

R. Roundy, W. Maxwell, Y. Herer, S. Tayur and A. Getzler (1991) "A Price-Directed Approach to Real-Time Scheduling of Production Operations", *IIE Transactions*, Vol. 23, pp. 449–462.

B. Roy and B. Sussmann (1964) "Les Problemes d'Ordonnancement avec Constraintes Disjonctives", Note DS No. 9 bis, SEMA, Montrouge.

I. Sabuncuoglu and M. Bayiz (2000) "Analysis of Reactive Scheduling Problems in a Job Shop Environment", *European Journal of Operational Research*, Vol. 126, pp. 567–586.

I. Sabuncuoglu and A. Toptal (1999) "Distributed Scheduling I: A Review of Concepts and Applications", Technical Paper No: IEOR 9910, Department of Industrial Engineering, Bilkent University, Ankara, Turkey.

I. Sabuncuoglu and A. Toptal (1999) "Distributed Scheduling II: Bidding Algorithms and Performance Evaluations", Technical Paper No: IEOR 9911, Department of Industrial Engineering, Bilkent University, Ankara, Turkey.

S. Sahni (1976) "Algorithms for Scheduling Independent Tasks", *Journal of the Association of Computing Machinery*, Vol. 23, pp. 116–127.

S. Sahni and Y. Cho (1979a) "Complexity of Scheduling Jobs with no Wait in Process", *Mathematics of Operations Research*, Vol. 4, pp. 448–457.

S. Sahni and Y. Cho (1979b) "Scheduling Independent Tasks with Due Times on a Uniform Processor System", *Journal of the Association of Computing Machinery*, Vol. 27, pp. 550–563.

T. Sandholm (1993) "An Implementation of the Contract Net Protocol Based on Marginal Cost Calculations", in *Proceedings of the 11th National Conference on Artificial Intelligence (AAAI-93)*, pp. 256–262.

S.C. Sarin, S. Ahn and A.B. Bishop (1988) "An Improved Branching Scheme for the Branch and Bound Procedure of Scheduling n Jobs on m Machines to Minimize Total Weighted Flow Time", *International Journal of Production Research*, Vol. 26, pp. 1183–1191.

S.C. Sarin, G. Steiner and E. Erel (1990) "Sequencing Jobs on a Single Machine with a Common Due Date and Stochastic Processing Times", *European Journal of Operational Research*, Vol. 51, pp. 188–198.

J. Sauer (1993) "Dynamic Scheduling Knowledge for Meta-Scheduling", in *Proceedings of the Sixth International Conference on Industrial Engineering Applications of Artificial Intelligence and Expert Systems (IEA/AIE)*, Edinburgh, Scotland.

M.W.P. Savelsbergh, R.N. Uma, and J. Wein (2005) "An Experimental Study of LP-Based Approximation Algorithms for Scheduling Problems", *INFORMS Journal on Computing*, Vol. 17, pp. 123–136.

A.-W. Scheer (1988) *CIM - Computer Steered Industry*, Springer Verlag, New York.

L. Schrage (1968) "A Proof of the Optimality of the Shortest Remaining Processing Time Discipline", *Operations Research*, Vol. 16, pp. 687–690.

A. Schrijver (1998) *Theory of Linear and Integer Programming*, John Wiley, New York.

A. Schrijver (2003) *Combinatorial Optimization - Polyhedra and Efficiency*, Springer, Berlin.

P. Schuurman and G.J. Woeginger (1999) "Polynomial Time Approximation Algorithms for Machine Scheduling: Ten Open Problems", *Journal of Scheduling*, Vol. 2, pp. 203–214.

P. Schuurman and G.J. Woeginger (2007) "Approximation Schemes - A Tutorial", Chapter in *Lectures on Scheduling*, R.H. Möhring, C.N. Potts, A.S. Schulz, G.J. Woeginger, and L.A. Wolsey (eds.), to appear.

R. Sethi (1977) "On the Complexity of Mean Flow Time Scheduling", *Mathematics of Operations Research*, Vol. 2, pp. 320–330.

S.V. Sevastianov and G. Woeginger (1998) "Makespan Minimization in Open Shops: a Polynomial Time Approximation Scheme", *Mathematical Programming*, Vol. 82, pp. 191–198.

J. Sgall (1998) "On-line Scheduling", Chapter 9 in *Online Algorithms: The State of the Art*, A. Fiat and G. Woeginger (eds.), Lecture Notes in Computer Science, No. 1442, Springer Verlag, Berlin.

N. Shakhlevich, H. Hoogeveen and M. Pinedo (1998) "Minimizing Total Weighted Completion Time in a Proportionate Flow Shop", *Journal of Scheduling*, Vol. 1, pp. 157–168.

G. Shanthikumar and D. Yao (1991) "Bivariate Characterization of Some Stochastic Order Relations", *Advances in Applied Probability*, Vol. 23, pp. 642–659.

M.J. Shaw (1987) "A Distributed Scheduling Method for Computing Integrated Manufacturing: the Use of Local Area Networks in Cellular Systems," *International Journal of Production Research*, Vol. 25, pp. 1285–1303.

M.J. Shaw (1988a) "Dynamic Scheduling in Cellular Manufacturing Systems; A Framework for Networked Decision Making", *Journal of Manufacturing Systems*, Vol. 7, pp. 83–94.

M.J. Shaw (1988b) "Knowledge-Based Scheduling in Flexible Manufacturing Systems: An Integration of Pattern-Directed Inference and Heuristic Search", *International Journal of Production Research*, Vol. 6, pp. 821–844.

M.J. Shaw (1989) "FMS Scheduling as Cooperative Problem Solving", *Annals of Operations Research*, Vol. 17, pp. 323–346.

M.J. Shaw, S. Park and N. Raman (1992) "Intelligent Scheduling with Machine Learning Capabilities: The Induction of Scheduling Knowledge", *IIE Transactions on Design and Manufacturing*, Vol. 24, pp. 156–168.

M.J. Shaw and A.B. Whinston (1989) "An Artificial Intelligence Approach to the Scheduling of Flexible Manufacturing Systems", *IIE Transactions*, Vol. 21, pp. 170–183.

D.B. Shmoys, J. Wein and D.P. Williamson (1995) "Scheduling Parallel Machines On-line", *SIAM Journal of Computing*, Vol. 24, pp. 1313–1331.

J.B. Sidney (1977) "Optimal Single Machine Scheduling with Earliness and Tardiness Penalties", *Operations Research*, Vol. 25, pp. 62–69.

J.B. Sidney and G. Steiner (1986) "Optimal Sequencing by Modular Decomposition: Polynomial Algorithms", *Operations Research*, Vol. 34, pp. 606–612.

B. Simons (1983) "Multiprocessor Scheduling of Unit-Time Jobs with Arbitrary Release Times and Deadlines", *SIAM Journal of Computing*, Vol. 12, pp. 294–299.

M. Singer and M. Pinedo (1998) "A Computational Study of Branch and Bound Techniques for Minimizing the Total Weighted Tardiness in Job Shops", *IIE Transactions Scheduling and Logistics*, Vol. 30, pp. 109–118.

M.L. Smith, S.S. Panwalkar and R.A. Dudek (1975) "Flow Shop Sequencing with Ordered Processing Time Matrices", *Management Science*, Vol. 21, pp. 544–549.

M.L. Smith, S.S. Panwalkar and R.A. Dudek (1976) "Flow Shop Sequencing Problem with Ordered Processing Time Matrices: A General Case", *Naval Research Logistics Quarterly*, Vol. 23, pp. 481–486.

S.F. Smith (1992) "Knowledge-based Production Management: Approaches, Results and Prospects", *Production Planning and Control*, Vol. 3, pp. 350–380.

S.F. Smith (1994) "OPIS: a Methodology and Architecture for Reactive Scheduling", in *Intelligent Scheduling*, M. Zweben and M. Fox (eds.), Morgan and Kaufmann, San Francisco, California.

S.F. Smith and M.A. Becker (1997) "An Ontology for Constructing Scheduling Systems", in Working Notes from 1997 AAAI Spring Symposium on Ontological Engineering, Stanford University, Stanford, California.

S.F. Smith, M.S. Fox and P.S. Ow (1986) "Constructing and Maintaining Detailed Production Plans: Investigations into the Development of Knowledge-Based Factory Scheduling Systems", *AI Magazine*, Vol. 7, pp. 45–61.

S.F. Smith and O. Lassila (1994) "Configurable Systems for Reactive Production Management", in *Knowledge-Based Reactive Scheduling (B–15)*, E. Szelke and R.M. Kerr (eds.), Elsevier Science, North Holland, Amsterdam.

S.F. Smith, N. Muscettola, D.C. Matthys, P.S. Ow and J. Y. Potvin (1990) "OPIS: An Opportunistic Factory Scheduling System", *Proceedings of the Third International Conference on Industrial and Expert Systems, (IEA/AIE 90)*, Charleston, South Carolina.

W.E. Smith (1956) "Various Optimizers for Single Stage Production", *Naval Research Logistics Quarterly*, Vol. 3, pp. 59–66.

J.J. Solberg (1989) "Production Planning and Scheduling in CIM", *Information Processing*, Vol. 89, pp. 919–925.

C. Sriskandarajah and S.P. Sethi (1989) "Scheduling Algorithms for Flexible Flow Shops: Worst and Average Case Performance", *European Journal of Operational Research*, Vol. 43, pp. 143–160.

M.S. Steffen (1986) "A Survey of Artificial Intelligence-Based Scheduling Systems", in *Proceedings of Fall 1986 Industrial Engineering Conference*, Institute of Industrial Engineers, pp. 395–405.

M. Stibor and M. Kutil (2006) "Torsche Scheduling Toolbox: List Scheduling", in *Proceedings of Process Control 2006*, University of Pardubice, Pardubice, Czech Republic.

R.H. Storer, S.D. Wu and R. Vaccari (1992) "New Search Spaces for Sequencing Problems with Application to Job Shop Scheduling", *Management Science*, Vol. 38, pp. 1495–1509.

D.R. Sule (1996) *Industrial Scheduling*, PWS Publishing Company, Boston.

G. Sullivan and K. Fordyce (1990) "IBM Burlington's Logistics Management System", *Interfaces*, Vol. 20, pp. 43–64.

C.S. Sung and S.H. Yoon (1998) "Minimizing Total Weighted Completion Time at a Pre-Assembly Stage Composed of Two Feeding Machines", *International Journal of Production Economics*, Vol. 54, pp. 247–255.

S. Suresh, R.D. Foley and S.E. Dickey (1985) "On Pinedo's Conjecture for Scheduling in a Stochastic Flow Shop", *Operations Research*, Vol. 33, pp. 1146–1153.

W. Szwarc (1971) "Elimination Methods in the $m \times n$ Sequencing Problem", *Naval Research Logistics Quarterly*, Vol. 18, pp. 295–305.

W. Szwarc (1973) "Optimal Elimination Methods in the $m \times n$ Flow Shop Scheduling Problem", *Operations Research*, Vol. 21, pp. 1250–1259.

W. Szwarc (1978) "Dominance Conditions for the Three-Machine Flow-Shop Problem", *Operations Research*, Vol. 26, pp. 203–206.

W. Szwarc (1998) "Decomposition in Single-Machine Scheduling", *Annals of Operations Research*, Vol. 83, pp. 271–287.

W. Szwarc and S.K. Mukhopadhyay (1995) "Optimal Timing Schedules in Earliness-Tardiness Single Machine Sequencing", *Naval Research Logistics*, Vol. 42, pp. 1109–1114.

E. Taillard (1990) "Some Efficient Heuristic Methods for the Flow Shop Sequencing Problem", *European Journal of Operational Research*, Vol. 47, pp. 65–74.

P.P. Talwar (1967) "A Note on Sequencing Problems with Uncertain Job Times", *Journal of the Operations Research Society of Japan*, Vol. 9, pp. 93–97.

C.S. Tang (1990) "Scheduling Batches on Parallel Machines with Major and Minor Setups", *European Journal of Operational Research*, Vol. 46, pp. 28–37.

T. Tautenhahn and G.J. Woeginger (1997) "Minimizing the Total Completion Time in a Unit Time Open Shop with Release Times", *Operations Research Letters*, Vol. 20, pp. 207–212.

V.G. Timkovsky (1998) "Is a Unit-Time Job Shop not Easier than Identical Parallel Machines?", *Discrete Applied Mathematics*, Vol. 85, pp. 149–162.

V.G. Timkovsky (2000) "Reducibility among Scheduling Classes", Technical Report, Star Data Systems Inc., Toronto, Canada.

V. T'kindt and J.-C. Billaut (2002) *Multicriteria Scheduling - Theory, Models, and Algorithms (First Edition)*, Springer, Berlin.

V. T'kindt and J.-C. Billaut (2006) *Multicriteria Scheduling - Theory, Models, and Algorithms (Second Edition)*, Springer, Berlin.

R. Uzsoy (1993) "Decomposition Methods for Scheduling Complex Dynamic Job Shops", in *Proceedings of the 1993 NSF Design and Manufacturing Systems Conference*, Vol. 2, pp. 1253–1258, Society of Manufacturing Engineers, Dearborn, MI

R. Uzsoy, C.-Y. Lee and L.A. Martin-Vega (1992a) "Scheduling Semiconductor Test Operations: Minimizing Maximum Lateness and Number of Tardy Jobs on a Single Machine", *Naval Research Logistics*, Vol. 39, pp. 369–388.

R. Uzsoy, C.-Y. Lee and L.A. Martin-Vega (1992b) "A Review of Production Planning and Scheduling Models in the Semiconductor Industry, Part I: System Characteristics, Performance Evaluation and Production Planning", *IIE Transactions*, Vol. 24, pp. 47–61.

G. Vairaktarakis and S. Sahni (1995) "Dual Criteria Preemptive Open-Shop Problems with Minimum Makespan" *Naval Research Logistics*, Vol. 42, pp. 103–122.

J.M. Van den Akker and J.A. Hoogeveen (2004) "Minimizing the Number of Tardy Jobs", Chapter 12 in *Handbook of Scheduling*, J. Y-T. Leung (ed.), Chapman and Hall/CRC, Boca Raton, Florida.

J.M. Van den Akker, J.A. Hoogeveen, and S.L. Van de Velde (1999) "Parallel Machine Scheduling by Column Generation", *Operations Research*, Vol. 47, pp. 862–872.

L. Van der Heyden (1981) "Scheduling Jobs with Exponential Processing and Arrival Times on Identical Processors so as to Minimize the Expected Makespan", *Mathematics of Operations Research*, Vol. 6, pp. 305–312.

S.L. Van de Velde (1990) "Minimizing the Sum of the Job Completion Times in the Two-Machine Flow Shop by Lagrangian Relaxation", *Annals of Operations Research*, Vol. 26, pp. 257–268.

S.L. Van de Velde (1991) *Machine Scheduling and Lagrangean Relaxation*, Ph.D thesis, Eindhoven University of Technology, Eindhoven, the Netherlands.

H. Van Dyke Parunak (1991) "Characterizing the Manufacturing scheduling Problem", *Journal of Manufacturing Systems*, Vol. 10, pp. 241–259.

P. Van Hentenryck and I.J. Lustig (1999) *The OPL Optimization Programming Language*, MIT Press, Cambridge, Massachusetts.

P. Van Hentenryck and L. Michel (2004) "Scheduling Abstractions for Local Search", in *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems - Proceedings of CPAIOR 2004*, J.-C. Régin and M. Rueher (eds.), Lecture Notes in Computer Science, No. 3011, pp. 319–334, Springer.

P. Van Hentenryck and L. Michel (2005) *Constraint-Based Local Search*, MIT Press, Cambridge, Massachusetts.

P.J.M. Van Laarhoven, E.H.L. Aarts and J.K. Lenstra (1992) "Job Shop Scheduling by Simulated Annealing", *Operations Research*, Vol. 40, pp. 113–125.

L.N. Van Wassenhove and F. Gelders (1980) "Solving a Bicriterion Scheduling Problem", *European Journal of Operational Research*, Vol. 4, pp. 42–48.

A. Vepsalainen and T.E. Morton (1987) "Priority Rules and Lead Time Estimation for Job Shop Scheduling with Weighted Tardiness Costs", *Management Science*, Vol. 33, pp. 1036–1047.

G.E. Vieira, J.W. Herrmann, and E. Lin (2003) "Rescheduling Manufacturing Systems: A Framework of Strategies, Policies and Methods", *Journal of Scheduling*, Vol. 6, pp. 39–62.

H.M. Wagner (1959) "An Integer Programming Model for Machine Scheduling", *Naval Research Logistics Quarterly*, Vol. 6, pp. 131–140.

E. Wagneur and C. Sriskandarajah (1993) "Open Shops with Jobs Overlap", *European Journal of Operational Research*, Vol. 71, pp. 366–378.

G. Wan and B.P.-C. Yen (2002) "Tabu Search for Total Weighted Earliness and Tardiness Minimization on a Single Machine with Distinct Due Windows", *European Journal of Operational Research*, Vol. 142, pp. 271–281.

R.R. Weber (1982a) "Scheduling Jobs with Stochastic Processing Requirements on Parallel Machines to Minimize Makespan or Flow Time", *Journal of Applied Probability*, Vol. 19, pp. 167–182.

R.R. Weber (1982b) "Scheduling Stochastic Jobs on Parallel Machines to Minimize Makespan or Flow Time", in *Applied Probability - Computer Science: The Interface*, R. Disney and T. Ott, (eds.), pp. 327–337, Birkhauser, Boston.

R.R. Weber (1992) "On the Gittins Index for Multi-Armed Bandits", *Annals of Applied Probability*, Vol. 2, pp. 1024–1033.

R.R. Weber, P. Varaiya and J. Walrand (1986) "Scheduling Jobs with Stochastically Ordered Processing Times on Parallel Machines to Minimize Expected Flow Time", *Journal of Applied Probability*, Vol. 23, pp. 841–847.

S. Webster (2000) "Frameworks for Adaptable Scheduling Algorithms", *Journal of Scheduling*, Vol. 3, pp. 21–50.

L.M. Wein (1988) "Scheduling Semi-Conductor Wafer Fabrication", *IEEE Transactions on Semiconductor Manufacturing*, Vol. 1, pp. 115–129.

L.M. Wein and P.B. Chevelier (1992) "A Broader View of the Job-Shop Scheduling Problem", *Management Science*, Vol. 38, pp. 1018–1033.

G. Weiss (1982) "Multiserver Stochastic Scheduling", in *Deterministic and Stochastic Scheduling*, Dempster, Lenstra and Rinnooy Kan, (eds.), D. Reidel, Dordrecht, pp. 157–179.

G. Weiss (1990) "Approximation Results in Parallel Machines Stochastic Scheduling", *Annals of Operations Research*, Vol. 26, pp. 195–242.

G. Weiss and M. Pinedo (1980) "Scheduling Tasks with Exponential Processing Times to Minimize Various Cost Functions", *Journal of Applied Probability*, Vol. 17, pp.187–202.

M.P. Wellman, W.E. Walsh, P. Wurman and J.K. MacKie-Mason (2001) "Auction Protocols for Decentralized Scheduling", *Games and Economic Behavior*, Vol. 35, pp. 271–303.

P. Whittle (1980) "Multi-armed Bandits and the Gittins Index", *Journal of the Royal Statistical Society Series B*, Vol. 42, pp. 143–149.

P. Whittle (1981) "Arm Acquiring Bandits", *Annals of Probability,* Vol. 9, pp. 284–292.

M. Widmer and A. Hertz (1989) "A New Heuristic Method for the Flow Shop Sequencing Heuristic", *European Journal of Operational Research*, Vol. 41, 186–193.

V.C.S. Wiers (1997) *Human Computer Interaction in Production Scheduling: Analysis and Design of Decision Support Systems in Production Scheduling Tasks*, Ph.D Thesis, Eindhoven University of Technology, Eindhoven, the Netherlands.

D.A. Wismer (1972) "Solution of Flowshop Scheduling Problem with No Intermediate Queues", *Operations Research*, Vol. 20, pp. 689–697.

R.J. Wittrock (1985) "Scheduling Algorithms for Flexible Flow Lines", *IBM Journal of Research and Development*, Vol. 29, pp. 401–412.

R.J. Wittrock (1988) "An Adaptable Scheduling Algorithm for Flexible Flow Lines", *Operations Research*, Vol. 36, pp. 445–453.

R.J. Wittrock (1990) "Scheduling Parallel Machines with Major and Minor Setup Times", *International Journal of Flexible Manufacturing Systems*, Vol. 2, pp. 329–341.

I.W. Woerner and E. Biefeld (1993) "HYPERTEXT-Based Design of a User Interface for Scheduling", in *Proceedings of the AIAA Computing in Aerospace 9*, San Diego, California.

R.W. Wolff (1970) "Work-Conserving Priorities", *Journal of Applied Probability*, Vol. 7, pp. 327–337.

R.W. Wolff (1989) *Stochastic Modeling and the Theory of Queues*, Prentice-Hall, Englewood Cliffs, New Jersey.

L.A. Wolsey (1998) *Integer Programming,* John Wiley, New York.

S.D. Wu, E.S. Byeon and R.H. Storer (1999) "A Graph-Theoretic Decomposition of Job Shop Scheduling Problems to Achieve Schedule Robustness", *Operations Research*, Vol. 47, pp. 113–124.

S.D. Wu, R.H. Storer and P.C. Chang (1991) "A Rescheduling Procedure for Manufacturing Systems under Random Disruptions," in *New Directions for Operations Research in Manufacturing*, T. Gulledge and A. Jones (eds.), Springer Verlag, Berlin.

S.H. Xu (1991a) "Minimizing Expected Makespans of Multi-Priority Classes of Jobs on Uniform Processors", *Operations Research Letters*, Vol. 10, pp. 273–280.

S.H. Xu (1991b) "Stochastically Minimizing Total Delay of Jobs subject to Random Deadlines", *Probability in the Engineering and Informational Sciences*, Vol. 5, pp. 333–348.

S.H. Xu, P.B. Mirchandani, S.P. Kumar and R.R. Weber (1990) "Stochastic Dispatching of Multi-Priority Jobs to Heterogenous Processors", *Journal of Applied Probability*, Vol. 28, pp. 852–861.

J. Yang and M.E. Posner (2005) "Scheduling Parallel Machines for the Customer Order Problem", *Journal of Scheduling*, Vol. 8, pp. 49–74.

Y. Yang, S. Kreipl and M. Pinedo (2000) "Heuristics for Minimizing Total Weighted Tardiness in Flexible Flow Shops", *Journal of Scheduling*, Vol. 3, pp. 89–108.

C.A. Yano and A. Bolat (1989) "Survey, Development and Applications of Algorithms for Sequencing Paced Assembly Lines", *Journal of Manufacturing and Operations Management*, Vol. 2, pp. 172–198.

P.-C. Yen (1995) *On the Architecture of an Object-Oriented Scheduling System*, Ph.D thesis, Department of Industrial Engineering and Operations Research, Columbia University, New York, New York.

B.P.-C. Yen (1997) "Interactive Scheduling Agents on the Internet", in *Proceedings of the Hawaii International Conference on System Science (HICSS–30)*, Hawaii.

P.-C. Yen and M. Pinedo (1994) "Scheduling Systems: A Survey", Technical Report, Department of Industrial Engineering and Operations Research, Columbia University, New York, New York.

Y. Yih (1990) "Trace-driven Knowledge Acquisition (TDKA) for Rule-Based Real Time Scheduling Systems", *Journal of Intelligent Manufacturing*, Vol. 1, pp. 217–230.

E. Yourdon (1994) "Object-Oriented Design: An Integrated Approach", Prentice-Hall, Englewood Cliffs, New Jersey.

W. Zhang and T.G. Dietterich (1995) "A Reinforcement Learning Approach to Job-Shop Scheduling", in *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95)*, C.S. Mellish (ed.), pp. 1114–1120, Conference held in Montreal, Canada, Morgan Kaufmann Publishers, San Francisco, California.

M. Zweben and M. Fox (eds.) (1994) *Intelligent Scheduling*, Morgan Kaufmann Publishers, San Francisco, California.

Subject Index

Name Index