Yuri Malitsky

Instance-Specific Algorithm Configuration

Springer

Instance-Specific Algorithm Configuration

Yuri Malitsky

Instance-Specific Algorithm Configuration

Springer

Yuri Malitsky
IBM Thomas J. Watson Research Center
Yorktown Heights
New York
USA

ISBN 978-3-319-11229-9 ISBN 978-3-319-11230-5 (eBook)
DOI 10.1007/978-3-319-11230-5
Springer Cham Heidelberg New York Dordrecht London

Library of Congress Control Number: 2014956556

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

When developing a new heuristic or complete algorithm for a constraint satisfaction or constrained optimization problem, we frequently face the problem of choice. There may be multiple branching heuristics that we can employ, different types of inference mechanisms, various restart strategies, or a multitude of neighborhoods from which to choose. Furthermore, the way in which the choices we make affect one another is not readily perceptible. The task of making these choices is known as algorithm configuration.

Developers often make many of these algorithmic choices during the prototyping stage. Based on a few preliminary manual tests, certain algorithmic components are discarded even before all the remaining components have been implemented. However, by making the algorithmic choices beforehand developers may unknowingly discard components that are used in the optimal configuration. In addition, the developer of an algorithm has limited knowledge about the instances that a user will typically employ the solver for. That is the very reason why solvers have parameters: to enable users to fine-tune a solver for their specific needs.

Alternatively, manually tuning a parameterized solver can require significant resources, effort, and expert knowledge. Before even trying the numerous possible parameter settings, the user must learn about the inner workings of the solver to understand what each parameter does. Furthermore, it has been shown that manual tuning often leads to highly inferior performance.

This book shows how to automatically train a multi-scale, multi-task approach for enhanced performance based on machine learning techniques. In particular this work presents the methodology of Instance-Specific Algorithm Configuration (ISAC). ISAC is a general configurator that focuses on tuning different categories of parameterized solvers according to the instances they will be applied to. Specifically, this book shows that the instances of many problems can be decomposed into a representative vector of features. It further shows that instances with similar features often cause similar behavior in the applied algorithm. ISAC exploits this observation by automatically detecting the different subtypes of a problem and then training a solver for each variety. This technique is explored on a number of problem domains, including set covering, mixed integer, satisfiability,

and set partitioning. ISAC is then further expanded to demonstrate its application to traditional algorithm portfolios and adaptive search methodologies. In all cases, marked improvements are shown over the existing state-of-the-art solvers. These improvements were particularly evident during the 2011 SAT Competition, where a solver based on ISAC won seven medals, including a gold in the handcrafted instance category, and another gold in the randomly generated instance category. Later, in 2013, solvers based on this research won multiple gold medals in both the SAT Competition and the MaxSAT Evaluation.

The research behind ISAC is broken down into ten chapters. Chapters 1 and 2, respectively, introduce the problem domain and the relevant research that has been done on the topic. Chapter 3 then introduces the ISAC methodology, while Chap. 4 demonstrates its effectiveness in practice. Chapters 5, 6, and 7 demonstrate how the methodology can be applied to the problem of algorithm selection. Chapter 8 takes an alternate view and shows how ISAC can be used to create a framework that dynamically switches to the best heuristic to utilize as the problem is being solved. Chapter 9 introduces the concept that sometimes problems change over time, and that a portfolio needs a way to effectively retrain to accommodate the changes. Chapter 10 demonstrates how the ISAC methodology can be transparently expanded, while Chap. 11 wraps up the topics covered and offers ideas for future research.

The research presented in this book was carried out as the author's Ph.D. work at Brown University and his continuing work at the Cork Constraint Computation Centre. It would not have been possible without the collaboration with my advisor, Meinolf Sellmann, and my supervisor, Barry O'Sullivan. I would also like to thank all of my coauthors, who have helped to make this research possible. In alphabetical order they are: Tinus Abell, Carlos Ansotegui, Marco Collautti, Barry Hurley, Serdar Kadioglu, Lars Kotthoff, Christian Kroer, Kevin Leo, Giovanni Di Liberto, Deepak Mehta, Ashish Sabharwal, Horst Samulowitz, Helmut Simonis, and Kevin Tierney.

This work has also been partially supported by Science Foundation Ireland Grant No. 10/IN.1/I3032 and by the European Union FET grant (ICON) No. 284715.

Cork, Ireland Yuri Malitsky
September 2013

Contents

Chapter 1
Introduction

In computer science it is often the case that programs are designed to repeatedly solve many instances of the same problem. In the stock market, for example, there are programs that must continuously evaluate the value of a portfolio, determining the most opportune time to buy or sell stocks. In container stowage, each time a container ship comes into port, a program needs to find a way to load and unload its containers as quickly as possible while not compromising the ship's integrity and making sure that at the next port the containers that need to be unloaded are stacked closer to the top. In database management systems, programs need to schedule jobs and store information across multiple machines continually so that the average time to complete an operation is minimized. A robot relying on a camera needs to process images detailing the state of its current environment continually. Whenever dealing with uncertainty, as in the case of a hurricane landfall, an algorithm needs to evaluate numerous scenarios to choose the best evacuation routes. Furthermore, these applications need not only be online tasks, but can be offline as well. Scheduling airplane flights and crews for maximum profit needs to be done every so often to adjust to any changes that might occur due to disruptions, delays and mechanical issues, but such schedules do not need to be computed within a certain short allowable time frame.

In all the above-mentioned cases, and many more similar ones, the task of the program is to solve different instantiations of the same problem continually. In such applications, it is not enough just to solve the problem; it is also necessary that this be done with increasing accuracy and/or efficiency. One possible way to achieve these improvements is to have developers and researchers design progressively better algorithms. And there is still a lot of potential that can be gained through better understanding and utilization of existing techniques. Yet, while this is essential for continual progress, it is becoming obvious that there is no singular universally best algorithm. Instead, as will be shown in subsequent chapters, an algorithm that is improved for average performance must do so by sacrificing performance on some subset of cases.

Y. Malitsky, *Instance-Specific Algorithm Configuration*,
DOI 10.1007/978-3-319-11230-5_1

Furthermore, in practice, developers often make decisive choices about the internal parameters of a solver when creating it. But because a solver can be used to address many different problems, certain settings or heuristics are beneficial for one group of instances while different settings could be better for other problems. It is therefore important to develop configurable solvers, whose internal behavior can be adjusted to suit the application at hand.

Let us take the very simple example of a simulated annealing (SA) search. This probabilistic local search strategy was inspired by a phenomenon in metallurgy where repeated controlled heating and cooling would result in the formation of larger crystals with fewer defects, a desirable outcome of the process. Analogously, the search strategy tries to replace its current solution with a randomly selected neighboring solution. If this neighbor is better than the current solution, it is accepted as the new current solution. However, if this random solution is worse, it is selected with some probability depending on the current temperature parameter and how much worse it is than the current solution. Therefore, the higher the temperature, the more likely the search is to accept the new solution, thus exploring more of the search space. Alternatively, as the temperature is continually lowered as the search proceeds, SA focuses more on improving solutions and thus exploiting a particular portion of the search space. In practice, SA has been shown to be highly effective on a multitude of problems, but the key to its success lies in the initial setting of the temperature parameter and the speed with which it is lowered. Setting the temperature very high can be equivalent to random sampling but works well in a very jagged search space with many local optima. Alternatively, a low temperature is much better for quickly finding a solution in a relatively smooth search area but the search is unlikely ever to leave a local optima. The developer, however, often does not know the type of problem the user will be solving, so fixing these parameters beforehand can be highly counterproductive. Yet in many solvers, constants like the rate of decay, the frequency of restarts, the learning rate, etc. are all parameters that are deeply embedded within the solver and manually set by the developer.

Generalizing further from individual parameters, it is clear that even the deterministic choice of the employed algorithms must be left open to change. In optimization, there are several seminal papers advocating the idea of exploiting statistics and machine learning technology to increase the efficiency of combinatorial solvers. At a high level, these solvers try to find a value assignment to a collection of variables while adhering to a collection of constraints. A trivial example would be to find which combination of ten coins are needed to sum up to two euros, while requiring twice as many 10 cent coins as 20 cent coins. One feasible solution is obviously to take four 5s, four 10s, two 20s, and two 50s. One popular way to solve these types of problems is an iterative technique called branch-and-bound, where one variable is chosen to take a particular value which in turn causes some assignments to become infeasible. In our example, first choosing to take two 20 cent coins means that we must also take four 10 cent coins, which in turn means we can not take any two euro coins. Once all infeasible values are filtered out, another variable is heuristically assigned a value, and the process is repeated. If a chain of assignments leads to an infeasible solution, the solver backtracks to a

previous decision point, and tries an alternate chain of assignments to the variables. When solving these types of problems, the heuristic choice of the next variable and value pair to try is crucial, often resulting in the problem being solvable in a few seconds as being something that can run until the end of the universe. Imagine, for example, the first decision being to take at least one penny.

Yet, in practice, there is no single heuristic or approach that has been shown to be best over all scenarios. For example, it has been suggested that we gather statistics during the solution process of a constraint satisfaction problem on variable assignments that cause a lot of filtering and to base future branching decisions on this data. This technique, called impact-based search, is one of the most successful in constraint programming and has become part of the IBM Ilog CP Solver. The developer might know about the success of this approach and choose it as the only available heuristic in the solver. Yet, while the method works really well in most cases, there are scenarios where just randomly switching between multiple alternate heuristics performs just as well, if not better.

As a bottom line, while it is possible to develop solvers to improve the average-case performance, any such gains are made at the expense of decreased performance on some subset of instances. Therefore, while some solvers excel in one particular scenario, there is no single solver or algorithm that works best in all scenarios. Due to this fact, it is necessary to make solvers as accessible as possible, so users can tailor methodologies to the particular datasets of interest to them. Meanwhile, developers should make all choices available to the user.

One of the success stories of such an approach from the Boolean satisfiability domain is SATenstein [62]. Observing the ideas and differences behind some of the most successful local search SAT solvers, the creators of SATenstein noticed that all solvers followed the same general structure. The solver selected a variable in the SAT formula and then assigned it to be either true or false. The differences in the solvers were mainly due to how the decision was made to select the variable and what value was assigned to it. Upon this observation, SATenstein was developed such that all existing solvers could be replicated by simply modifying a few parameters. This not only created a single base for any existing local search SAT solver, but also allowed users to easily try new combinations of components to experiment with previously unknown solvers. It is therefore imperative to make solvers and algorithms configurable, allowing for them to be used to maximum effect for the application at hand.

Yet while solvers like SATenstein provide the user with a lot of power to fine-tune a solver to exact specifications, the problem of choice arises. SATenstein has over 40 parameters that can be defined. A mathematical programming solver like IBM Cplex [59] has over 100. Without expert knowledge of exactly how these solvers work internally, setting these parameters becomes a guessing game rather than research. Things are further complicated if a new version of the solver becomes available with new parameters or the non-linear relations between some parameters change. This also makes switching to a new solver a very expensive endeavor, requiring time and resources to become familiar with the new environment. On top of the sheer expense of manually tuning parameters, it has been consistently shown

that even the developers who originally made the solver struggle when setting the parameters manually.

The research into artificial intelligence algorithms and techniques that can automate the setting of a solver's parameters has resulted in a paradigm shift with advantages beyond improved performance of solvers. For one, research in this direction can potentially improve the quality of comparisons between existing solvers. As things are now, when a new solver needs to be compared to the existing state of the art, it is often the case that the developers find a new dataset and then carefully tweak their proposed approach. However, when running the competing solver, much less time is devoted to making sure it is running optimally. There are many possible reasons for this, but the result is the same. Through automating the configuration of solvers for the problems at hand, a much more fair comparison can be achieved. Furthermore, through this approach towards configuration, it will be possible to claim the benefits of a newly proposed heuristic or method definitively if it is automatically chosen as best for a particular dataset or, even better, if the configuration tool can automatically find the types of instances where the new approach is best.

Through tuning, researchers would also be allowed to better focus their efforts on the development of new algorithms. When improving the performance of a solver, it is important to note whether the benefits are coming from small improvements on many easy instances or a handful of hard ones. It would also be possible to identify the current bounds on performance, effectively homing in on cases where a breakthrough can have the most benefit. Furthermore, by studying benchmarks, it might be possible to discern the structural differences between these hard instances and the easy ones, which can lead to insights into what makes the problems hard and how these differences can be exploited advantageously.

Additionally, what if we can automatically identify the hard problems? What if by studying the structure of hard instances we notice that it can be systematically perturbed to make the instances easier. What if we can intelligently create hard instances that have a particular internal structure instead of randomly trying to achieve interesting benchmarks? What if we can create adaptive solvers that detect changes in the problem structure and can completely modify their strategy based on these changes?

This book presents a methodology that is motivated by these issues, discussing a clear infrastructure that can be readily expanded and applied to a variety of domains.

1.1 Outline

This book shows how to automatically train a multi-scale, multi-task approach for enhanced performance based on machine learning techniques.

Although the idea of automatically tuning algorithms is not new, the field of automatic algorithm configuration has experienced a renaissance in the past decade. There now exist a number of techniques that are designed to select a parameter set

automatically that on average works well on all instances in the training set [5, 56]. The research outlined here takes the current approaches a step further by taking into account the specific problem instances that need to be solved. Instead of assuming that there is one optimal parameter set that will yield the best performance on all instances, it assumes that there are multiple types of problems, each yielding to different strategies. Furthermore, the book assumes that there exists a finite collection of features for each instance that can be used to correctly identify its structure, and thus used to identify the subtypes of the problems. Taking advantage of these two assumptions we present Instance-Specific Algorithm Configuration (ISAC), an automated procedure to provide instance-specific tuning.

This book proceeds with an outline of related work in the field of training solvers in Chap. 2. Chapter 3 then explains the approach, ISAC, and how it is different from prior research. Chapter 4 presents the application of ISAC to set covering, mixed integer programs, satisfiability, and local search problems. Chapter 5 shows how the approach can be used to train algorithm portfolios, improving performance over existing techniques that use regression. Chapter 6 shows how ISAC can be modified to handle dynamic training, where a unique algorithm is tuned for each instance. Chapter 7 shows how to tune parallel portfolio algorithms. Chapter 8 shows how ISAC can be used to create an adaptive solver that changes its behavior based on the current subproblem observed during search. Chapter 9 introduces a scenario where problems change over time, necessitating an approach that is able to identify the most opportune moments to retrain the portfolio. In Chapter 10 we continue to study and confirm some of the assumptions made by ISAC, and then introduce modifications to refine the utilized clusterings by taking into account the performances of the solvers in a portfolio. Each of these chapters is supported by numerical evaluation. The book concludes with a discussion of the strengths and weaknesses of the ISAC methodology and of potential future work.

Chapter 2
Related Work

Automatic algorithm configuration is a quickly evolving field that aims to overcome the limitations and difficulties associated with manual parameter tuning. Many techniques have been attempted to address this problem, including metaheuristics, evolutionary computation, local search, etc. Yet despite the variability in the approaches, this flood of proposed work mainly ranges between four ideas: algorithm construction, instance-oblivious tuning, instance-specific regression, and adaptive methods. The four sections of this chapter discuss the major works for each of these respective ideas and the final section summarizes the chapter.

2.1 Algorithm Construction

Algorithm construction focuses on automatically creating a solver from an assortment of building blocks. These approaches define the structure of the desired solver, declaring how the available algorithms and decisions need to be made. A machine learning technique then evaluates different configurations of the solver, trying to find the one that performs best on a collection of training instances.

The MULTI-TAC system [80] is an example of this approach applied to the constraint satisfaction problem (CSP). The backtracking solver is defined as a sequence of rules that determine which branching variable and value selection heuristics to use under what circumstances, as well as how to perform forward checking. Using a beam search to find the best set of rules, the system starts with an empty configuration. The rules or routines are then added one at a time. A small Lisp program corresponding to these rules is created and run on the training instances. The solver that properly completes the most instances proceeds to the next iteration. The strength of this approach is the ability to represent all existing solvers while automatically finding changes that can lead to improved performance. The algorithm, however, suffers from the search techniques used to

Y. Malitsky, *Instance-Specific Algorithm Configuration*,
DOI 10.1007/978-3-319-11230-5_2

find the best configurations. Since the CSP solver is greedily built one rule or routine at a time, certain solutions can remain unobserved. Furthermore, as the number of possible routines and rules grows or the underlying schematic becomes more complicated, the number of possible configurations becomes too large for the described methodology.

Another approach from this category is the CLASS system, developed by Fukunaga [35]. This system is based on the observation that many of the existing local search (LS) algorithms used for SAT are seemingly composed of the same building blocks with only minor deviations. The Novelty solver [78], for example, is based on the earlier GWSAT solver [102], except instead of randomly selecting a variable in a broken clause, it chooses the one with the highest net gain. Minor changes like these have continuously improved LS solvers for over a decade. The CLASS system tries to automate this reconfiguration and fine tune the process by developing a concise language that can express any existing LS solver. A genetic algorithm then creates solvers that conform to this language. To avoid overly complex solvers, all cases having more than two nested conditionals are automatically collapsed by replacing the problematic sub-tree with a random function of depth 1. The resulting solvers were shown to be competitive with the best existing solvers. The one issue with this approach, however, is that developing such a grammar for other algorithms or problem types can be difficult, if not impossible.

As another example, in [86] Oltean proposed constructing a solver that uses a genetic algorithm (GA) automatically. In this case, the desired solver is modeled as a sequence of the selection, combination and mutation operations of a GA. For a given problem type and collection of training instances, the objective is to find the sequence of these operations that results in the solver requiring the fewest iterations to train. To find this optimal sequence of operations, Oltean proposes using a linear genetic program. The resulting algorithms were shown to outperform the standard implementations of genetic algorithms for a variety of tasks. However, while this approach can be applied to a variety of problem types, it ultimately suffers from requiring a long time to train. To evaluate an iteration of the potential solvers, each GA needs to be run 500 times on all the training instances to determine the best solver in the population accurately. This is fine for rapidly evaluated instances, but once each instance requires more than a couple of seconds to evaluate, the approach becomes too time-consuming.

Algorithm construction has also been applied to create a composite sorting algorithm used by a compiler [71]. The authors observed that there is no single sorting strategy that works perfectly on all possible input instances, with different strategies yielding improved performance on different instances. With this observation, a tree-based encoding was used for a solver that iteratively partitioned the elements of an instance until reaching a single element in the leaf node, and then sorted the elements as the leaves were merged. The primitives defined how the data is partitioned and under what conditions the sorting algorithm should change its approach. For example, the partitioning algorithm employed would depend on the amount of data that needs to be sorted. To make their method instance-specific, the authors use two features encoded as a six-bit string. For training, all instances are

split according to the encodings and each encoding is trained separately. To evaluate the instance, the encoding of the test instance is computed and the algorithm of the nearest and closest match is used for evaluation. This approach has been shown to be better than all existing algorithms at the time, providing a factor two speedup. The issue with the approach, however, is that it only uses two highly disaggregated features to identify the instance and that during training it tries to split the data into all possible settings. This becomes intractable as the number of features grows.

2.2 Instance-Oblivious Tuning

Given a collection of sample instances, instance-oblivious tuning attempts to find the parameters resulting in the best average performance of a solver on all the training data. There are three types of solver parameters. First, parameters can be categorical, controlling decisions such as what restart strategy to use or which branching heuristic to employ. Alternatively, parameters can be ordinal, controlling decisions about the size of the neighborhood for a local search or the size of the tabu list. Finally, parameters can be continuous, defining an algorithm's learning rate or the probability of making a random decision. Due to these differences, the tuning algorithms used to set the parameters can vary wildly. For example, the values of a categorical parameter have little relation to each other, making it impossible to use regression techniques. Similarly, continuous parameters have much larger domains than ordinal parameters. Here we discuss a few of the proposed methods for tuning parameters.

One example of instance-oblivious tuning focuses on setting continuous parameters. Coy et al. [29] suggested that by computing a good parameter set for a few instances, averaging all the parameters will result in parameters that would work well in the general case. Given a training set, this approach first selects a small diverse set of problem instances. The diversity of the set is determined by a few handpicked criteria specific to the problem type being solved. Then analyzing each of these problems separately, the algorithm tests all possible extreme settings of the parameters. After computing the performance at these points, a response surface is fitted, and greedy descent is used to find a locally optimal parameter set for the current problem instance. The parameter sets computed for each instance are finally averaged to return a single parameter set expected to work well on all instances. This technique was empirically shown to improve solvers for set covering and vehicle routing. The approach, however, suffers when more parameters need to be set or if these parameters are not continuous.

For a small set of possible parameter configurations, F-Race [20] employs a racing mechanism. During training, all potential algorithms are raced against each other, whereby a statistical test eliminates inferior algorithms before the remaining algorithms are run on the next training instance. But the problem with this is that F-Race prefers small parameter spaces, as larger ones would require a lot of testing in the primary runs. Careful attention must also be given to how and when certain

parameterizations are deemed pruneable, as this greedy selection is likely to end with a suboptimal configuration.

Alternatively, the CALIBRA system, proposed in [4], starts with a factorial design of the parameters. Once these initial parameter sets have been run and evaluated, an intensifying local search routine starts from a promising design, whereby the range of the parameters is limited according to the results of the initial factorial design experiments.

For derivative-free optimization of continuous variables, [11] introduced a mesh adaptive direct search (MADS) algorithm. In this approach, the parameter search space is partitioned into grids, and the corner points of each grid are evaluated for the best performance. The grids associated with the current lower bound are then further divided and the process is repeated until no improvement can be achieved. One of the additional interesting caveats to the proposed method was to use only short-running instances in the training set to speed up the tuning. It was observed that the parameters found for the easy instances tended to generalize to the harder ones, thus leading to significant improvements over classical configurations.

As another example, a highly parameterized solver like SATenstein [62] was developed, where all the choices guiding the stochastic local search SAT solver were left open as parameters. SATenstein can therefore be configured into any of the existing solvers as well as some completely new configurations. Among the methods used to tune such a solver is ParamILS.

In 2007, ParamILS [56] was first introduced as a generic parameter tuner, able to configure arbitrary algorithms with very large numbers of parameters. The approach conducts focused iterated local search, whereby starting with a random assignment of all the parameters, a local search with a one-exchange neighborhood is performed. The local search continues until a local optimum is encountered, at which point the search is repeated from a new starting point. To avoid randomly searching the configuration space, at each iteration the local search gathers statistics on which parameters are important for finding improved settings, and focuses on assigning them first. This blackbox parameter tuner has been shown to be successful with a variety of solvers, including Cplex [59], SATenstein [62], and SAPS [57], but suffers due to not being very robust, and depending on the parameters being discretized.

As an alternative to ParamILS, in 2009 the gender-based genetic algorithm [5] (GGA) was introduced. This black box tuner conducts a population-based local search to find the best parameter configuration. This approach presented a novel technique of introducing competitive and non-competitive genders to balance exploitation and exploration of the parameter space. Therefore, at each generation, half of the population competes on a collection of training instances. The subset of parameter settings that yield the best overall performance are then mated with the non-competitive population, with the children removing the worst-performing individuals from the competitive population. This approach was shown to be remarkably successful in tuning existing solvers, often outperforming ParamILS.

Recently, a sequential model-based algorithm configuration (SMAC) [55] was introduced, in 2010. This approach proposes generating a model over the solver's

parameters to predict the likely performance. This model can be anything from a random forest to marginal predictors and is used to identify aspects of the parameter space, such as what parameters are the most important. Possible configurations are then generated according to this model and compete against the current incumbent. The best configuration continues onto the next iteration. While this approach has been shown to work on some problems, it ultimately depends on the accuracy of the model used to capture the interrelations of the parameters.

2.3 Instance-Specific Regression

One of the main drawbacks of instance-oblivious tuning is ignoring the specific instances, striving instead for the best average case performance. However, works like [82, 112], and many others have observed that not all instances yield to the same approaches. This observation supports the "no free lunch" theorem [122], which states that no single algorithm can be expected to perform optimally over all instances. Instead, in order to gain improvements in performance for one set of instances, it will have to sacrifice performance on another set. The typical instance-specific tuning algorithm computes a set of features for the training instances and uses regression to fit a model that will determine the solver's strategy.

Algorithm portfolios are a prominent example of this methodology. Given a new instance, the approach forecasts the runtime of each solver and runs the one with the best predicted performance. SATzilla [126] is an example of this approach as applied to SAT. In this case the algorithm uses ridge regression to forecast the log of the run times. Interestingly, for the instances that timeout during training, the authors suggest using the predicted times as the observed truth, a technique they show to be surprisingly effective. In addition, SATzilla uses feedforward selection over the features it uses to classify a SAT instance. It was found that certain features are more effective at predicting the runtimes of randomly generated instances as opposed to industrial instances, and vice-versa. Overall, since its initial introduction in 2007, SATzilla has won medals at the 2007 and 2009 SAT Competitions [1].

In algorithm selection the solver does not necessarily have to stick to the same algorithm once it is chosen. For example, [40] proposed running in parallel (or interleaved on a single processor) multiple stochastic solvers that tackle the same problem. These "algorithm portfolios" were shown to work much more robustly than any of the individual stochastic solvers. This insight has since led to the technique of randomization with restarts, which is commonly used in all state-of-the-art complete SAT solvers. Algorithm selection can also be done dynamically. As was shown in [36], instead of choosing the single best solver from a portfolio, all the solvers are run in parallel. However, rather than allotting equal time to everything, each solver is biased, depending on how quickly the algorithm thinks it will complete. Therefore, a larger time share is given to the algorithm that is assumed to be the first to finish. The advantage of this technique is that it is less susceptible to an early error in the performance prediction.

In [88], a self-tuning approach is presented that chooses parameters based on the input instance for the local search SAT solver WalkSAT. This approach computes an estimate of the invariant ratio of a provided SAT instance, and uses this value to set the noise of the WalkSAT solver, or how frequently a random decision is made. This was shown to be effective on four DIMACS benchmarks, but failed for those problems where the invariant ratio did not relate to the optimal noise parameter.

In another approach, [53, 54] tackle solvers with continuous and ordinal (but not categorical) parameters. Here, Bayesian linear regression is used to learn a mapping from features and parameters into a prediction of runtime. Based on this mapping for given instance features, a parameter set that minimizes predicted runtime is searched for. The approach in [53] led to a twofold speedup for the local search SAT solver SAPS [57].

An alternative example expands on the ideas introduced in SATzilla by presenting Hydra [124]. Instead of using a set of existing solvers, this approach uses a single highly parameterized solver. Given a collection of training instances, a set of different configurations is produced to act as the algorithm portfolio. Instances that are not performing well under the current portfolio are then identified and used as the training set for a new parameter configuration that is to be added to the portfolio. Alternatively, if a configuration is found not to be useful any longer, it is removed from the portfolio. A key ingredient to making this type of system work is the provided performance metric, which uses a candidate's actual performance when it is best and the overall portfolio's performance otherwise. This way, a candidate configuration is not penalized for aggressively tuning for a small subset of instances. Instead, it is rewarded for finding the best configurations and thus improving overall performance.

An alternative to regression-based approaches for instance-specific tuning, CPHydra [87] attempts to schedule solvers to maximize the probability of solving an instance within the allotted time. Given a set of training instances and a set of available solvers, CPHydra collects information on the performance of every solver on every instance. When a new instance needs to be solved, its features are computed and the k nearest neighbors are selected from the training set. The problem then is set as a constraint program that tries to find the sequence and duration in which to invoke the solvers so as to yield the highest probability of solving the instance. The effectiveness of the approach was demonstrated when CPHydra won the CSP Solver Competition in 2008, but also showed the difficulties of the approach since the dynamic scheduling program only used three solvers and a neighborhood of 10 instances.

Most recently, a new version of SATzilla was entered into the 2012 SAT Competition [127]. Foregoing the original regression-based methodology, this solver trained a tree classifier for predicting the preferred choice for each pair of solvers in the portfolio. Therefore when a new instance had to be addressed, the solver that was chosen most frequently was the one that got evaluated. In practice this worked very well, with the new version of SATzilla winning gold in each of the three categories. Yet this approach is also restricted to a very small portfolio of

solvers, as each addition to the portfolio requires exponentially many new classifiers to be trained.

2.4 Adaptive Methods

All of the works presented so far were trained offline before being applied to a set of test instances. Alternative approaches exist that try to adapt to the problem they are solving in an online fashion. In this scenario, as a solver attempts to solve the given instance, it learns information about the underlying structure of the problem space, trying to exploit this information in order to boost performance.

Algorithm selection is closely related to the algorithm configuration scenario that is tuning one categorical variable. For example, in [73] a sampling technique selects one of several different branching variable selection heuristics in a branch-and-bound approach. A similar approach was later presented in [64], but instead of choosing a single heuristic, this approach tries to learn the best branching heuristic to use at each node of a complete tree search.

An example of this technique is STAGE [21], an adaptive local search solver. While searching for a local optima, STAGE learned an evaluation function to predict the performance of a local search algorithm. At each restart, the solver would predict which local search algorithm was likely to find an improving solution. This evaluation function was therefore used to bias the trajectory of the future search. The technique was empirically shown to improve the performance of local search solvers on a variety of large optimization problems.

Impact-based search strategies for constraint programming (CP) [96] are another example of a successful adaptive approach. In this work, the algorithm would keep track of the domain reduction of each variable after the assignment of a variable. Assuming that we want to reduce the domains of the variables quickly and thus shrink the search space, this information about the impact of each variable guides the variable selection heuristic. The empirical results were so successful that this technique is now standard for Ilog CP Solver, and used by many other prominent solvers, like MiniSAT [33].

In 1994, an adaptive technique was proposed for tabu search [13]. By observing the average size of the encountered cycles, and how often the search returned to a previous state, this algorithm dynamically modified the size of its tabu list.

Another interesting result for transferring learned information between restarts was presented in Disco–Novo–GoGo [101]. In this case, the proposed algorithm uses a value-ordering heuristic while performing a complete tree search with restarts. Before a restart takes place, the algorithm observes the last tried assignment and changes the value ordering heuristic to prefer the currently assigned value. In this way, the search is more likely to explore a new and more promising portion of the search space after the restart. When applied to constraint programming and satisfiability problems, orders of magnitude performance gains were observed.

2.5 Chapter Summary

In this chapter, related work for automatic algorithm configuration was discussed. The first approach of automatic algorithm construction focused on how solving strategies and heuristics can be automatically combined to result in a functional solver by defining the solver's structure. Alternatively, given that a solver is created where all the controlling parameters are left open to the user, the instance-oblivious methodology finds the parameter settings that result in the best average-case performance. When a solver needs to be created to perform differently depending on the problem instance, instance-specific regression is often employed to find an association between the features of the instance and the desired parameter settings. Finally, to avoid extensive offline training on a set of representative instances, adaptive methods that adapt to the problem dynamically are also heavily researched.

All these techniques have been shown empirically to provide significant improvements in the quality of the tuned solver. Each approach, however, also has a few general drawbacks. Algorithm construction depends heavily on the development of an accurate model of the desired solver; however, for many cases, a single model that can encompass all possibilities is not available. Instance-oblivious tuning assumes that all problem instances can be solved optimally by the same algorithm, an assumption that has been frequently shown impossible in practice. Instance-specific regression, on the other hand, depends on accurately fitting a model from the features to a parameter, which is intractable and requires a lot of training data when the features and parameters have non-linear interactions. Later developments with trees alleviate the issues presented by regression, but existing approaches don't tend to scale well with the size of the portfolio. Adaptive methods require a high overhead since they need to spend time exploring and learning about the problem instance while attempting to solve it. The remainder of this book focuses on how instance-oblivious tuning can be extended to create a modular and configurable framework that is instance-specific.

Chapter 3
Instance-Specific Algorithm Configuration

Instance-Specific Algorithm Configuration, ISAC, the proposed approach, takes advantage of the strengths of two existing techniques, instance-oblivious tuning and instance-specific regression, while mitigating their weaknesses. Specifically, ISAC combines the two techniques to create a portfolio where each solver is tuned to tackle a specific type of problem instance in the training set. This is achieved using the assumption that problem instances can be accurately represented by a finite number of features. Furthermore, it is assumed that instances that have similar features can be solved optimally by the same solver. Therefore, given a training set, the features of each instance are computed and used to cluster the instances into distinct groups. The ultimate goal of the clustering step is to bring instances together that prefer to be solved by the same solver. An automatic parameter tuner then finds the best parameters for the solver of each cluster. Given a new instance, its features are computed and used to assign it to the appropriate cluster, where it is evaluated with the solver tuned for that particular cluster.

This three-step approach is versatile and applicable to a number of problem types. Furthermore, the approach is independent of the precise algorithms employed for each step. This chapter first presents two clustering approaches that can be used, highlighting the strengths and weaknesses of each. Additional clustering approaches will be discussed in Chap. 10. The chapter then presents the two methods of tuning the solver. Due to its problem-specific nature, the feature computation will be presented in Chap. 4.

Y. Malitsky, *Instance-Specific Algorithm Configuration*,
DOI 10.1007/978-3-319-11230-5_3

3.1 Clustering the Instances

There are many clustering techniques available in recent research [14]. This section, however, first shows how to define the distance metric, which is important regardless of the clustering method employed. It then presents the two clustering approaches initially tested for ISAC.

3.1.1 Motivation

One of the underlying assumptions behind ISAC is that there are groups of similar instances, all of which can be solved efficiently by the same solver. Here we further postulate that these similarities can be identified automatically. Figure 3.1 highlights the validity of these assumptions. The figures are based on the standard 48 SAT features (which will be introduced in detail in Chap. 4) for 3,117 instances from the 2002 to 2012 SAT Competitions [1]. The features were then normalized, and using PCA, projected onto two dimensions. We ran 29 solvers available in 2012 with a 5,000 s timeout and recorded the best possible time for each instance. Figure 3.1 shows the performance of two of these solvers (CCASat [27] and lingering [19]). In the figure, an instance is marked as a green cross if the runtime of the solver on this instance was no worse than 25 % more time than the best recorded time for that instance. All other instances are marked with a black circle unless the solver timed out, in which case it is a red triangle.

What is interesting to note here is that there is a clear separation between the instances where the solvers do not timeout. This is likely attributed to the fact that CCASat was designed to solve randomly generated instances, while lingering is better at industrial instances. Therefore it is no surprise that in the instances where one solver does well, the other is likely to timeout. What is also interesting is that the instances where either of the solvers does not timeout appear to be relatively well clustered. This complementary and cluster-like behavior is also evident for the other 27 solvers, and is the motivation behind embracing a cluster-based approach.

3.1.2 Distance Metric

The quality of a clustering algorithm strongly depends on how the distance metric is defined in the feature space. Features are not necessarily independent. Furthermore, important features can range between small values while features with larger ranges could be less important. Finally, some features can be noisy, or worse, completely useless and misleading. For the current version of ISAC, however, it is assumed that the features are independent and not noisy. Chapter 10 will show how to handle situations where this is not the case.

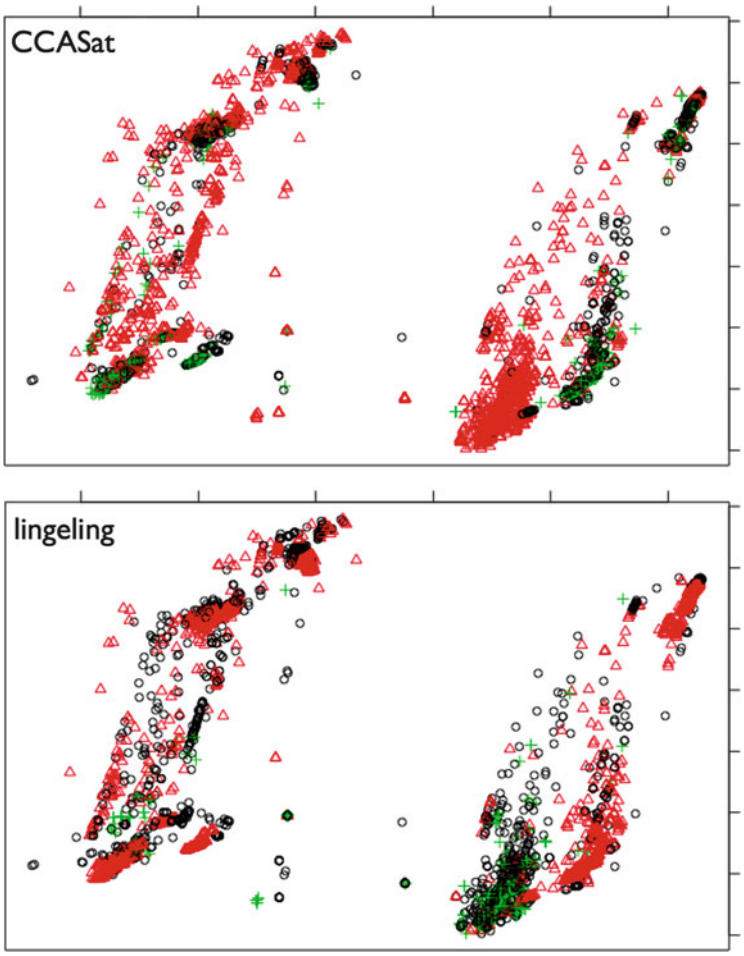

Fig. 3.1 Performance of CCASat and lingering on 3,117 SAT instances. A feature vector was computed for each instance and then projected onto 2D using PCA. *Green* crosses mark good instances, which perform no worse than 25 % slower than the best solver on that instance. An ok instance (*black circle*) is one that is more than 25 % worse than the best solver. An instance that takes more than 5,000 s is marked as a timeout (*red triangle*)

A weighted Euclidean distance metric can handle the case where not all features are equally important to a proper clustering. This metric also handles the case where the ranges of the features vary wildly. To automatically set the weights for the metric an iterative approach is needed. Here all the weights can be first set to 1 and the training instances clustered accordingly. Once the solvers have been tuned for each cluster, the quality of the clusters is evaluated. To this end, for each pair of clusters $i \neq j$, the difference is computed between the performance on all instances in cluster i that is achieved by the solver for that cluster and the solver of the other

cluster. The distance between an instance a in cluster C_i and the centers of gravity of cluster C_j is then the maximum of this regret and 0. Using these desired distances, the feature metric is adjusted and the process continues to iterate until the feature metric stops changing.

This iterative approach works well when improving a deterministic value like the solution quality, where it is possible to perfectly assess algorithm performance. The situation changes when the objective is to minimize runtime. This is because parameter sets that are not well suited for an instance are likely to run for a very long time, necessitating the introduction of a timeout. This then implies that the real performance is not always known, and all that can be used is the lower bound. This complicates learning a new metric for the feature space. In the experiments, for example, it was found that most instances from one cluster timed out when run with the parameters of another. This not only leads to poor feature metrics, but also costs a lot in terms of processing time. Furthermore, because runtime is often a noisy measurement, it is possible to encounter a situation where instances oscillate between two equally good clusters. Finally, this approach is very computationally expensive, requiring several retuning iterations which can take CPU days or even weeks for each iteration.

Consequently, for the purpose of tuning the speed of general solvers, this chapter suggests a different approach. Instead of learning a feature metric over several iterations, the features are normalized using translation and scaling so that, over the set of training instances, each feature spans exactly the interval $[-1, 1]$. That is, for each feature there exists at least one instance for which this feature has value 1 and at least one instance where the feature value is -1. For all other instances, the value lies between these two extremes. By normalizing the features in this manner, it is found that features with large and small ranges are given equal consideration during clustering. Furthermore, the assumption that there are no noisy or bad features does not result in bad clusterings. However, Chap. 10 shows how filtering can be applied to further improve performance.

3.1.3 k-Means

One of the most straightforward clustering algorithms is Lloyd's k-means [72]. As can be seen in Algorithm 1, the algorithm first selects k random points in the feature space. It then alternates between two steps until some termination criterion is reached. The first step assigns each instance to a cluster according to the shortest distance to one of the k points that were chosen. The next step then updates the k points to the centers of the current clusters.

While this clustering approach is very intuitive and easy to implement, the problem with k-means clustering is that it requires the user to specify the number of clusters k explicitly. If k is too low, this means that some of the potential is lost for tuning parameters more precisely for different parts of the instance feature space. On the other hand, if there are too many clusters, the robustness and generality of

Algorithm 1: k-means clustering algorithm

k-Means(X, k)
Choose k random points $C_1, \ldots, C_k$ from X.
while not done **do**
  for $i = 1, \ldots, k$ **do**
    $S_i \leftarrow \{j : ||X_j - C_i|| \leq ||X_j - C_l|| \; \forall \; l = 1, \ldots, k\}$
    $C_i \leftarrow \frac{1}{|S_i|} \sum_{j \in S_i} X_j$
  end for
end while
return (C, S)

Algorithm 2: g-means clustering algorithm

g-Means(X)
$k \leftarrow 1, i \leftarrow 1$
$(C, S) \leftarrow k$-Means(X, k)
while $i \leq k$ **do**
  $(\bar{C}, \bar{S}) \leftarrow k$-Means$(S_i, 2)$
  $v \leftarrow \bar{C}_1 - \bar{C}_2, w \leftarrow \sum v_i^2$
  $y_i \leftarrow \sum v_i x_i / w$
  if Anderson-Darling-Test(y) failed **then**
    $C_i \leftarrow \bar{C}_1, S_i \leftarrow \bar{S}_1$
    $k \leftarrow k + 1$
    $C_k \leftarrow \bar{C}_2, S_k \leftarrow \bar{S}_2$
  else
    $i \leftarrow i + 1$
  end if
end while
return (C, S, k)

the parameter sets that are optimized for these clusters are sacrificed. Furthermore, for most training sets, it is unreasonable to assume that the value of k is known.

3.1.4 g-Means

In 2003, Hamerly and Elkan proposed an extension to k-means that automatically determines the number of clusters [44]. This work proposes that a good cluster exhibits a Gaussian distribution around the cluster center. The algorithm, presented in Algorithm 2, first considers all inputs as forming one large cluster. In each iteration, one of the current clusters is picked and is assessed for whether it is already sufficiently Gaussian. To this end, g-means splits the cluster into two by running 2-means clustering. All points in the cluster can then be projected onto the line that runs through the centers of the two sub-clusters, giving a one-dimensional distribution of points. g-means now checks whether this distribution is normal using the widely accepted Anderson–Darling statistical test. If the current cluster does not

pass the test, it is split into the two previously computed clusters, and the process is continued with the next cluster.

It was found that the g-means algorithm works very well for our purposes, except sometimes clusters can be very small, containing very few instances. To obtain robust parameter sets we do not allow clusters that contain fewer than a manually chosen threshold, a value which depends on the size of the dataset. Beginning with the smallest cluster, the corresponding instances are redistributed to the nearest clusters, where proximity is measured by the Euclidean distance of each instance to the cluster's center.

3.2 Training Solvers

Once the training instances are separated into clusters, the parameterized solver must be tuned for each cluster. As shown in existing research, manual tuning is a complex and laborious process that usually results in subpar performance of the solver. Chapter 2 shows that there is growing number of methods for tackling this problem, ParamILS [56] and SMAC [55] being notable examples. The subsequent chapters, however, will utilize the two algorithms presented here.

3.2.1 Local Search

With automatic parameter tuning being a relatively new field, there are not many off-the-shelf tuners available. Furthermore, some problems seem to be outside the scope of existing tuners, requiring the development of problem-specific tuners. One such scenario is when the parameters of the solvers are a probability distribution; where the parameters are continuous variables between 0 and 1 and sum up to 1. For this kind of problem we developed [76], a local search shown in Algorithm 3.

This search strategy is presented with an algorithm A for a combinatorial problem as well as a set S of training instances. Upon termination, the procedure returns a probability distribution for the given algorithm and benchmark set.

The problem of computing this favorable probability distribution can be stated as a continuous optimization problem: $\text{Minimize}_{\text{distr}} \sum_{i \in S} \text{Perf}(\text{A},\text{distr},i)$ such that "distr" is a probability distribution used by A. Each variable of the distribution is initialized randomly and then normalized so that all variables sum up to 1. In each iteration, two variables a, b are picked randomly and their joint probability mass m is redistributed among themselves while keeping the probabilities of all other advisors the same.

It is expected that the one-dimensional problem which optimizes the percentage of m assigned to advisor a (the remaining percentage is determined to go to advisor b) is convex. The search seeks the best percentage using a method for minimizing one-dimensional convex functions over closed intervals that is based on the golden

Algorithm 3: Local search for tuning variables that are part of a probability distribution.

LSTuner(Algorithm A, BenchmarkSet S)
distr ← RandDistr()
$\lambda_l \leftarrow \frac{\sqrt{5}-1}{\sqrt{5}+1}$, $\lambda_r \leftarrow \frac{2}{\sqrt{5}+1}$
while termination criterion not met **do**
  $(a, b) \leftarrow$ ChooseRandPair(), m ← $\text{distr}_a + \text{distr}_b$
  $X \leftarrow \lambda_l$, $Y \leftarrow \lambda_r$
  $L \leftarrow 0$, $R \leftarrow 1$, length ← 1
  $p_X \leftarrow \sum_{i \in S}$ Perf(A,distr[a=m X,b=m $(1-X)$], i)
  $p_Y \leftarrow \sum_{i \in S}$ Perf(A,distr[a=m Y,b=m $(1-Y)$], i)
  while length $> \epsilon$ **do**
    if $p_X < p_Y$ **then**
      $p_Y \leftarrow p_X$
      $R \leftarrow Y$, length ← $R - L$
      $Y \leftarrow X$, $X \leftarrow L + \lambda_l$ length
      $p_X \leftarrow \sum_{i \in S}$ Perf(A,distr[a=m X,b=m $(1-X)$], i)
    else
      $p_X \leftarrow p_Y$
      $L \leftarrow X$, length ← $R - L$
      $X \leftarrow Y$, $Y \leftarrow L + \lambda_r$ length
      $p_Y \leftarrow \sum_{i \in S}$ Perf(A,distr[a=m Y,b=m $(1-Y)$], i)
    end if
  end while
  distr ← distr[a=m X,b=m $(1-X)$]
end while
return distr

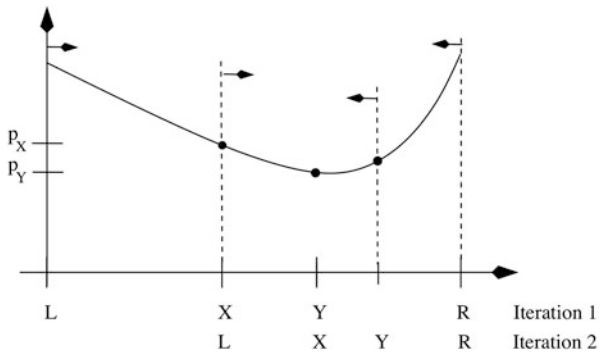

Fig. 3.2 Minimizing a one-dimensional convex function by golden section

section (see Fig. 3.2): two points $X < Y$ are considered within the interval $[0, 1]$ and their performance is measured as "p_X" and "p_Y". The performance at X is assessed by running the algorithm A on the given benchmark with distribution "distr [a=m X,b=m $(1-X)$]", which denotes the distribution resulting from "distr" when assigning probability mass 'Xm' to variable a and probability mass '$(1-X)$m' to

variable "b". Now, if the function is indeed convex, if $p_X < p_Y$ ($p_X \geq p_Y$), then the minimum of this one-dimensional function lies in the interval $[0, Y]$ ($[X, 1]$). The search continues splitting the remaining interval (which shrinks geometrically fast) until the interval size "length" falls below a given threshold "ϵ". By choosing points X and Y based on the golden section, we need in each iteration only one new point to be evaluated rather than two. Moreover, the points considered in each iteration are reasonably far apart from each other to make a comparison meaningful, which is important for us as our function evaluation may be noisy (due to the randomness of the algorithm invoked) and points very close to each other will likely produce very similar results.

3.2.2 GGA

One drawback to developing proprietary tuning algorithms is the difficulty of transferring the technique across problem types. To test a more general procedure, the Gender-Based Genetic Algorithm (GGA [5]), a state-of-the-art automatic parameter tuner, is explored. This tuner uses a genetic algorithm to find the parameters for a specified solver. Representing the parameters in an and–or tree, the tuner randomly generates two populations of possible parameter configurations. These two groups are classified as being *competitive* or *non-competitive*. A random subset of the individuals from the competitive population are selected and run against each other over a subset of the training instances. This tournament is repeated several times until all members of the competitive population participated in exactly one tournament. Each member of the non-competitive competition is mated with one of the tournament winners. This process is repeated for 100 iterations, when the best parameter setting is returned as the parameter set to be used by the solver.

In this scenario the parameters of the tuned solver are represented as an and–or tree (Fig. 3.3). This representation allows the user to specify the relation between the parameters. For example, parameters that are independent are separated by an *and*

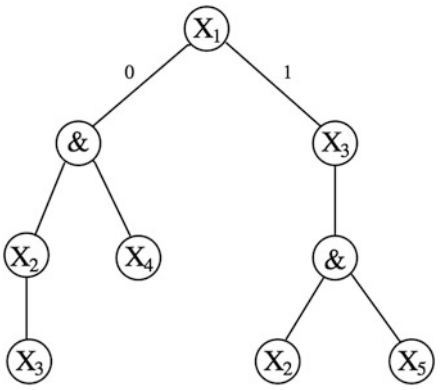

Fig. 3.3 And–or tree used by GGA representing the parameters of the tuned algorithm

parent. On the other hand, if a parameter depends on the setting of another parameter it is defined as a child of that parameter. This representation allows GGA to better search the parameter space by maintaining certain parameter settings constant as a group instead of randomly changing different parameters.

Each mating of a couple results in one new individual with a random gender. The genome of the offspring is determined by traversing the variable tree top-down. A node can be labelled O ("open"), C ("competitive"), or N ("non-competitive"). If the root is an and node, or if both parents agree on the value of the root-variable, it is labeled O. Otherwise, the node is labeled randomly as C or N. The algorithm continues by looking at the children of the root (and so on for each new node). If the label of the parent node is C (or N) then with high probability P% the child is also labeled C (N); otherwise the label is switched. By default P is set to 90 %.

Finally, the variable assignment associated with the offspring is given by the values from the C (N) parent for all nodes labelled C (N). For variable nodes labelled O, both parents agree on its value, and this value is assigned to the variable. Note that this procedure effectively combines a uniform crossover for child variables of open and-nodes in the variable tree (thus exploiting the independence of different parts of the genome) and a randomized multiple-point crossover for variables that are more tightly connected.

As a final step to determine the offspring's genome, each variable is mutated with low probability M %. By default M is set to 10 %. When mutating a categorical variable, the new value in the domain is chosen uniformly at random. For continuous and integer variables, the new value is chosen according to a Gaussian distribution where the current value marks the expected value and the variance is set as 10 % of the variable's domain.

3.3 ISAC

Ultimately the ISAC methodology is summarized in Algorithm 4, where the application of all three components is displayed, particularly, a parameterized algorithm A, a list of training instances T, and their corresponding feature vectors F. First, the features in the set are normalized and the scaling and translation values are memorized for each feature (s, t).

Then, an algorithm is used to cluster the training instances based on the normalized feature vectors. The final result of the clustering is a number of k clusters S_i, a list of cluster centers C_i, and, for each cluster, a distance threshold d_i which determines when a new instance will be considered as close enough to the cluster center to be solved with the parameters computed for instances in this cluster.

Then, for each cluster of instances S_i, favorable parameters P_i are computed using an instance-oblivious tuning algorithm. After this is done, the parameter set R is computed for all the training instances. This serves as the recourse for all future instances that are not near any of the clusters.

Algorithm 4: Instance-Specific Algorithm Configuration

ISAC-Learn(A, T, F)
$(\bar{F}, s, t) \leftarrow$ Normalize(F)
$(k, C, S, d) \leftarrow$ Cluster $(T, \bar{F})$
for all $i = 1, \ldots, k$ **do**
  $P_i \leftarrow Train(A, S_i)$
end for
$R \leftarrow Train(A, T)$
return (k, P, C, d, s, t, R)

ISAC-Run$(A, x, k, P, C, d, s, t, R)$
$\bar{f} \leftarrow$ Features(x)
$\bar{f}_i \leftarrow (f_i - t_i)/s_i \ \forall\ i$
for all $j = 1, \ldots, k$ **do**
  if $||\bar{f} - C_i|| \leq d_i$ **then**
    return $A(x, P_i)$
  end if
end for
return $A(x, R)$

When running algorithm A on an input instance x, we first compute the features of the input and normalize them using the previously stored scaling and translation values for each feature. Then, we determine whether there is a cluster such that the normalized feature vector of the input is close enough to the cluster center. If so, A is run on x using the parameters for this cluster. If the input is not near enough to any of our clusters, the instance-oblivious parameters R are used, which work well for the entire training set.

3.4 Chapter Summary

This chapter presents the components of the proposed instance-specific automatic parameter tuner, ISAC. The approach partitions the problem of automatic algorithm configuration into three distinct pieces. First, the feature values are computed for each instance. Second, the training instances are clustered into groups of instances that have similar features. Finally, an automatic parameter tuner is used to find the best parameters for the solver for each cluster. This chapter shows several basic configurations of the last two steps of ISAC. Being problem-specific, the features used for clustering are explained in the numerical section of the subsequent chapters.

Chapter 4
Training Parameterized Solvers

This chapter details the numerical results of applying ISAC on four different types of combinatorial optimization problems. The first section covers the set covering problem (SCP), showing that instance-oblivious tuning of the parameters can yield significant performance improvements and that ISAC can perform better than an instance-specific regression approach. The second section presents the mixed integer problem (MIP) and shows that even a state-of-the-art solver like Cplex can be improved through instance-specific tuning. The third section introduces the satisfiability problem (SAT) and shows how an algorithm portfolio can be enhanced by the proposed approach. The fourth example presents a real-world application from the 2012 Roadef Challenge. The chapter concludes with a brief summary of the results and benefits of the ISAC approach.

Unless otherwise noted, experiments were run on a dual-processor, dual-core Intel Xeon 2.8 GHz computer with 8 GB of RAM. SCP solvers Hegel and Nysret were evaluated on a quad-core dual-processor Intel Xeon 2.53 Ghz processors with 24 GB of RAM.

4.1 Set Covering Problem

The empirical evaluation begins with one of the most studied combinatorial optimization problems: the set covering problem (SCP). In SCP, given a finite set $S := \{1, \ldots, n\}$ of items, a family $F := \{S_1, \ldots, S_m \subseteq S\}$ of subsets of S, and a cost function $c : F \rightarrow \mathbb{R}^+$, the objective is to find a subset $C \subseteq F$ such that $S \subseteq \bigcup_{S_i \in C} S_i$ and $\sum_{S_i \in C} c(S_i)$ is minimized. In the *unicost* SCP, the cost of each set is set to 1. Plainly put, there are a number of sets that each have a certain variety of items, like different candy bars in bags handed out at a party. Each set has an associated cost depending on the items it contains. The objective is to acquire sets, at a minimal cost, such that there is at least one copy of each item present.

Y. Malitsky, *Instance-Specific Algorithm Configuration*,
DOI 10.1007/978-3-319-11230-5_4

This problem formulation appears in numerous practical applications such as crew scheduling [28, 48, 50], location of emergency facilities [115], and production planning in various industries [117].

In accordance with ISAC, the first step deals with the identification of a set of features that accurately distinguish the problem instances. Following the process introduced in [76], the features are generated by computing the maxima, minima, averages, and standard deviations of the following vectors:

- vector of normalized subset costs $c' \in [1, 100]^m$,
- vector of subset densities $(|S_i|/n)_{i=1...m}$,
- vector of item costs $(\sum_{i \mid j \in S_i} c'_i)_{j=1...n}$,
- vector of item coverings $(|\{i \mid j \in S_i\}|/m)_{j=1...n}$,
- vector of costs over density $(c'_i/|S_i|)_{i=1...m}$,
- vector of costs over square density $(c'_i/|S_i|^2)_{i=1...m}$,
- vector of costs over $k \log k$-density $(\frac{c'_i}{(|S_i| \log |S_i|)})_{i=1...m}$, and
- vector of root-costs over square density $(\sqrt{c'_i}/|S_i|^2)_{i=1...m}$.

Computation of these feature values on average takes only 0.01 s per instance.

Due to a sparsity of well-established benchmarks for set covering problems, a new highly diverse set of instances is generated. Specifically, a large collection of instances is randomly generated, each comprised of 100 items and 10,000 subsets. To generate these instances, an SCP problem is considered as a binary matrix where each column represents an item and each row represents a subset. A 1 in this matrix corresponds to an item being included in the subset. The instance generator then randomly makes three decisions. One, to fill the matrix by either row or column. Two, if the density (ratio of 1s to 0s) of the row or column is constant, has a mean of 4 %, or has a mean of 8 %. Three, whether the cells to be set to 1 are chosen uniformly at random or with a Gaussian bias centered around some cell. The cost of each subset is chosen uniformly at random from [1, 1000]. For the unicost experiments, all the subset costs are reset to 1. The final dataset comprises 200 training instances and 200 test instances.

4.1.1 Solvers

Due to the popularity of SCP, a rich and diverse collection of algorithms was developed to tackle this problem. To analyze the effectiveness and applicability of the approach, three orthogonal approaches are focused on that cover the spectrum of incomplete algorithms. Relying on local search strategies, these algorithms are not guaranteed to return optimal solutions.

The first algorithm is the greedy randomized set covering solver from [76]. This approach repeatedly adds subsets one at a time until reaching a feasible solution. Which subset to add next is determined by one of the following six heuristics,

chosen randomly during the construction of the cover:

- The subset that costs the least ($\min c$).
- The subset that covers the most new items ($\max k$).
- The subset that minimizes the ratio of costs to the number of newly covered items ($\min c/k$).
- The subset that minimizes the ratio of costs to newly covered items times the logarithm of newly covered items ($\min \frac{c}{k \log k}$).
- The subset that minimizes the ratio of costs to the square of newly covered items ($\min \frac{c}{k^2}$).
- The subset that minimizes the ratio of square root of costs to the square of newly covered items ($\min \frac{\sqrt{c}}{k^2}$).

The second solver, "Hegel" [61], uses a specialized type of local search called dialectic search. Designed to optimally balance exploration and exploitation, dialectic search begins with two greedily obtained feasible solutions called the "thesis" and "antithesis" respectively. Then a greedy walk traverses from the thesis to the antithesis, first removing all subsets from the solution of the thesis that are not in the antithesis and then greedily adding the subsets from the antithesis that minimize the overall cost. The Hegel approach was shown to outperform the fastest algorithms on a range of SCP benchmarks.

The third solver, Nysret [83], uses an alternate type of local search algorithm called tabu search. A greedily obtained feasible solution defines the initial state. For each consequent step, the neighborhood is composed of all states obtained by adding or removing one subset from the current solution. The fitness function is then evaluated as the cumulative cost of all the included subsets plus the number of the uncovered items. The neighbor with the lowest cost is chosen as the starting state for the next iteration. During this local search, the subsets that are included or removed are kept track of in the tabu list for a limited number of iterations. To prevent cycles in the local search, neighbors that change the status of a subset in the tabu list are excluded from consideration. In 2006, Nysret was shown empirically to be the best solver for unicost SCP.

4.1.2 Numerical Results

This section presents three results. First it compares the performance of the instance-specific tuning approach of multinomial regression to that of an instance-oblivious parameter tuning. Showing the strength of parameter tuning, two configurations of ISAC are then presented, and compared to the instance-oblivious tuning approach. The section concludes by showing that the ISAC approach can be applied out of the box to two state-of-the-art solvers and results in significant improvements in performance.

Table 4.1 Comparison of the default assignment of the greedy randomized solver (GRS) with parameters found by the instance-specific multinomial regression tuning approach and an instance-oblivious parameter tuning approach

	Optimality gap closed (%)	
Approach	Train	Test
Untuned GRS solver	25.9 (4.2)	40.0 (4.1)
GRS with instance-specific regression	32.8 (3.6)	38.1 (3.7)
GRS with instance-oblivious tuning	**40.0 (3.6)**	**46.1 (3.8)**

The table shows the percentage of the optimality gap closed over using a single best heuristic. The standard deviation is presented in parentheses

Table 4.1 compares the effectiveness of parameter tuning to that of the classical multinomial regression approach. The experiment is performed on the greedy randomized solver, where the parameters are defined as the probabilities of each heuristic being chosen. It was found that using only one heuristic during a greedy search leaves, on average, a 7.2 % (7.6 %) optimality gap on the training (testing) data [76]. A default assignment of equal probabilities to all heuristics can close up to 40 % of this gap on the test instances. For the multinomial regression approach, the algorithm learns a function for each parameter that converts the instance feature vector to a single value, called score. These scores are then normalized to sum to 1 to create a valid probability distribution. However, while this approach leads to some improvements on the training set, the learned functions are not able to carry over to the test set, closing only 38.1 % of the optimality gap. Training on all the instances simultaneously, using a state-of-the-art parameter tuner like GGA leads to superior performance on both training and test sets. This result emphasizes the effectiveness of multi-instance parameter tuners over instance-specific regression models.

As stated in the previous chapter, straight application of a parameter tuner ignores the diversity of instances that might exist in the training and test sets. ISAC addresses this issue by introducing a cluster-based training approach consisting of three steps: computation of features, clustering of training instances, and cluster-based parameter tuning. The first configuration of ISAC [76] uses a weighted Euclidean distance for the features, k-means for clustering, and a proprietary local search for parameter tuning. To set the distance metric weights, this configuration iterated the clustering and tuning steps, trying to minimize the number of training instances yielding better performance for solvers tuned on another cluster. The next configuration [60] built off the first attempt, streamlining each part of the ISAC procedure. As a result, this configuration normalized the features, using g-means for clustering, and GGA for parameter tuning.

As can be seen in Table 4.2, both configurations of ISAC improve on the performance of a solver tuned on all instances. This highlights the benefit of clustering the instances before training. Furthermore, while the numerical results of both configurations are relatively similar it is important that the second is much more efficient and more general of the two. The first configuration was designed specifically to tune the greedy SCP solver and required multiple tuning iterations

Table 4.2 Comparison of two versions of ISAC to an instance-oblivious parameter tuning approach

Approach		Optimality gap closed (%)	
		Train	Test
GRS with instance-oblivious tuning		40.0 (3.6)	46.1 (3.8)
GRS with ISAC	Configuration 1	**47.7 (2.4)**	50.3 (3.7)
	Configuration 2	44.4 (3.3)	**51.3 (3.8)**

The table shows the percent of the optimality gap closed over a greedy solver that only uses the single best heuristic throughout the construction of the solution. The standard deviation is presented in parentheses

Table 4.3 Comparison of default parameters, instance-oblivious parameters provided by GGA, and instance-specific parameters provided by ISAC for Hegel and Nysret

Solver		Avg. run time		Geo. avg.		Avg. slow down	
		Train	Test	Train	Test	Train	Test
Nysret	Default	2.79	3.45	2.36	2.60	1.49	1.79
	GGA	2.58	3.40	2.27	2.63	1.35	1.72
	ISAC	**1.99**	**2.04**	**1.96**	**1.97**	**1.00**	**1.00**
Hegel	Default	3.04	3.15	2.52	2.49	2.20	2.03
	GGA	1.58	1.95	**1.23**	**1.33**	1.10	1.15
	ISAC	**1.45**	**1.92**	**1.23**	1.36	**1.00**	**1.00**

The table presents the arithmetic and geometric mean runtimes in seconds, as well as the average degradation when comparing each solver to ISAC

to achieve the observed result. The second configuration on the other hand uses out of the box tools that can be easily adapted to any solver. It also only requires one clustering and tuning iteration, which makes it much faster than the first. Because of its versatility, unless otherwise stated, all further comparisons to ISAC refer to the second configuration.

To explore the ISAC approach, we next evaluate it on two state-of-the-art local search SCP solvers, Hegel and Nysret. For both solvers the time to find a set covering solution that is within 10 % of optimal is measured. Hegel and Nysret had a timeout of 10 s during training and testing. Table 4.3 compares the default configuration of the solvers, the instance-oblivious configuration obtained by GGA, and the instance-specific tuned versions found by ISAC. To provide a more holistic view of ISAC's performance, three evaluation metrics are presented: the arithmetic and geometric means of the runtime in seconds and the average slow down (the arithmetic mean of the ratios of the performance of the competing solver to that of ISAC). For these experiments the size of the smallest cluster is set to be at least 30 instances. This setting results in four clusters of roughly equal size.

The first experiments show that the default configuration of both solvers can be improved significantly by automatic parameter tuning. For the Nysret solver, an arithmetic mean runtime of 2.18 s for ISAC-Nysret, 3.33 s for GGA-Nysret, and 3.44 s for the default version are measured. That is, instance-oblivious parameters

run 50 % slower than instance-specific parameters. For Hegel, it is found that the default version runs more than 60 % slower than ISAC-Hegel.

It is worth noting the high variance of runtimes from one instance to another, which is caused by the diversity of our benchmark. For a better understanding, the average slow down of each solver compared with that of the corresponding ISAC version is provided. For this measure we find that, for an average test instance, default Nysret requires more than 1.70 times the time of ISAC-Nysret, and GGA-Nysret needs 1.62 times that of ISAC-Nysret. For default Hegel, an average test instance takes 2.10 times the time of ISAC-Hegel while GGA-Hegel only runs 10 % slower. This confirms the findings in [61] that Hegel runs robustly over different instance classes with one set of parameters.

It is concluded that even advanced, state-of-the-art solvers can greatly benefit from ISAC. Depending on the solver, the proposed method works as well as or significantly better than instance-oblivious tuning. Note that this is not self-evident since the instance-specific approach runs the risk of over-tuning by considering fewer instances per cluster. In these experiments, these problems are not observed. Instead it is found that the instance-specific algorithm configurator offers the potential for great performance gains without over-fitting the training data.

4.2 Mixed Integer Programming

For NP-hard problems, mixed integer programming (MIP) involves optimizing a linear objective function while obeying a collection of linear inequalities and variable integrality constraints. Mixed integer programming is an area of great importance in operations research as it can be used to model just about any discrete optimization problem. It is used especially heavily to solve problems in transportation and manufacturing: airline crew scheduling, production planning, vehicle routing, etc.

For the feature computation we use the information about the objective vector, the right-hand side (RHS) vector, and the constraint matrix to formulate the feature vector. The following general statistics on the variables in the problem are computed:

- number of variables and number of constraints,
- percentage of binary (integer or continuous) variables,
- percentage of variables (all, integer, or continuous) with non-zero coefficients in the objective function, and
- percentage of $\leq$ ($\geq$ or $=$) constraints.

Additionally, the mean, min, max, and standard deviation of the following vectors are also used, where $U = Z \cup R$, $R = \{x_i \mid x_i \text{ is real valued}\}$, and $Z = \{x_i \mid x_i \text{ is restricted to be integer}\}$. These vectors focus on the actual coefficient values of each of the variables:

- vector of coefficients of the objective function (of all, integer, or continuous variables): $(c_i | x_i \in X)$ where $X = U \vee X = Z \vee X = R$,
- vector of RHS of the $\leq$ ($\geq$ or $=$) constraints: $(b_j | A_j x \circ b_j)$ where $\circ = (\geq) \vee \circ = (\leq) \vee \circ = (=)$,
- vector of number of variables (all, integer, or continuous) per constraint j: $(\#\{A_{(i,j)} \mid A_{(i,j)} \neq 0, x_i \in X\})$ where $X = U \vee X = Z \vee X = R$,
- vector of the coefficients of variables (all, integer, or continuous) per constraint j: $(\sum_i A_{(i,j)} | \forall j, x_i \in X)$ where $X = U \vee X = Z \vee X = R$, and
- vector of the number of constraints each variable i (all, integer, or continuous) belongs to: $(\#\{A_{(i,j)} \mid A_{(i,j)} \neq 0, x_i \in X\}$ where $X = U \vee X = Z \vee X = R$.

Computation of these feature values on average took only 0.02 s per instance.

MIPs are used to model a wide variety of problem types. Therefore, in order to capture the spectrum of possible instances, we assembled a highly diverse benchmark dataset composed of problem instances from six different sources. Network flow instances, capacitated facility location instances, bounded and unbounded mixed knapsack instances and capacitated lot sizing problems, all taken from [100], as well as combinatorial auction instances from [68]. In total there are 588 instances in this set, which was split into 276 training and 312 test instances.

- Given some graph, the *network flow problem* aims to find the maximal flow that can be routed from node s to node t while adhering to the capacity constraints of each edge. The interesting characteristic of these problems is that special-purpose network flow algorithms can be used to solve such problems much faster than general-purpose solvers.
- In the *capacity facility problem*, a collection of demand points and a distance function are defined. The task is then to place n supply nodes that minimize some distance objective function while maintaining that each supply node does not service too many demand points. Problems of this type are generally solved using Lagrangian relaxation and matrix column generation methods.
- The *knapsack problem* is a highly popular problem type that frequently appears in real-world problems. Given a collection of items, each with an associated profit and weight, the task of a solver is to find a collection of items that results in the highest profit while remaining below a specified weight capacity constraint. In the bounded knapsack version, there are multiple copies of each item while in the unbounded version there is an unlimited number of copies of each item. Usually these types of problems are solved using a branch-and-bound approach.
- The task of the *capacitated lot sizing problem* is to determine the amount and timing of production to generate a plan that best satisfies all the customers. Specifically, at each production step, certain items can be produced using a specific resource. Switching the available resource incurs a certain price, as does maintaining items in storage. The problem also specifies the number of copies of each item that need to be generated and by what time. This is a very complex problem that in practice usually is defined as a MIP.
- In a *combinatorial auction*, participants place bids on combinations of discrete items rather just on a single item. These auctions have been traditionally used to

auction estates, but have recently also been applied to truckload transportation and bus routes. Another important recent application was the auction of the radio spectrum for wireless communications. The problem specification is given a collection of bids to find the most profitable allocation of items to bidders. In practice, these problems usually are modeled as the set packing problem.

4.2.1 Solver

For these experiments, Cplex 12.1 is used. Since 1999, IBM's Cplex [59] has represented the state-of-the-art optimization package. Used by many of the world's leading commercial firms and researchers in over 1,000 universities, Cplex has become a critical part of optimization research. Although the specific techniques and implementations are kept proprietary, the solver provides flexible, high-performance optimizers for solving linear programming, mixed integer programming, quadratic programming, and constraint programming problems. For each of these problem types, Cplex can handle problems with millions of constraints and variables, often setting performance records. The solver also has numerous options for tuning solving strategies for specific problems, which makes it ideal for the purposes of this example.

4.2.2 Numerical Results

Experiments were carried out with a timeout of 30 s for training and 300 s for evaluation on the training and testing sets. The size of the smallest cluster is set to be 30 instances. This resulted in five clusters, where four consisted of only one problem type, and one cluster combined network flow and capacitated lot sizing instances.

Table 4.4 compares instance-specific ISAC with instance-oblivious GGA and the default Cplex. It is observed again that the default parameters can be significantly improved by tuning the algorithm for a representative benchmark. For the average

Table 4.4 Comparison of ISAC versus the default and the instance-oblivious parameters provided by GGA when tuning Cplex

Solver		Avg. run time		Geo. avg.		Avg. slow down	
		Train	Test	Train	Test	Train	Test
Cplex	Default	6.1	7.3	2.5	2.5	2.0	1.9
	GGA	3.6	5.2	1.7	1.8	1.3	1.2
	ISAC	**2.9**	**3.4**	**1.5**	**1.6**	**1.0**	**1.0**

The table presents the arithmetic and geometric mean runtimes as well as the average slowdown per instance

test instance ISAC-Cplex needs 3.4 s, GGA-Cplex needs 5.2 s and default Cplex requires 7.3 s. Instance-obliviously tuned Cplex is 50 % slower, and the default Cplex even more than 114 % slower than ISAC-Cplex.

The improvements achieved by automatic parameter tuning can be seen when considering the average per-instance slow-downs. According to this measure, for a randomly chosen instance in the test set it is expected that GGA-Cplex needs 1.2 times the time required by ISAC-Cplex. Default Cplex needs 1.9 times the time of ISAC-Cplex.

It is necessary to note that due to license restrictions only a very small training set of 276 instances could be used, which is very few given the high diversity of the considered benchmark. Taking this into account and seeing that Cplex is a highly sophisticated and extremely well-tuned solver, the fact that ISAC boosts performance so significantly is surprising and shows the great potential of instance-specific tuning.

4.3 SAT

Our next evaluation of ISAC is on the propositional satisfiability problem (SAT), the prototypical NP-complete problem that has far-reaching effects in many areas of computer science. For SAT, given a propositional logic formula F in conjunctive normal form, the objective is to determine whether there exists a satisfying truth assignment to the variables of F. Since early 2000, there has been tremendous progress in solving SAT problems, so that modern SAT solvers can now tackle instances with hundreds of thousands of variables and over one million clauses.

The well-established features proposed by [126] are used to classify each problem instance. However, in preliminary experiments it is found that the local search features mentioned in [126] take a considerable amount of time to compute and are not imperative to finding a good clustering of instances. Consequently, these features were excluded, and only the following are used:

- problem size features: number of clauses c, number of variables v, and their ratio c/v,
- variable-clause graph features: degree statistics for variable and clause nodes,
- variable graph features: node degree statistics,
- balance features: ratio of positive to negative literals per clause, ratio of positive to negative occurrences of each variable, fraction of binary and ternary clauses,
- proximity to Horn clauses: fraction of Horn clauses and statistics on the number of occurrences in a Horn clause for each variable,
- unit propagations at depths 1, 4, 16, 64, and 256 on a random path in the DPLL [31] search tree, and
- search space size estimate: mean depth to contradiction and estimate of the log of the number of nodes.

Computation of these feature values on average took only 0.01 s per instance.

Table 4.5 Datasets used to evaluate ISAC on SAT

Dataset	Train	Test	Reference
QCP	1,000	1,000	[39]
SWGCP	1,000	1,000	[38]
3SAT-random	800	800	[81]
3SAT-structured	1,000	1,000	[104]

The collection of SAT instances described in Table 4.5 is considered. The objective of the quasigroup completion problem (QCP) is to determine if a partially filled N by N matrix can be filled with numbers $\{1 \ldots N\}$ so that the elements of each row are different and the elements of each column are different. As a benchmark we take a collection of QCP instances that have been encoded as a SAT instance. The graph coloring problem is given a graph G and N possible colors; the objective is to assign each node in the graph a color such that no adjacent nodes are assigned the same color. As one type of instance in our benchmarks, a set of these graph coloring problems (SWGCP) that have been encoded as SAT instances is used. The third problem type in the benchmark is randomly generated SAT instances (3SAT-random) using the G2 generator [116]. Finally considered are randomly generated SAT instances (3SAT-structured) that more closely resemble industrial SAT instances by our introducing structure into the instances.

4.3.1 Solver

ISAC is tested on the highly parameterized stochastic local search solver SAPS [57]. Unlike most existing SAT solvers, SAPS was originally designed with automatic tuning in mind and therefore all of the parameters influencing the solver are readily accessible to users. Furthermore, since it was first released, the default parameters of the solver have been improved drastically by general-purpose parameter tuners [5, 56].

In addition to tuning a single solver, a portfolio-like solver is created. This solver is comprised of nine competitive SAT solvers, where the job of ISAC is to identify not only which solver is best suited for the instance but also the best parameters for that solver. This is done by making the choice of the solver an additional categorical parameter to be set by ISAC. The used solvers have all been ranked either in first, second or third place in a recent SAT competition: clasp 1.3.2, jerusat 1.3, kcnfs 2006, march pl, minisat 2.0, mxc 09, rsat 2.01, and zchaf.

4.3.2 Numerical Results

Experiments were carried out with a timeout of 30 s for training and 300 s for evaluation on the training and testing sets. The size of the smallest cluster was set to be at least 100 instances. This resulted in 18 clusters, each with roughly 210 instances. Here not only were all of the four types of instances correctly separated into distinct clusters, a further partition of instances from the same class was provided.

The performance of SAPS was evaluated using the default parameters, GGA, and ISAC and we present the results in Table 4.6.

Even though the default parameters of SAPS have been tuned heavily in the past [56], tuning with GGA solves the benchmark over five times faster than default SAPS. Instance-specific tuning allows us to gain another factor of 2.9 over the instance-oblivious parameters, resulting in a total performance improvement of over one order of magnitude. This refutes the conjecture of [54] that SAPS may not be a good solver for instance-specific parameter tuning.

It is worth noting that over 95 % of instances in this benchmark can be solved in under 15 s. Consequently, some exceptionally hard, long-running instances greatly dilute the average runtime. The average slow-down per instance is therefore presented again. For the average SAT instance in our test set, default SAPS runs 274 times slower than ISAC. Even if GGA is used to tune a parameter set specifically for this benchmark, GGA is still expected to run almost five times slower than ISAC.

Table 4.7 presents the performance of an algorithm portfolio-style solver tuned using ISAC. The table shows that by creating an algorithm where one of the parameters identifies the solver to use to evaluate the instance, the resulting solver can perform three times better than the best overall solver on all the instances.

Table 4.6 Comparison of the SAPS solvers with default, GGA-tuned, and ISAC parameters

Solver		Avg. run time		Geo. avg.		Avg. slow down	
		Train	Test	Train	Test	Train	Test
SAPS	Default	79.7	77.4	0.9	0.9	292.5	274.1
	GGA	14.6	14.6	0.2	0.2	5.5	4.7
	ISAC	**4.0**	**5.0**	**0.1**	**0.1**	**1.0**	**1.0**

The arithmetic and geometric mean runtimes in seconds are presented as well as the average slow-down per instance

Table 4.7 Performance of a portfolio style SAT solver tuned using ISAC compared to performance of each of the solvers in the portfolio

	clasp	jerusat	kcnfs	march	minisat	mxc	rsat	zchaf	ISAC	Oracle
Train	15.1	42.1	98.5	57.1	22.1	19.2	43.8	64.3	**5.9**	3.4
Test	16.8	39.9	95.6	51.9	22.7	20.1	42.9	64.6	**5.8**	2.9

The table presents the average runtime. Oracle is a portfolio algorithm that always chooses the best solver for the given instance

However, it can also be seen that there is still room for improvement. Oracle is a best-case scenario algorithm portfolio that holistically always chooses the best solver for the given instance. This best-case scenario still requires half the time of the algorithm tuned by ISAC.

4.4 Machine Reassignment

Given the growing level of interest from the optimization community in data center optimization and virtualization, the 2012 Roadef Challenge [41] was focused on machine reassignment, a common task in virtualization and service configuration on data centers. Informally, the machine reassignment problem is defined by a set of machines and a set of processes. Each machine is associated with a set of available resources, e.g., CPU, RAM, etc., and each process is associated with a set of required resource values and a currently assigned machine. The task is provided with an assignment of processes to machines, to find an improving reassignment within a 5 min timeout.

There are several reasons for reassigning one or more processes from their current machines to different machines. For example, if the load of the machine is high, then one might want to move some of the processes from that machine to other machines. Similarly, if the machine is about to shut down for maintenance then one might want to move all the processes of the machine. Also, if there exists a machine in a different location where the electricity price is cheaper, then one might want to reassign processes to the machines such that the cost of electricity consumption is reduced. In general, the task is to reassign the processes to machines while respecting a set of hard constraints in order to improve the usage of the machines, as defined by a complex cost function.

For the example presented in this section, a collection of 1,245 instances was generated based on the 10 set B instances from the 2012 Roadef Challenge. For details on how these instances were generated, we refer the reader to [74]. The generated dataset was split to contain 745 training instances and 500 test instances. Table 4.8 then shows the features of the machine reassignment problem and their limits on the instances of the problem that we are interested in solving.

Table 4.8 Features

Feature	Limit
Machines	5,000
Processes	50,000
Resources	20
Services	5,000
Locations	1,000
Neighborhoods	1,000
Dependencies	5,000

4.4.1 Solver

Due to the size of the problems and the speed with which an improving solution needs to be found, a large neighborhood search (LNS) approach was employed. The problem itself was formulated using constraint programming (CP), which is described in [79]. In this book, details of the CP model and the general solving methodology are omitted but can be found in [79] and [74] respectively. Here we will discuss only the general scheme of the solver and the relevant parameters.

LNS combines the power of systematic search with the scaling of local search. The overall solution method for CP-based LNS is shown in Fig. 4.1. The current assignment is maintained, which is initialized to the initial solution given as input. At every iteration step, a subset of the processes to be reassigned is selected, and the domains of the variables of the CP model are updated accordingly. The resulting CP problem is solved with a threshold on the number of failures, and the best found solution is kept as the new current assignment.

In the case of the LNS solver, Table 4.9 lists and explains the parameters that can be controlled. Although there are only six parameters, half of them are continuous and have large domains. Therefore, it is impractical to try all possible configurations. Furthermore, the parameters are not independent of each other. This was found by gathering a small set of 200 problem instances and evaluating them with 400

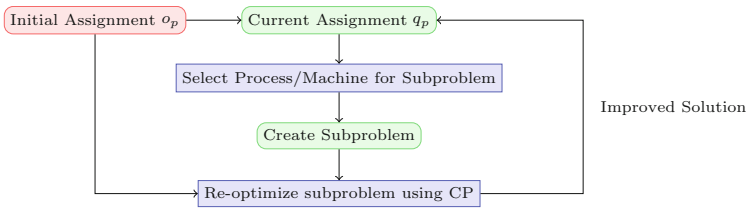

Fig. 4.1 Principles of the CP-based LNS approach

Table 4.9 Parameters of LNS for the machine reassignment problem

Notation	Type	Range	Description
u_p	Integer	[1, 50]	Upper bound on the number of processes that can be selected from one machine for reassignment
t_p	Integer	[1, 100]	Upper bound on the total number of processes that can be selected for reassignment
r_p	Continuous	[0.1, 1]	Ratio between the average number of processes on a machine
t_m	Integer	[2, 25]	Upper bound on the number of machines selected for subproblem selection
r_m	Continuous	[0.1, 10]	Ratio between the upper bound on the consecutive non-improving iterations and the average number of processes on a machine
t_f	Continuous	[0.1, 10]	Ratio between the threshold on the number of failures and the total number of processes selected for reassignment

randomly selected parameter settings. Using the average performance on the dataset as our evaluation metric and the parameter settings as attributes, feature selection algorithms from Weka [42] were run. All the attributes were found to be important. Adding polynomial combinations of the parameter settings further revealed that some pairs of parameters were more important than others when predicting expected performance.

4.4.2 Numerical Results

This section compares three parameterizations of the LNS solver. The first approach is the LNS solver, denoted by Default, which was the runner-up in the challenge. Here Default stands for a single set of parameters resulting from manual tuning of LNS solver on the 20 instances provided by the 2012 Roadef Challenge organizers. The second approach, marked GGA, was tuned in an instance-oblivious manner using all the training instances. Finally, the third approach, labeled ISAC, was trained using the methodology proposed in this book. All the experiments were run on Linux 2.6.25 x64 on a dual quad-core Xeon CPU machine with overall 12 GBs of RAM and processor speed of 2.66 GHz.

For evaluation of a solver's performance, the metric utilized for the Roadef competition was used:

$$Score_S(I) = 100 * (Cost(S) - Cost(B))/Cost(R).$$

Here, I is the instance, B is the best observed solution using any approach, R is the original reference solution, and S is the solution using a particular solver. The benefit of this evaluation function is that it is not influenced by some instances having a higher cost than others, and instead focuses on a normalized value that ranks all of the competing approaches. The best observed cost is used because for most of the instances it is not possible to find the optimal cost. The lower bound for a given instance was obtained by aggregating the resource requirements over all processes and (safety) capacities over all machines and then computing the sum of the load and balance costs.

To streamline the evaluations, the best performance was approximated using the performance achieved by running the LNS solver with default parameters for 1 h. While this caused some of the scores to be negative during training, this approximation still correctly differentiated the best solver while avoiding placing more weight on instances with higher costs.

The performances of the learned parameterizations from the Default, GGA, and ISAC methodologies are compared in Figs. 4.2 and 4.3. The figures plot the average performance of each method on the instances in each of the 10 discovered clusters. What we observe is that even though the default parameters perform very close to as well as they can for clusters 4, 5, 6, and 7, for clusters 2, 8, and 10 the performance is very poor. Tuning the solver using GGA can improve the average performance

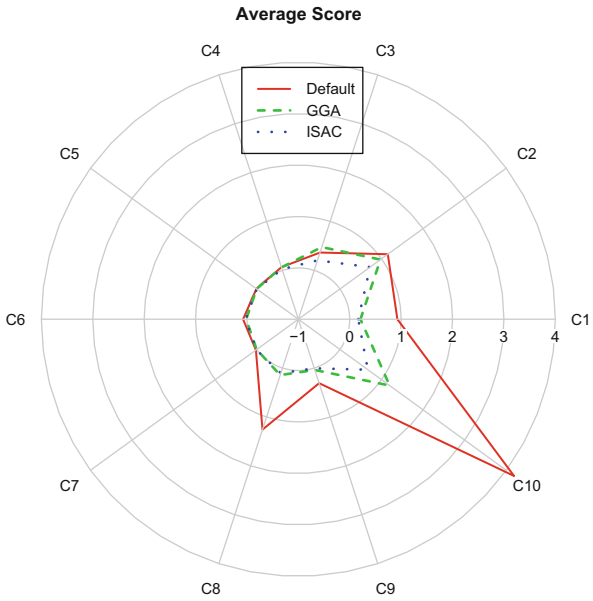

Fig. 4.2 Mean scores of each cluster for Default, GGA and ISAC approaches

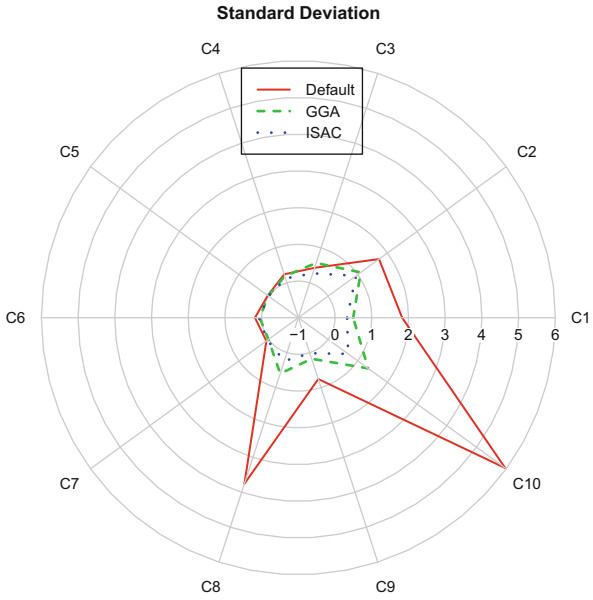

Fig. 4.3 Standard deviations of each cluster for Default, GGA and ISAC approaches

dramatically. Furthermore, we see that if we focus on each cluster separately, we can further improve performance, highlighting that different parameters should be employed for different types of instances. Interestingly, we observe that ISAC also dramatically improves on the standard deviation of the scores (in Fig. 4.3),

suggesting that the tuned solvers are consistently better than the default and GGA-tuned parameters.

4.5 Chapter Summary

This chapter presented the possible enhancements that can be achieved by using the cluster-based training approach ISAC. The experiments were done on four different optimization problem types and seven different solvers. In all cases solvers trained using the ISAC approach outperformed their alternatively tuned counterparts. This chapter began by showing that instance-oblivious parameter tuning is a powerful technique that can yield better results than an instance-specific regression approach. It was then shown how, by using the cluster-based approach, ISAC is able to enhance the instance-oblivious parameter tuning. Subsequent experiments were aimed at applying ISAC to a variety of solvers, showing improvements for each. Finally, the experiments on the Roadef Challenge highlighted that this approach is also beneficial on real-world problems.

Chapter 5
ISAC for Algorithm Selection

The inception of algorithm portfolios [40, 67, 87, 125] has had a dramatic impact on constraint programming, operations research, and many other fields. Based on the observation that solvers have complementary strengths and therefore exhibit incomparable behavior on different problem instances, the ideas of running multiple solvers in parallel or selecting one solver based on the features of a given instance were introduced. Appropriately, these approaches have been named *algorithm portfolios*. Portfolio research has led to a wealth of different approaches and an amazing boost in solver performance in the past decade.

One of the biggest success stories is that of SATzilla, which combined existing Boolean satisfiability (SAT) solvers and dominated various categories of the SAT Competition for about half a decade [1]. Another example is CP-Hydra [87], a portfolio of CP solvers which won the CSP 2008 Competition. Instead of choosing a single solver for an instance, Smith–Miles [103] proposed a Dirichlet Compound Multinomial distribution to create a schedule of solvers to be run in sequence. Approaches like [51] dynamically switched between a portfolio of solvers based on the predicted completion time. Alternatively, ArgoSmart [84] and Hydra [124] focus not only on choosing the best solver for an instance, but also on the best parametrization of that solver. For a further overview of the state of the art in portfolio generation, see the thorough survey in [105].

Ultimately, here we aim to develop algorithm portfolios that are able to deal effectively with a vast range of input instances from a variety of sources. For example, in the context of SAT, one would ideally want to design a *single* portfolio that manages to determine the most appropriate solver given an instance regardless of the instance "type" (such as random, crafted, or industrial).

As discussed in previous chapters, ISAC is a generalization of instance-oblivious configurators such as ParamILS [56] and GGA [5]. Interestingly, through extensive experimentation, this chapter shows that the ideas behind ISAC can be effectively applied to algorithm selection, resulting in solvers that significantly outperform highly efficient SAT solver selectors.

Y. Malitsky, *Instance-Specific Algorithm Configuration*,
DOI 10.1007/978-3-319-11230-5_5

5.1 Using ISAC as Portfolio Generator

This chapter presents how the ISAC methodology can be used to generate a portfolio of SAT solvers. Three ways, of differing complexity, are considered. Assume we are given a set of (potentially parameterized) algorithms $A_1, \ldots, A_n$, a set of training inputs T, the set of associated feature vectors F, and a function $Features$ that returns a feature vector for any given input x.

- **Pure Solver Portfolio:** Cluster the training instances according to their normalized features. For each cluster, determine the overall best algorithm. At runtime, determine the closest cluster and tackle the input with the corresponding solver.
- **Optimized Solver Portfolio:** Proceed as before. For each cluster, instance-obliviously tune the preferred algorithm for the instances within that cluster.
- **Instance-Specific Meta-Solver Configuration:** Define a parameterized meta-algorithm where the first parameter determines which solver is invoked, and the remaining parameters determine the parameters for the underlying solvers. Use ISAC to tune this solver.

The difference between the pure solver portfolio and the other two approaches is that the first is limited to using the solvers with their default parameters. This means that the performance that can be achieved maximally is limited by that of the "virtually best solver" (a term used in the SAT competition), which gives the runtime of the best solver (with default parameters) for each respective instance. The difference between the optimized solver portfolio and the instance-specific meta-solver configuration is that the latter may find that a solver that is specifically tuned for a particular cluster of instances may work better overall than the best default solver for that cluster, even if the latter is tuned. Therefore, note that the potential for performance gains strictly increases from stage to stage. For the optimized solver portfolio as well as the instance-specific meta-solver configuration it is possible to outperform the virtually best solver.

The remainder of this chapter presents numerical results comparing these three approaches to the state-of-the art portfolio generators for SAT.

5.2 Algorithm Configuration vs. Algorithm Selection of SAT Solvers

This section begins the experimental study by comparing ISAC with the 2009 SATzilla_R portfolio. To this end portfolios are generated based on the following solvers: Ag2wsat0 [120], Ag2wsat+ [121], gnovelty+ [92], Kcnfs04 [32], March_dl04 [47], Picosat 8.46 [15], and SATenstein [62]. Note that these solvers are *identical* to the ones that the SATzilla09_R [123] solver was based on when

it was entered in the 2009 SAT solver competition.[1] To make the comparisons as fair as possible, these experiments use the same feature computation program made public by the developers of SATzilla to get the 48 core features to characterize a SAT input instance (see [126] for a detailed list of features).

The training set was comprised of the random instances from the 2002 to 2007 SAT Competitions, where instances that are solved in under a second by all the solvers in our portfolio are removed. Also removed were instances that cannot be solved with any of the solvers within a time limit of 1,200 s (this is the same timeout as that used in phase 1 of the SAT Competition). This left 1,582 training instances. The test set consisted of the 570 instances from the 2009 SAT Solver Competition [1] where SATzilla_R won gold. SATzilla_R is the version of SATzilla tuned specifically for random SAT instances. The random instances were chosen because they belonged to the category where SATzilla showed the most marked improvements over the competing approaches. The cluster-based approaches were trained on dual Intel Xeon 5540 (2.53 GHz) quad-core Nehalem processors and 24 GB of DDR-3 memory (1,333 MHz).

Like SATzilla, ISAC utilized the PAR10 score, a penalized average of the runtimes: for each instance that is solved within 1,200 s, the actual runtime in seconds defines the penalty for that instance; for each instance that is not solved within the time limit, the penalty is set to 12,000, which is 10 times the original timeout. Note that for the pure solver portfolio, we require much less time than SATzilla to generate the portfolio. Clustering takes negligible time compared to running the instances on the various solvers. However, when determining the best solver for each cluster, we can race them against each other, which means that the total CPU time for each cluster is the number of solvers multiplied by the time taken by the fastest solver (as opposed to the total time of running all solvers on all instances).

For the optimized solver portfolio generator and the meta-solver configurator, the instance-oblivious configurator GGA was employed on each cluster of training instances. In particular, the parameters used for GGA were the following (please refer to [5] for details): The standard population size was set to 100 genomes, split evenly between the competitive and noncompetitive groups. Initial tournaments considered five randomly chosen training instances. The size of this random subset grows linearly with each iteration until the entire training set is included by iteration 75. GGA then proceeded tuning until the 100th generation or until no further improvement in performance was observed. These default parameters of GGA were used because of their performance in prior research.

For the meta-solver, parameters needed to be trained for 16 clusters. These clusters were built to have at least 50 instances each, which resulted in the average cluster having 99 instances and the largest cluster having 253 instances. In total, building the MSC required 260 CPU days of computation time. However, since

[1] Note that the benchmark for a portfolio generator consists of both the training and test sets of problem instances as well as the solvers used to build the portfolio!

each of the clusters could be tuned independently in parallel, only 14 days of tuning were required.

5.2.1 *Pure Solver Portfolio vs. SATzilla*

Table 5.1 reports the results of the experiments. As a pure solver, gnovelty+ performs best, solving 51 % of all test instances in time, whereby a number of the other solvers exhibit similar performance. Even though no individual solver can do better, when they join forces in a portfolio performance can be significantly boosted, as the pioneers of this research thread have pointed out [40, 66]. SATzilla_R, for example, solves 71 % of the test instances within the given captime of 1,200 s. Seeing that the virtually best solver (VBS) sets a hard limit of 80 % of instances that can be solved in time by these solvers, this performance improvement is very significant: more than two-thirds of the gap between the best pure solver and the VBS is closed by the SATzilla portfolio. Here, VBS (virtual best solver) assumes an oracle-based portfolio approach that always chooses the fastest solver for each instance.

The table also shows the performance of the various portfolios based on ISAC. Observe that on the data SATzilla_R was trained on, it performs very well, but on the actual test data even the simple pure solver portfolio generated by ISAC manages to

Table 5.1 Comparison of SATzilla, the pure solver portfolio (PSP), the instance-specific meta-solver configuration (MSC), and the virtually best solver (VBS)

Solver	gnovelty+	SATzilla_R	PSP	Cluster	MSC	VBS	MSC+pre
Training dataset							
PAR10	4,828	685	1,234	1,234	505	58.6	456
σ	5,846	2,584	3,504	3,504	2,189	156	2,051
Avg	520	153	203	203	129	58.6	128
σ	574	301	501	501	267	156	261
Solved	951	1,504	1,431	1,431	1,527	1,582	1,534
%	60.1	95.1	90.5	90.5	96.5	100	97.0
Testing dataset							
PAR10	5,874	3,578	3,569	3,482	3,258	2,482	2,480
σ	5,963	5,384	5,322	5,307	5,187	4,742	4,717
Avg	626	452	500	470	452	341	357
σ	578	522	501	508	496	474	465
Solved	293	405	408	411	422	457	458
%	51.4	71.1	71.6	72.1	74.0	80.2	80.4

Also shown is the best possible performance that can be achieved if the same solver must be used for all instances in the same cluster (Cluster). The last columns show the performance of the meta-solver configuration with a pre-solver (MSC+pre). For the penalized and regular average of the time, σ, the standard deviation, is also presented

outperform SATzilla_R. On the test set, the pure solver portfolio has a slightly better PAR10 score (the measure that SATzilla was trained for), and it solves a few more instances (408 compared to 405) within the given time limit. That means, in terms of the average runtime, while SATzilla closes about two-thirds of the gap between the best individual solver and the VBS, the simple PSP already closes about 60 % of the remaining gap between SATzilla and the VBS.

It is important to note here that in the 2011 SAT Competition, the difference between the winning solver in the industrial instance category and the tenth placed solver was 24 instances. This tenth place solver was also the winner of the 2009 SAT Competition. The improvements observed here with the PSP approach are significant.

In Table 5.1, given under "Cluster" is the best PAR10 score, the average runtime, and the number of solved instances *when a single solver is committed to each cluster*. It can be observed that the clustering itself incurs some cost in performance when compared to the VBS.

5.2.2 Meta-Solver Configuration vs. SATzilla

When considering the optimized solver portfolio, where the best solver is tuned for each cluster, none of the best solvers chosen for each cluster had parameters. Therefore, the performance of the OSP is identical to that of the PSP. The situation changes when the meta-solver configuration approach is used.

As Table 5.1 shows, the MSC provides a significant additional boost in test performance. This portfolio manages to solve 74 % of all instances within the time limit, 17 instances more than SATzilla. This improvement over the VBS is due to the introduction of two new configurations of SATenstein that MSC tuned and assigned to two clusters.

In SATzilla, the portfolio is not actually a pure algorithm selector. In the first minute, SATzilla employs both the mxc-sr08 [24] SAT solver and a specific parameterization of SATenstein. That is, SATzilla runs a schedule of three different solvers for each instance. In Table 5.1, MSC+pre is a version of an ISAC-tuned portfolio that uses the first minute of allotted time to run these same two solvers. The resulting portfolio outperformed the VBS. This was possible because the MSC added new parameterizations of SATenstein, and also because mxc-sr08 is not one of our pure solvers. As a result, the new portfolio solved 80 % of all competition instances, 9 % more than SATzilla. At the same time, runtime variance was also greatly reduced: not only did the portfolio run more efficiently, it also worked more robustly. Seeing that ISAC was originally not developed with the intent to craft solver portfolios, this performance improvement over a portfolio approach that had dominated SAT competitions for half a decade was significant. Based on these results, the 3S Solver Portfolio was entered in the 2011 SAT Competition. 3S is just one portfolio (no sub-versions _R or _I) for all different categories, which comprises 36 different SAT solvers. 3S was the first sequential portfolio that won gold in more

than one main category (SAT+UNSAT instances). The specifics of this solver are presented in Chap. 6.

Although not explicitly shown, all the results are significant as per the Wilcoxon signed rank test with continuity correction. MSC is faster than SATzilla_R with $p \leq 0.1\,\%$.

5.2.3 *Improved Algorithm Selection*

When compared with PSP, MSC replaced some of the default solvers for some clusters with other default solvers and, while lowering the training performance, this resulted in an improved test performance. To explain this effect it is important to understand how GGA tunes this meta-solver. As discussed in previous chapters, GGA is a genetic algorithm with a specific mating scheme. Namely, some individuals need to compete against each other to gain the right of mating. This competition is executed by racing several individual parameter settings against one another *on a random subset of training instances*. That means that GGA will likely favor the solver for a cluster that has the greatest chance of winning the tournament on a random subset of instances.

Note that this is different from choosing the solver that achieves the best score on the entire cluster, as was done for the pure solver portfolio (PSP). What is observed here is that the PSP over-fits the training data. GGA implicitly performs a type of bagging [25], which results in solver assignments that generalize better.

Motivated by this insight, two more methods were tested for the generation of a pure solver portfolio. The two alternative methods for generating cluster-based portfolios are:

- **Most Preferred Instances Portfolio (PSP-Pref):** Here, for each cluster, the fastest solving algorithm is determined for each instance in that cluster. The cluster is then associated with the solver that most instances prefer.
- **Bagged Portfolio (PSP-Bag):** For each cluster: a random subset of our training instances in that cluster is chosen and the fastest (in terms of PAR10 score) solver is determined (note that each solver is only run once for each instance). This solver is the winner of this "tournament." The process is repeated 100 times and the solver that wins the most tournaments is associated with this cluster.

In Table 5.2, these three cluster-based algorithm selectors are compared with SATzilla_R (whereby these portfolios are again augmented by running SATenstein and mxc-sr08 for the first minute, which is indicate by adding "+pre" to the portfolio name). Observe that PSP-Pref+pre is clearly not resulting in good performance. This is likely because it is important to note not only which solver is best, but also how much better it is than its contenders. PSP+pre works much better, but it does not generalize as well on the test set as PSP-Bag+pre. Therefore, when the base solvers of a portfolio have no parameters, the PSP-Bag approach to develop a high-performance algorithm selector is recommended.

Table 5.2 Comparison of alternate strategies for selecting a solver for each cluster

Solver	SATzilla	PSP+pre	PSP-Pref+pre	PSP-Bag+pre
Training dataset				
PAR10	685	476	2,324	531
σ	2,584	2,070	4,666	2,226
Avg	153	141	289	142
σ	301	489	465	280
Solved	1,504	1,533	1,284	1,525
%	95.1	97.0	81.2	96.4
Testing dataset				
PAR10	3,578	2,955	5,032	2,827
σ	5,384	5,024	5,865	4,946
Avg	452	416	560	402
σ	522	489	562	484
Solved	405	436	334	442
%	71.1	76.5	58.6	77.5

5.2.4 Latent-Class Model-Based Algorithm Selection

In [103], an alternative model-based portfolio approach was presented. The paper addressed the problem of computing the prediction of a solver's performance on a given instance using natural generative models of solver behavior. Specifically, the authors used a Dirichlet Compound Multinomial (DCM) distribution to create a schedule of solvers; that is, instead of choosing just one solver, they gave each solver a reduced time limit and ran this schedule until the instance was solved or the time limit was reached. For their experiments, the authors used the 570 instances from the 2009 SAT Competition in the Random category, along with the 40 additional random instances from the same competition originally used for a tie-breaking round. This data set of 610 instances was then used to train a latent-class model using random sub-sampling.

The authors found that this portfolio led to a slight improvement over SATzilla_R. However, they also mentioned that the comparison was not fully adequate because the latent-class model scheduler used newer solvers than SATzilla and also the 610 instances were used for both training and testing.

The experiments used the same data used in the original research of DCM.[2] These times were run on an Intel quad-core Xeon X5355 (2.66 GHz) processor with 32 GB of RAM. As competitors, we trained our algorithm selection portfolios based

[2]Our thanks go to Bryan Silverthorn, who provided the 610 instances used in the experiments in [103], as well as the runtime of the constituent solvers on his hardware, and also the final schedule of solvers that the latent class model found (see [103] for details).

Table 5.3 Comparison with the DCM Portfolio developed by Silverthorn and Miikkulainen [103] (results presented here were reproduced by Silverthorn and sent to us in personal communication)

Solver	SATzilla	DCM	PSP	PSP-Bag
PAR10	12,794	12,265	7,092	7,129
σ	182	314	180	293
Avg	1,588	1,546	1,242	1,250
σ	16.6	21.7	14.8	19.5
Solved	458	465	531	530
σ	2.38	4.03	2.36	3.83
%	75.1	76.2	87.0	86.9
σ	0.39	0.66	0.39	0.63
Solved (median)	458	464	531	531
% (median)	75.1	76.0	87.0	87.1

The table presents mean run-times and median number of solved instances for 10 independent experiments

on the previously mentioned 1,582 instances from the Random category of the 2002 to 2007 SAT Competitions.

Table 5.3 shows the performance of SATzilla_R, DCM, and our PSP and PSP-Bag (without the "-pre" option!) using a 5,000 s timeout. To account for the random nature of the underlying solvers, the evaluation of the DCM schedule and our portfolios was repeated ten times. The table shows mean and median statistics. Even though, as mentioned earlier, the comparison with SATzilla_R is problematic, it is included here to make sure that our comparison is consistent with the finding in [103] that DCM works slightly better than SATzilla. The results in the table confirm this. However, the PSP and PSP-Bag portfolios can do much better and boost the performance from 76 % of all instances solved by the DCM to 87 % solved by PSP-Bag. Keeping in mind the simplicity of clustering and solver assignment, this improvement in performance is noteworthy.

5.3 Comparison with Other Algorithm Configurators

As shown in the previous section, when the employed solvers have parameters, ISAC and the meta-solver configuration approach offer more potential than the pure solver portfolios PSP and PSP-Bag, which serve merely as algorithm selectors. In this section, ISAC is compared with two other approaches that train the parameters of their solvers, ArgoSmart [84] and Hydra [124].

5.3.1 ISAC vs. ArgoSmart

An alternate version of the idea that parameterized solvers can be used in a portfolio is also in considered in ArgoSmart [84]. Using a supervised clustering approach, the authors build groups of instances based on the directory structure in which the SAT Competition placed these instances. The authors enumerate all possible parameterizations of ArgoSAT (60 in total) and find the best parameterization for each family. For a test instance, ArgoSmart then computes 33 of the 48 core SATzilla features that do not involve runtime measurements [126] and then assigns the instance to one of the instance families based on majority k nearest neighbor classification, based on a non-Euclidean distance metric. The best parameterization for that family is then used to tackle the given instance.

ISAC is more widely applicable, as it clusters instances in an unsupervised fashion. Moreover, ISAC employs GGA to find the solver parameters instead of enumerating all possible configurations. Therefore, if the parameter space were much bigger, the ArgoSmart approach would need to be augmented with an instance-oblivious parameter tuner to find parameters for each of the instance families that it inferred from the directory structure. Despite these current limitations of the ArgoSmart methodology, we compared our assignment of test instances to clusters based on the Euclidean distance to the nearest cluster center with more elaborate machine learning techniques.

To make this assessment, a PSP and a PSP-Bag were generated based on the time of each of ArgoSAT's parameterizations on each instance.[3] These times were computed on Intel Xeon 2 GHz processors with 2 GB RAM. In Table 5.4, ArgoSmart is compared with two versions of PSP and PSP-Bag, respectively. Both use the same 33 features of ArgoSmart to classify a given test instance. In one version, unsupervised clustering of the training instances is used. The other version uses the supervised clustering gained from the directory structure of the training instances which ArgoSmart used as part of its input. For both variants the best possible cluster-based performance is given. Observe that the supervised clustering offers more potential. Moreover, when PSP-Bag has access to this clustering, despite its simple classification approach, it performs as well as the machine learning approach from [84]. However, even when no supervised clustering is available as part of the input, ISAC can still tune ArgoSAT effectively.

Note that the times of ArgoSmart are different from those reported in [84] because the authors only had the times for all parameterizations for the 2002 SAT data, and not the 2007 SAT data they originally used for evaluation. The authors generously retuned their solver for a new partitioning of the 2002 dataset, to give the presented results.

[3] Information that was generously provided by Mladen Nikolic

Table 5.4 Comparison with ArgoSmart [84] (results presented here were reproduced by Nikolic and sent to us in personal communication)

Solver	ArgoSat	ArgoSmart	Unsupervised clustering			Supervised clustering			VBS
			PSP	PSP-Bag	Cluster	PSP	PSP-Bag	Cluster	
Training dataset									
PAR10	2,704	–	2,515	2,527	2,515	2,464	2,473	2,464	2,343
σ	2,961	–	2,935	2,967	2,935	2,927	2,959	2,927	2,906
Avg	294	–	276	276	276	270	271	270	255
σ	285	–	283	284	283	283	283	283	283
Solved	736	–	778	775	778	789	787	789	815
%	55.4	–	58.5	58.3	58.5	59.4	59.2	59.4	61.3
Testing dataset									
PAR10	2,840	2,650	2,714	2,705	2,650	2,650	2,650	2,628	2,506
σ	2,975	2,959	2,968	2,967	2,959	2,959	2,959	2,959	2,941
Avg	306	286	291	290	286	286	286	281	269
σ	286	286	287	287	286	286	286	287	286
Solved	337	357	350	351	357	357	357	359	372
%	53.1	56.2	55.1	55.3	56.2	56.2	56.2	56.5	58.6

5.3.2 ISAC vs. Hydra

The methodology behind our final competitor, Hydra [124], enjoys the same generality as ISAC. Hydra consists of a portfolio of various configurations of the highly parameterized local search SAT solver SATenstein. In Hydra, a SATzilla-like approach is used to determine whether a new configuration of SATenstein has the potential of improving a portfolio of parameterizations of SATenstein, and a ParamILS-inspired procedure is used to iteratively propose new instantiations of SATenstein. In other words Hydra creates and adds solvers to its portfolio one at a time, even removing those same solvers when they are deemed to no longer help the overall performance of the portfolio.

To cover the breadth of possibilities, three different approaches are considered for building a portfolio of local search SAT solvers and are compared with Hydra[4] in Tables 5.5 and 5.6. The respective benchmarks BM and INDU were introduced in [124]. Both instance sets appear particularly hard for algorithm configuration: in [124], Hydra was not able to outperform an algorithm selection portfolio with 17 constituent solvers. The BM and INDU benchmarks consist of 1,500 training and 1,500 test instances, and 500 training and 500 test instances, respectively. The INDU dataset is comprised of only satisfiable industrial instances, while BM is composed of a mix of satisfiable crafted and industrial instances. These experiments used dual

[4]We are grateful to Lin Xu, who provided the Hydra-tuned SATensteins as well as the mapping of test instances to solvers.

Table 5.5 Comparison of local-search SAT solvers and portfolios thereof on BM data

Solver	Saps	Stein (FACT)	Hydra	MSC-stein	PSP-Bag 11	PSP-Bag 17	MSC-12
Training dataset							
PAR10	102	26.8	–	1.78	18.03	1.41	1.41
σ	197	109	–	13.6	87.9	4.09	4.16
Avg	13.5	4.25	–	1.48	3.63	1.11	1.41
σ	19.6	11.3	–	4.41	10.4	3.05	4.16
Solved	1,206	1,425	–	1,499	1,452	1,499	1,500
%	80.4	95.0	–	99.9	96.8	99.9	100
Testing dataset							
PAR10	861	220	1.43	1.27	73.5	1.21	1.21
σ	2,086	1,118	5.27	3.73	635	4.42	3.27
Avg	97.8	26.0	1.43	1.27	12.3	1.20	1.21
σ	210	114	5.27	3.73	69.0	4.42	3.27
Solved	1,288	1,446	1,500	1,500	1,483	1,500	1,500
%	85.9	96.4	100	100	98.9	100	100

Table 5.6 Comparison of local-search SAT solvers and portfolios thereof on INDU data

Solves	Saps	Stein (CMBC)	Hydra	MSC-stein	PSP-Bag 11	PSP-Bag 17	MSC-12
Training dataset							
PAR10	54.6	6.40	–	2.99	51.7	3.97	3.00
σ	147	23.5	–	3.94	143	22.6	4.47
Avg	10.5	5.50	–	2.99	10.3	3.07	3.00
σ	15.5	8.07	–	3.94	15.3	6.54	4.47
Solved	451	499	–	500	454	499	500
%	90.2	99.8	–	100	90.8	99.8	100
Testing dataset							
PAR10	208	5.35	5.11	2.97	209	3.34	2.84
σ	1,055	8.54	9.41	4.08	1,055	7.05	4.07
Avg	35.7	5.35	5.11	2.97	36.4	3.34	2.84
σ	116	8.54	9.41	4.08	116	7.05	4.07
Solved	484	500	500	500	484	500	500
%	96.8	100	100	100	96.8	100	100

Intel Xeon 5540 (2.53 GHz) quad-core Nehalem processors and 24 GB of DDR-3 memory (1,333 GHz) to compute the runtimes.

The training of the portfolios was conducted using a 50 s timeout; for testing a 600 s timeout was used. It is important to point out that, despite using a tenfold longer training timeout than [124], the total training time for each portfolio was about 72 CPU days, which is comparable with the 70 CPU days reported in [124] (note also that significantly slower machines for tuning were used). The reason is that GGA was used instead of ParamILS to train the solvers on each cluster. GGA is population-based and races parameter sets against each other, which means that

runs can be interrupted prematurely when a better parameter set has already won the race. It is an inherent strength of ISAC that it can handle longer timeouts than Hydra. Compared in the presented results are the two approaches, assuming they are given the same number of CPU days during which to tune.

The portfolio closest to Hydra is denoted MSC-stein. Here, like Hydra, only SATenstein is tuned. As usual, this approach clusters our training instances, and for each cluster SATenstein is tuned using GGA. For evaluation, as for the original Hydra experiments, each solver is run three times and the median time is presented. Observe again that the clustering approach to portfolio generation offers advantages. While Hydra uses a SATzilla-type algorithm selector to decide which tuned version of SATenstein an instance should be assigned to, ISAC employs clusters for this task. As a result, ISAC has a 12 % reduction in runtime over Hydra on the BM dataset and more than 40 % reduction on INDU. There is also a significant reduction in runtime variance over Hydra: again, not only does the new portfolio work faster, it also works more robustly across various instances.

Next, the ISAC methodology is used further to build portfolios with more constituent solvers. Following the same setting as in [124], an algorithm selector was built using 11 local search solvers (PSP-Bag 11): paws [113], rsaps [58], saps [114], agwsat0 [120], agwsat+ [121], agwsatp [119], gnovelty+ [92], g2wsat [69], ranov [91], vw [94], and anov09 [49]. In this setting, saps' performance is the best. The number of constituent solvers is further augmented through the addition of six fixed parameterizations of SATenstein, resulting in a total of 17 constituent solvers. The respective portfolio is denoted PSP-Bag 17. Finally, MSC-12 is built based on the (unparameterized) 11 original solvers plus the (highly parameterized) SATenstein.

Consistent with [124] are the following:

- Except on the INDU dataset, where the portfolio of 11 solvers cannot improve the performance of the best constituent solver, the portfolios boost significantly the performance compared to the best constituent solver (saps for the 11 solvers on both benchmarks and, for the 17 solvers, SATenstein-FACT on the BM dataset and SATenstein-CMBC on the INDU dataset).
- The portfolio of 17 solvers dramatically improves performance over the portfolio of 11 solvers. Obviously the variants of SATenstein work very well and, on the INDU benchmark, also provide some much needed variance so that the portfolio is now able to outperform the best solver.

In [124] it was found that Hydra, based on only the SATenstein solver, can match the performance of the portfolio of 17 solvers on both benchmarks. While this may be true when the portfolios are built using the SATzilla methodology, it is not true when using our algorithm selector PSP-Bag 17. On BM, PSP-Bag 17 works more than 15 % faster than Hydra and on the INDU benchmark set it runs more than 33 % faster.

The full potential of the ISAC approach is of course only realized when a portfolio is built using parameterized and unparameterized solvers. The result is

MSC-12, which clearly outperforms all others, working on average almost 18 % faster than Hydra on BM and more than 45 % faster than Hydra on INDU.

5.4 Chapter Summary

This chapter presented the idea of using instance-specific algorithm configuration (ISAC) for the construction of SAT solver portfolios. The approach works by clustering training instances according to normalized feature vectors. Then, for each cluster, it determines the best solver or computes a high-performance parameterization for a solver. At runtime, the nearest cluster is identified for each instance and the corresponding solver/parameterization is invoked. In all experiments, to compare competing approaches, every precaution was taken to make sure that the conditions under which they were developed were as close as possible. This included using the same solvers in the portfolio, the same tuning times, and the same training and testing sets.

The chapter showed that this very simple approach results in portfolios that clearly outperform the SAT portfolio generator SATzilla [126], a recent SAT solver scheduler based on a latent-class model, and the algorithm configuration method Hydra [124]. At the same time, ISAC is widely applicable and works completely unsupervised.

This study shows that instance-specific algorithm tuning by means of clustering instances and tuning parameters for the individual clusters is highly efficient even as an algorithm portfolio generator. The fact that, when tuning instance-specifically, ISAC considers portfolios of a potentially infinite number of solvers does not mean that it is necessary to revert to sub-standard portfolio selection. On the contrary: unsupervised clustering, which originally was a mere concession to tuning portfolios with extremely large numbers of solvers, has resulted in a new state of the art in portfolio generation.

Chapter 6
Dynamic Training

ISAC is a powerful tool for training solvers for the instances they will be applied on. One of ISAC's strengths lies in its configurability. Composed of three steps (computing features, clustering, and training), this methodology is not restricted to any single approach for any of them. For example, so far we have shown how a local search, GGA, and selecting the single best solver in a portfolio are all possibilities for training a solver for a cluster. In this chapter we show how by changing the clustering portion of the methodology, it is possible to train the portfolio dynamically for each new test instance. Using SAT as the testbed, the chapter demonstrates through extensive numerical experiments that this new technique is able to handle even highly diverse benchmarks, in particular a mix of Random, Crafted, and Industrial instances, even when the training set is not fully representative of the test set that needs to be solved.

6.1 Instance-Specific Clustering

The previous chapters used g-means [44] to analyze the training data and find a stable set of clusters. Although this has been shown to work very well in practice, improved performance is possible for the instances that are far from the found cluster centers. For example, in our version of g-means, a minimum cluster size is imposed to ensure that there are enough instances in each cluster to train a solver accurately. This means that instances in clusters that are smaller than this threshold are reassigned to the nearest neighboring cluster. This reassignment can potentially bias the training of the solver. To help prevent this scenario, a possible solution is to use the k nearest neighbor approach to create the clusters dynamically for each new test instance.

Y. Malitsky, *Instance-Specific Algorithm Configuration*,
DOI 10.1007/978-3-319-11230-5_6

6.1.1 Nearest Neighbor-Based Solver Selection

Nearest neighbor classification (k-NN) is a classical machine learning approach. In essence, the decision for a new example is based on prior experience in the k most similar cases. In our context, this means that we first identify which k training instances are the most "similar" to the one given at runtime, and then choose the solver that worked the "best" on these k training instances. As before, we use Euclidean distance on 48 normalized[1] core features of SAT instances that SATzilla is based on [126] as the similarity measure, and the PAR10 score of a solver on these k instances as the performance measure.

When $k = 1$, it is assumed that each training example is unique, and therefore that there are no errors on the training set as each instance is its own nearest neighbor. However, it is well known in machine learning that 1-NN often does not generalize well to formerly unseen examples, as it tends to over-fit the training data. A very large value of k also obviously defeats the purpose of considering local neighborhoods. To address the challenge of finding the "right" value of k, another classical strategy in machine learning is employed, namely *random subsampling cross validation*. The idea is to utilize only a subset of the training data and to assess how well a learning technique performs when trained on this subset and evaluated on the remaining training instances. A split ratio of $67/33$ is used to partition the training data and perform random subsampling 100 times to obtain a fairly good understanding of how well the technique generalizes to instances on which it was not trained. Finally, the k yielding the best average performance on the 100 validation sets is chosen.

Algorithm 5 gives a more formal description of the entire algorithm, in terms of its usage as a portfolio solver (i.e., algorithm selection given a new instance, as described above) and of the random subsampling-based training phase performed to compute the best value for k to use. The training phase starts out by computing the runtimes of all solvers on all training instances, as well as the features of these instances. It then removes all instances that cannot be solved by any solver in the portfolio within the time limit, or that are solved by every solver in the portfolio within marginal time (e.g., one second for reasonably challenging benchmarks); learning to distinguish between solvers based on data from such instances is pointless. Along with the estimated best k, the training phase passes this reduced set of training instances, their runtimes for each solver, and their features to the main solver selection phase. Note that the training phase does not learn any sophisticated model (e.g., a runtime prediction model); rather, it simply memorizes the training performances of all solvers and only actually "learns" the value of k.

Despite the simplicity of this approach—compared, for example, with the description of SATzilla in [126]—it is highly efficient and outperforms

[1] We associate each feature with a linear normalization function, ensuring that the feature's minimum and maximum values across the set of training instances are 0 and 1, respectively.

Algorithm 5: Algorithm selection using nearest neighbor classification

k**-NN-Algorithm-Selection Phase**;
Input : a problem instance F
Params: nearest neighborhood size k, candidate solvers $\mathcal{S}$, training instances $\mathcal{F}_{\text{train}}$ along with feature vectors and solver runtimes
Output: sat or unsat
begin
  compute normalized features of F;
  $\mathcal{F} \leftarrow$ set of k instances from $\mathcal{F}_{\text{train}}$ that are closest to F;
  $S \leftarrow$ solver in $\mathcal{S}$ with the best PAR10 score on $\mathcal{F}$;
  return $S(F)$;
end

Training Phase;
Input : candidate solvers $\mathcal{S}$, training instances $\mathcal{F}_{\text{train}}$, time limit T_{max}
Params: nearest neighborhood range $[k_{\text{min}}, k_{\text{max}}]$, perform random subsampling m times and split ratio m_b/m_v (default 70/30)
Output: best performing k, reduced $\mathcal{F}_{\text{train}}$ along with feature and runtimes
begin
  run each solver $S \in \mathcal{S}$ for time T_{max} on each $F \in \mathcal{F}_{\text{train}}$; record runtimes;
  remove from $\mathcal{F}_{\text{train}}$ instances solved by no solver, or by all within 1 s;
  compute feature vectors for each $F \in \mathcal{F}_{\text{train}}$ **for** $k \in [k_{min}, k_{max}]$ **do**
    score$[k] \leftarrow 0$;
    for $i \in [1..m]$ **do**
      $(\mathcal{F}_{\text{base}}, \mathcal{F}_{\text{validation}}) \leftarrow$ a random m_b/m_v split of $\mathcal{F}$;
      score$[k] \leftarrow$ score$[k]+$ performance of k-NN portfolio on $\mathcal{F}_{\text{validation}}$ using; training instances $\mathcal{F}_{\text{base}}$ and solver selection based on PAR10 score;
    end
  end
  $k_{\text{best}} \leftarrow \text{argmin}_k \text{score}[k]$;
  return $(k_{\text{best}}, \mathcal{F}_{\text{train}}$, feature vectors, runtimes);
end

SATzilla2009_R, the gold-medal winning solver in the random category of SAT Competition 2009. In Table 6.1 the k-NN algorithm selection is compared with SATzilla_R, using the 2,247 random category instances from SAT Competitions 2002 to 2007 as the training set and the 570 such instances from SAT Competition 2009 as the test set. As in the previous chapter, both portfolios are based on the following local search solvers: Ag2wsat0 [120], Ag2wsat+ [121], gnovelty+ [92], Kcnfs04 [32], March_dl04 [47], Picosat 8.46 [15], and SATenstein [62], all in the versions that are *identical* with the ones that were used when SATzilla09_R [123] entered in the 2009 SAT solver competition. To make the comparison as fair as possible, k-NN uses only the 48 core instance features that SATzilla is based on (see [126] for a detailed list of features), and is trained for the PAR10 score. For both training and testing, the time limit is set to 1,200 s. Table 6.1 shows that SATzilla boosts performance of individual solvers dramatically. The pure k-NN approach pushes the performance level substantially further. It solves 22 more instances than SATzilla and closes about one third of the gap between SATzilla and the virtual best

Table 6.1 Comparison of baseline solvers, portfolio, and virtual best solver performances: PAR10, average runtime in seconds, and number of instances solved (timeout 1,200 s)

	Pure solvers							Portfolios		
	agw-sat0	agw-sat+	**gnov-elty+**	SAT-enstein	march	pico-sat	kcnfs	SAT-zilla	*k*-**NN**	VBS
PAR10	5,940	6,017	5,874	5,892	8,072	10,305	6,846	3,578	**3151**	2482
σ	5,952	5,935	5,951	5,921	5,944	5,828	5,891	5,684	**5,488**	5,280
Avg time	634	636	626	625	872	1,078	783	452	**442**	341
σ	574	576	573	570	574	574	580	542	**538**	527
# solved	290	286	293	292	190	83	250	405	**427**	457
% solved	50.9	50.2	51.4	51.2	33.3	14.6	43.9	71.1	**74.9**	80.2

solver (VBS),[2] which solves 457 instances. Given the utter simplicity of the k-NN approach, this performance is quite remarkable.

6.1.2 Improving Nearest Neighbor-Based Solver Selection

This section discusses two techniques to improve the performance of the algorithm selector further. First, inspired by [87], training instances that are closer to the test instance are given more weight. Second, the neighborhood size k is adapted depending on the properties of the test instance to be solved.

6.1.2.1 Distance-Based Weighting

A natural extension of k-NN is to scale the scores of the k neighbors of an instance based on the Euclidean distance to it. Intuitively, larger weights are assigned to instances that are closer to the test instance, assuming that closer instances more accurately reflect the properties of the instance at hand. Hence, Line 17 in Algorithm 5 is updated to:

$$\text{score}[k] \leftarrow \text{score}[k] + PAR10 \times \left(1 - \frac{dist}{totalDist}\right),$$

where *dist* is the distance between the neighboring training instance and the current instance, and *totalDist* corresponds to the sum of all such distances. We proceed analogously in Line 5 when computing the best solver for a given test instance.

[2] VBS refers to the "oracle" selector that always selects the solver that is the fastest on the given test instance. Its performance is the best one can hope to achieve with algorithm selection.

6.1.2.2 Adaptive Neighborhood Size

Another idea is to learn not a single value for k, but to adapt the size of the neighborhood based on the given test instance. It is possible to partition the instance feature space by pre-clustering the training instances (we use g-means clustering [44] for this purpose). Then, a given instance belongs to a cluster when it is nearest to that cluster, whereby ties can be broken arbitrarily. This way, during training, rather than only one k that is supposed to work uniformly well being learned, a different k is learned for each cluster.

Algorithm 5 can be adapted easily to determine such cluster-based ks. Given a test instance, first the cluster to which it belongs is identified and then the value of k that was associated with this cluster during training is used. Observe that this clustering is not used to limit the neighborhood of a test instance. This means that neighboring instances from other clusters can still be used to determine the best solver for a given instance. The clusters are only used to determine the size of the neighborhood.

6.1.2.3 Experimental Evaluation

Observe that the two techniques, weighting and adaptive neighborhoods, are orthogonal to each other and can be combined. In the following, weighting, clustering, and their combination are compared with the pure k-NN portfolio.

Benchmark Solvers

In order to illustrate the improvements achieved by the extensions of k-NN, a new benchmark setting is introduced that mixes incomplete and complete solvers as well as industrial, crafted, and random instances. The following 21 state-of-the-art complete and incomplete SAT solvers are considered: Clasp [37], CryptoMiniSat [106], Glucose [10], LySat i/c [43], March-hi [45], March-nn [46], MXC [23], MiniSAT 2.2.0 [108], Lineling [17], PrecoSAT [16], Adaptg2wsat2009 [70], Adaptg2wsat2009++ [70], Gnovelty+2 [93], Gnovelty+2-H [93], HybridGM3 [12], Kcnfs04SAT07 [32], Picosat [15], Saps [58], TNM [118], and six parametrizations of SATenstein [62]. In addition, all industrial and crafted instances are preprocessed with SatElite (version 1.0, with default option "+pre"), where the following solvers were run on both the original and the preprocessed version of each instance: Clasp, CryptoMiniSat, Glucose, Lineling, LySat c, LySat i, March-hi, March-nn, MiniSat, MXC, and Precosat. That way, the portfolio was composed of 37 solvers.

Benchmark Instances

As before the set of benchmark instances was comprised of 5,464 instances selected from all SAT Competitions and Races during 2002 and 2010 [1], filtered for all instances that cannot be solved by any of the aforementioned solvers within the competition time limit of 5,000 s (i.e., the VBS can solve 100 % of all instances).

These instances were partitioned randomly multiple times into disjoint sets of training and testing instances, as well as into more challenging groups. The complex partition was based on omitting certain sets of instances from the training set, but on including them all in the test set. To assess which instances were related, it was assumed that instances starting with the same three characters belong to the same benchmark family. To this end, at random, a fraction of about 5 % of benchmark families were selected from among all families. This usually resulted in roughly 15 % of all instances being in the test partition. Aiming for a balance of 70 % training instances and 30 % test instances, the second step randomly chose instances until 30 % of all instances had been assigned to the test partition. Unless stated otherwise, all of the following experiments are conducted on this set of solvers and instances.

Results

Table 6.2 shows a comparison of the basic k-NN approach with the extensions of using weighting, clustering, and the combination of the two on this benchmark. Shown is the average performance in terms of number of instances solved/not solved, average runtime, and PAR10 score achieved across the 10 test sets mentioned in the previous paragraph. Note that a perfect oracle can solve all instances as instances that could not be solved by any solver within the given time limit of 5,000 s were discarded.

According to all of these measures, both weighting and clustering are able to improve the performance of the basic k-NN approach. This improvement is amplified when both methods are used simultaneously. The combined approach consistently outperforms basic k-NN on all our splits, solving about 0.5 % more instances.

Table 6.2 Average performance comparison of basic k-NN, weighting, clustering, and the combination of both using the k-NN portfolio

	Basic k-NN	Weighting	Clustering	Weight.+Clust.
# solved	1,609	1,611	1,615	**1,617**
# unsolved	114	112	108	**106**
% solved	93.5	93.6	93.8	**93.9**
Avg runtime	588	584	584	**577**
PAR10 score	3,518	3,459	3,368	**3,314**

For completeness, note that these results also translate to the SATzilla_R benchmark discussed earlier in Table 6.1. In this setting, the combination of weighting and clustering is able to solve seven more instances than the basic k-NN approach and 29 more than SATzilla_R. Here, the gap to the virtual best solver in terms of instances solved is narrowed down further to only 5 %, compared to the 6.6 % and 11.4 % lost by basic k-NN and SATzilla_R, respectively.

6.2 Building Solver Schedules

While the previous section shows that the k-NN tuned algorithm portfolio is able to significantly outperform not only the single best solver but also the highly successful SATzilla portfolio, there is still room for improvement with regard to the virtual best solver. To increase the robustness of the approach further, an alternate training methodology is considered. It is no longer feasible to tune solvers offline using the nearest neighbor clustering. As an alternative it is possible to compute a schedule that defines the sequence of solvers, along with individual time limits, given a new test instance. This sequence of solvers is then used to solve the instance. This approach is well justified by the different runtime distributions of constraint solvers. While one solver may fail to solve a given instance even in a very long time, another solver may well be able to solve the instance very quickly.

The general idea of scheduling for algorithm portfolios was previously introduced by Streeter [110] and in CP-Hydra [87]. In fact, Streeter [110] uses the idea of scheduling to *generate* algorithm portfolios. While he suggested using schedules that can suspend solvers and let them continue later on in exactly the same state they were suspended in, this section will focus on solver schedules without preemption, i.e., each solver will appear in the schedule at most once. This setting was also used in CP-Hydra, which computes a schedule of CP solvers based on k nearest neighbors. Specifically, a schedule is devised that determines which solver is run for how much time in order to attempt to solve the given instance.

It is important to first note that the *optimal* performance cannot be improved by a schedule of solvers, simply because using the fastest solver and sticking to it is the best we can hope for. Consequently, a solver schedule is still limited by the optimal performance of the VBS. In fact, the best performance possible for a schedule of solvers is limited by the VBS *with a reduced captime of the longest running solver in the schedule*. Therefore, trivial schedules that split the available time evenly between all solvers have inherently limited performance.

Nevertheless, the reason to be interested in solver schedules is to hedge our bets: it is often observed that instances that cannot be solved by one solver even in a very long time can in fact be solved by another very quickly. Consequently, by allocating a reasonably small amount of time to other solvers, it is possible to provide a safety net in case the solver selection happens to be unfortunate.

6.2.1 Static Schedules

The simplest approach is to compute a static schedule of solvers. For example, one could compute a schedule that solves the most training instances within the allowed time (cf. [87]). This section does slightly more, namely computing a schedule that, first, solves most training instances and that, second, requires the least amount of time all schedules that are able to solve the same number of training instances.

This problem can be formulated as an integer program (IP), more precisely as a resource constrained set covering problem (RCSCP):

Solver Scheduling IP:

$$\min \quad (C+1)\sum_{i} y_i + \sum_{S,t} t x_{S,t} \tag{6.1}$$

$$s.t. \quad y_i + \sum_{(S,t)\mid i \in V_{S,t}} x_{S,t} \geq 1 \qquad \forall i \tag{6.2}$$

$$\sum_{S,t} t x_{S,t} \leq C \tag{6.3}$$

$$y_i, x_{S,t} \in \{0,1\} \qquad \forall i, S, t \tag{6.4}$$

The constraints (6.2) in this model enforce that all training instances are covered; the additional resource constraint (6.3) ensures that the overall captime C is not exceeded. Binary variables $x_{S,t}$ in (6.4) correspond to sets of instances that can be solved by solver S within time t. These sets have cost t and a resource consumption coefficient t. Finally, to make it possible that all training instances can be covered, additional binary variables y_i are introduced. These correspond to the set that contains only item i; they have cost $C+1$ and time resource consumption coefficient 0. The objective is obviously to minimize the total cost. Due to the high costs for variables y_i (which will be 1 if and only if instance i cannot be solved by the schedule), the schedules which solve most instances are favored, and among those the fastest schedule (cost of $x_{S,t}$ is t) is chosen.

6.2.2 A Column Generation Approach

The main problem with the above formulation is the sheer number of variables. For the benchmark with 37 solvers and more than 5,000 training instances, solving the above problem is impractical, even when the timeouts t are chosen smartly such that from timeout $t1$ to the next timeout $t2$ at least one more instance can be solved by the respective solver ($V_{S,t1} \subsetneq V_{S,t2}$). In our experiments we found that the actual time to solve these IPs may at times still be tolerable, but the memory consumption was in many cases so high that we could not solve the instances.

The above stated problem can be resolved by means of column generation. Column generation (aka Dantzig-Wolfe decomposition) [30, 38] is a well-known technique for handling linear programs (LPs) with a lot of variables:

$$\min c^T x, \quad \text{s.t. } A\,x \geq b, x \geq 0. \tag{6.5}$$

Due to its size it is often not practical to solve the large system (6.5) directly. The core observation underlying column generation is that only a few variables will be non-zero in any optimal LP solution (at most as many as there are constraints). Therefore, if we know which variables are important, we can consider a much smaller system $A'\,x' = b$, where A' contains only a few columns of A. When we choose only some columns in the beginning, LP duality theory tells us which columns that we have left out so far are of interest for the optimization of the global LP. Namely, only columns with *negative reduced costs* (which are defined based on the optimal duals of the system $A'\,x' = b$) can be candidates for variables that can help the objective to decrease further.

Column generation proceeds by considering, in turn, a *master problem* (the reduced system $A'\,x' = b$) and a *subproblem* where we select a new column to be added to the master based on its current optimal dual solution. This process is iterated until there are no more columns with a negative reduced cost. At this point, we know that an optimal solution to (6.5) has been found—even though most columns have never been added to the master problem!

When using standard LP solvers to solve the master problem and obtain its optimal duals, all that is left is solving the subproblem. To develop a subproblem generator, we need to understand how exactly the reduced costs are computed. Assume we have a dual value $\lambda_i \geq 0$ for each constraint in A'. Then, the reduced cost of a column $\alpha := (\alpha_1, \ldots, \alpha_z)^T$ is defined as $\bar{c_\alpha} = c_\alpha - \sum_i \lambda_i \alpha_i$, where c_α is the cost of column α.

Equipped with this knowledge we compute a new column for A' that has minimal reduced costs. The process is begun by adding all columns to A' that correspond to variables y. Therefore, when we want to add a new column to the model it will regard a variable $x_{S,t}$ which corresponds to the solver-runtime pair (S, t). The goal of the subproblem at each step is to suggest a solver-runtime pair that is likely to increase the objective value of the (continuous) master problem the most.

To find this solver-runtime pair, first, for all solvers S, we compute a permutation π of the instances such that the time that S needs to solve instance $\pi_S(i)$ is less than or equal to the time that the solver needs to solve instance $\pi_S(i+1)$ (for appropriate i). Obviously, we only need to do this once for each solver and not each time we want to generate a new column.

Now, let us denote with $\lambda_i \geq 0$ the optimal dual value for the restriction to cover instance i (6.2). Moreover, denote with $\mu \leq 0$ the dual value of the resource constraint (6.3) (since that constraint enforces a lower-or-equal restriction, μ is guaranteed to be non-positive).

Now, for each solver S we iterate over i and compute the term $T \leftarrow \sum_{k \leq i} \lambda_{\pi_S(k)}$ (which in each iteration we can obviously derive from the previous value for T). Let

Algorithm 6: Subproblem: Column generation

```
begin
    minRedCosts ← ∞;
    forall the Solvers S do
        T ← 0 ;
        forall the i do
            j ← π(i);
            T ← T + λ_j;
            t ← Time(S, j);
            redCosts ← t(1 − μ) − T;
            if redCosts < minRedCosts then
                Solver ← S;
                timeout ← t;
                minRedCosts ← redCost;
            end
        end
    end
    if minRedCosts < 0 then return x_{Solver,timeout};
    else return None;
end
```

t denote the time that solver S needs to solve instance $\pi(i)$. Then, the reduced costs of the column that corresponds to variable $x_{S,t}$ are $t - t\mu - T$. We choose the column with the most negative reduced costs and add it to the master problem. If there is no more column with negative reduced costs, we stop.

It is important to note two things. First, that what we have actually done is pretend that all columns were present in the matrix and computed the reduced costs for all of them. This is not usually the case in column generation approaches, where most columns are usually found to have larger reduced costs *implicitly* rather than explicitly. Second, note that the solution returned from this process will in general not be integeral but contain fractional values. Therefore, the solution obtained cannot be interpreted as a solver schedule directly.

This situation can be overcome in two ways. The first is to start branching and to generate more columns—which may still be needed by the optimal integer solution even though they were superfluous for the optimal fractional solution. This process is known in the literature as branch-and-price.

Alternatively, what we do, and what is in fact the reason we solved the original problem by means of column generation in the first place, is stick to the columns that were added during the column generation process and solve the remaining system as an IP. Obviously, this is just a heuristic that may return suboptimal schedules for the training set. However, we found that this process is very fast and nevertheless provides high-quality solutions (see empirical results below). Even when the performance on the training set is at times slightly worse than optimal, the performance on the test set often turns out to be as good or sometimes even better

than that of the optimal training schedule—the cases when the optimal schedule overfits the training data.

The last case to address is the one where the final schedule does not utilize the entire available time. Recall that we even deliberately minimize the time needed to solve as many instances as possible. Obviously, at runtime it would be a waste of resources not to utilize the entire time that is at our disposal. In this case, we scale each solver's time in the schedule equally so that the total time of the resulting schedule will be exactly the captime C.

6.2.3 Dynamic Schedules

As mentioned earlier, CP-Hydra [87] is based on the idea of solver schedules. In their paper, the authors found that static schedules work only moderately well. Therefore, they introduced the idea of computing *dynamic* schedules: at runtime, for a given instance, CP-Hydra considers the ten nearest neighbors (in case of ties, up to 50 nearest instances) and computes a schedule that solves most of these instances in the given time limit. That is, rather than consider all training instances, it limits the constraints in the solver scheduling IP to the instances in the neighborhood.

In [87], the authors use a brute-force approach to compute dynamic schedules and observe that this works due to the small neighborhood size and the fact that CP-Hydra only has three constituent solvers (note that the time to produce a dynamic schedule takes away time for solving the actual problem instance!). Our column generation approach, yielding potentially suboptimal but usually high-quality solutions, works fast enough to handle even 37 solvers and 5,000 instances within seconds. This allows us to embed the idea of dynamic schedules in the previously developed nearest neighbor approach, which selects optimal neighborhood sizes by random subsampling cross-validation—which requires us to solve hundreds of thousands of these IPs.

Note that the idea of adaptive neighborhoods is orthogonal to dynamic solver scheduling: we can select the size of the neighborhood based on the distance to the nearest training cluster, independently of whether we use that neighborhood size for solver selection or solver scheduling. Moreover, the idea of giving more weight to instances closer to the test instance can also be incorporated in solver scheduling. This is an idea that CP-Hydra also exploits, albeit in a slightly different fashion than shown here. Here we adapt the objective function in the solver scheduling IP by multiplying the costs for the variables y_i (recall that originally these costs were $C + 1$) with $2 - \frac{\text{dist}_i}{\text{totalDist}}$. This favors schedules that solve more training instances that are closer to the one that is to be solved.

Table 6.3 compares the four resulting dynamic schedules with our best algorithm selector from Sect. 6.1.2. In addition, we also used a setting inspired by the CP-Hydra approach. Here, we use a fixed-size neighborhood of ten instances to build a dynamic schedule by means of column generation. Moreover, for this approach

Table 6.3 Average performance of dynamic schedules

	No sched.	Dynamic schedules				
	Wtg+Clu	Basic k-NN	Weighting	Clustering	Wtg+Clu	SAT-Hydra
# solved	1,617	**1,621**	**1,621**	1,619	1,618	**1,621**
# unsolved	106	**102**	**102**	104	105	**102**
% solved	93.9	**94.2**	**94.2**	94.0	94.0	**94.2**
Avg runtime	**577**	637	629	629	631	626
PAR10 score	3,314	3,257	**3,246**	3,310	3,324	3,249

Additional comparison: *SAT-Hydra*

Table 6.4 Average performance of semi-static schedules compared with no schedules and with static schedules based only on the available solvers

	No sched.	Static sched.	Semi-static schedules			
	Wtg+Clu	Wtg+Clu	Basic k-NN	Weighting	Clustering	Wtg+Clu
# solved	1,617	1,572	1,628	1,635	1,633	**1,636**
# unsolved	106	151	94.6	87.5	90.2	**87.2**
% solved	93.9	91.2	94.6	94.9	94.8	**95.0**
Avg runtime	577	562	448	451	**446**	449
PAR10 score	3,314	4,522	2,896	2,728	2,789	**2,716**

we use the weighting scheme introduced in [87]. We refer to this approach as SAT-Hydra.

Observe that these dynamic schedules all achieve roughly the same performance. Weighting and clustering do not appear to have any significant impact on performance. Moreover, all dynamic portfolios consistently outperform even our best algorithm selector, albeit only slightly: the dynamic schedule increases the number of instances solved by roughly one quarter percent.

6.2.4 Semi-static Solver Schedules

Clearly, dynamic schedules do not result in the improvements that we had hoped for. Here, we therefore consider another way of creating a solver schedule. Observe that the algorithm selection portfolios that we developed in Sect. 6.1.1 can themselves be considered solvers. This means that we can add the portfolio itself to our set of constituent solvers and compute a "static" schedule for this augmented collection of solvers. We put "static" in quotes here because the resulting schedule is of course still instance-specific. After all, the algorithm selector portfolio chooses one of the constituent solvers based on the test instance's features. We refer to the result of this process as *semi-static solver schedules*.

Depending on the portfolios from Sect. 6.1.1 used, we obtain four semi-static schedules. We show the performance of these portfolios in Table 6.4. While

Table 6.5 Comparison of column generation and the solution to the optimal IP

Schedule by	# solved	# unsolved	% solved	Avg runtime (s)	PAR10 score
Optimal IP	1635.8	87.1	95.0	442.5	2708.4
Column generation	1635.7	87.2	95.0	448.9	2716.2

weighting and clustering do not lead to performance improvements for dynamic schedules, we observe that the relative differences in performance between basic k-NN and its extensions shown in Sect. 6.1.2 translate to the setting with scheduling as well.

Moreover, semi-static scheduling significantly improves the overall performance (compare with the first column in the table for the best results without scheduling). In terms of instances solved, all semi-static schedules solve at least 20 more instances within the time limit. Again, the combination of weighting and clustering achieves the best performance and it narrows the gap in percentage of instances solved to nearly 5 %. For further comparison, the second column shows the performance of a static schedule that was trained on the entire training set and that is the same for all test instances. This confirms the finding in [87] that static solver schedules are indeed inferior to dynamic schedules, and finds that they are considerably outperformed by semi-static solver schedules.

6.2.4.1 Quality of Results Generated by Column Generation

Table 6.5 illustrates the performance of the column generation approach. The table shows a comparison of the resulting performance achieved by the *optimal* schedule. In order to compute the optimal solution to the IP, we used Cplex on a machine with sufficient memory and a 15 s resolution to fit the problem into the available memory. As can be observed, the column generation is able to determine a high-quality schedule that results in a performance that nearly matches the one of the optimal schedule according to displayed measures.

6.2.5 Fixed-Split Selection Schedules

Based on this success, we consider a parametrized way of computing solver schedules. As discussed earlier, the motivation for using solver schedules is to increase robustness and hedge against an unfortunate selection of a long-running solver. At the same time, the best achievable performance of a portfolio is that of the VBS *with a captime of the longest individual run*. In both dynamic and semi-static schedules, the runtime of the longest running solver(s) was determined by the column generation approach working solely on training instances. This procedure inherently runs the risk of overfitting the training set.

Table 6.6 Average performance comparison of basic k-NN, weighting, clustering, and the combination of both using the k-NN portfolio with a static schedule for 10 % of the total available runtime and the portfolio on the remaining runtime

	Semi-static schedules	Fixed-split schedules			
	Wtg+Clu	Basic k-NN	Weighting	Clustering	Wtg+Clu
# solved	1,636	1,637	1,641	1,638	**1,642**
# unsolved	87.2	94.6	87.5	90.2	**87.2**
% solved	95.0	95.0	95.3	95.1	**95.3**
Avg runtime	449	455	447	452	**445**
PAR10 score	2,716	2,686	2,570	2,652	**2,554**

Consequently, we now consider splitting the time between an algorithm selection portfolio and the constituent solvers based on a parameter. For example, we could allocate 90 % of the available time for the solver selected by the portfolio. For the remaining 10 % of the time, we could run a static solver schedule. We refer to these schedules as *90/10-selection schedules*. Note that choosing a fixed amount of time for the schedule of constituent solvers is likely to be suboptimal for the training set but offers the possibility of improving test performance.

Table 6.6 captures the corresponding results. We observe clearly that by using this restricted application of scheduling we are able to outperform our best approach so far (semi-static scheduling, shown again in the first column). We are able to solve nearly 1,642 instances on average, which is six more than we were able to solve before. The gap to the virtual best solver is narrowed down to a mere 4.69 % ! Recall that we consider a highly diverse set of benchmark instances from the Random, Crafted, and Industrial categories. Moreover, we do not work with plain random splits, but splits where complete families of instances in the test set are not represented in the training set at all. In this setting, an accuracy above 95 % of the VBS is truly remarkable. Moreover, compared to the plain k-NN approach that we started with, the fixed-split selection schedules close roughly one third of the gap to the VBS.

6.3 Chapter Summary

This chapter showed how the ISAC methodology could be adopted to dynamically create clusters in a more refined instance-specific manner. Specifically, this chapter considered the problem of algorithm selection and scheduling so as to maximize performance when given a hard time limit within which a solution needs to be provided. Two improvements were considered for the simple nearest neighbor solver selection, weighting and adaptive neighborhood sizes based on clustering.

Table 6.7 Comparison of major portfolios for the SAT-Rand benchmark (570 test instances, timeout 1,200 s)

	SATzilla_R	SAT-Hydra	k-NN	90-10	VBS
# solved	405	419	427	**435**	457
# unsolved	165	151	143	**135**	113
% solved	71.5	73.5	74.9	**76.3**	80.2
Avg runtime	452	313	441.9	**400**	341
PAR10 score	3578	1211	3151	**2958**	2482

Furthermore, this chapter showed how the training of the solvers could be done dynamically by developing a lightweight optimization algorithm to compute near-optimal schedules for a given set of training instances. This allows us to provide an extensive comparison of pure algorithm selection, static solver schedules, dynamic solver schedules, and semi-static solver schedules, which are essentially static schedules combined with an algorithm selector.

It was shown that the semi-static schedules work the best from among these options. Finally, two alternatives were compared: using the optimization component or using a fixed percentage of the allotted time when deciding how much time to allocate to the solver suggested by the algorithm selector. In either case, a static schedule was used for the remaining time. This latter parametrization allowed us to avoid overfitting the training data and resulted in the best performance overall.

The discussed approach was tested on a highly diverse benchmark set with Random, Crafted, and Industrial SAT instances where we even deliberately removed entire families of instances from the training set. Semi-static selection schedules demonstrated an astounding performance and solved, on average, over 95 % of the instances that the virtual best solver is able to solve.

As a final remark, Table 6.7 closes the loop and considers again the first benchmark set from Sect. 6.1.1, which compared portfolios for the SAT Competition's random category benchmark set based on the same solvers as those of the gold-medal winning SATzilla_R. Overall, we go up from 405 (or 88.6 % of the VBS) instances solved for SATzilla_R to 435 (or 95.1 % of the VBS) instances solved for our fixed-split semi-static solver schedules. In other words, the fixed-split selection schedule closes over 50 % of the performance gap between SATzilla_R and the VBS.

Chapter 7
Training Parallel Solvers

In the last decade, solver portfolios have boosted the capability to solve hard combinatorial problems. Portfolios of existing solution algorithms have excelled in competitions in satisfiability (SAT), constraint programming (CP), and quantified Boolean formulae (QBF) [87, 109, 126].

Since around 2010, a new trend has emerged, namely the development of parallel solver portfolios. The gold-winning ManySAT solver [43] is, when features like clause sharing are ignored, a static parallel portfolio of the MiniSAT solver [108] with different parameterizations. At the 2011 SAT Competition, an extremely simple static parallel portfolio, ppfolio [98], dominated the wall-clock categories for random and crafted SAT instances and came very close to winning the applications category as well. In [90], another method was introduced to compute static parallel schedules that are optimal with respect to the training instances, based on formulating the problem as a non-linear optimization problem and considering only sequential constituent solvers.

The obvious next step is to therefore consider dynamic parallel portfolios, i.e., portfolios that are composed based on the features of the given problem instance. Traditionally, sequential portfolios simply *select* one of the constituent solvers that appears best suited for the given problem instance. And, as seen in Chap. 6, at least since the invention of CP-Hydra [87] and SatPlan [111], sequential portfolios also *schedule* solvers. That is, they may select more than just one constituent solver and assign each one a portion of the time available for solving the given instance.

The solver presented at the end of Chap. 6 dominated the sequential portfolio solvers at the 2011 SAT Competition, where it won gold medals in the CPU-time category for random and crafted instances. In this chapter, the 3S methodology is augmented to devise dynamic parallel SAT solver portfolios.

Y. Malitsky, *Instance-Specific Algorithm Configuration*,
DOI 10.1007/978-3-319-11230-5_7

7.1 Parallel Solver Portfolios

The objective of this chapter is to show how to generalize the ISAC technology for the development of parallel SAT solver portfolios. Recall that at the core of 3S lie two optimization problems. The first is the selection of the long running solver primarily based on the maximum number of instances solved. The second is the solver scheduling problem.

Consider the first problem when there are $p > 1$ processors available. The objective is to select p solvers that, as a set, will solve the most number of instances. Note that this problem can no longer be solved by simply choosing the one solver that solves the most instances in time. Moreover, it is now necessary to decide how to integrate the newly chosen solvers with the ones from the static schedule. The second problem is the solver scheduling problem discussed before, with the additional problem that solvers need to be assigned to processors so that the total makespan is within the allowed time limit.

A major obstacle in solving these problems efficiently is the symmetry induced by the identical processors to which each solver can be assigned. Symmetries can hinder optimization very dramatically as equivalent (partial) schedules (which can be transformed into one another by permuting processor indices) will be considered again and again by a systematic solver. For example, when there are eight processors, for each schedule over 40,000 (8 factorial) equivalent versions exist. An optimization that used to take about half a second may now easily take 6 h.

Another consideration is the fact that a parallel solver portfolio may obviously include parallel solvers as well. Assuming there are eight processors and a parallel solver employs four of them, there are 70 different ways to allocate processors for this solver. The developed portfolio will have 37 sequential and two four-core parallel solvers. The solver scheduling IP that needs to be solved for this case has over 1.5 million variables.

7.1.1 Parallel Solver Scheduling

Both optimization problems are addressed at the same time by considering the following IP. Let $t_S \geq 0$ denote the minimum time that solver S must run in the schedule, let $M = \{S; |; t_S > 0\}$ be the set of solvers that have a minimal runtime, let p be the number of processors, and let $n_S \leq p$ denote the number of processors that solver S requires.

Parallel Solver Scheduling IP: CPU time

$$\min \quad (pC+1)\sum_i y_i + \sum_{S,t,P} t n_S x_{S,t,P}$$

$$s.t. \quad y_i + \sum_{(S,t)\mid i\in V_{S,t}, P\subseteq\{1,\ldots,p\}, |P|=n_S} x_{S,t,P} \geq 1 \qquad \forall i$$

$$\sum_{S,t,P\subseteq\{1,\ldots,p\}\cup\{q\}, |P|=n_S} t x_{S,t,P} \leq C \qquad \forall q \in \{1,\ldots,p\}$$

$$\sum_{S,t,P\subseteq\{1,\ldots,p\}, |P|=n_S, t\geq t_S} x_{S,t,P} \geq 1 \qquad \forall S \in M$$

$$\sum_{S,t,P\subseteq\{1,\ldots,p\}, |P|=n_S} x_{S,t,P} \leq N$$

$$y_i, x_{S,t,P} \in \{0,1\} \qquad \forall i, S, t, P \subseteq \{1,\ldots,p\}, |P| = n_S$$

Variables y_i are exactly what they were before. There are now variables $x_{S,t,P}$ for all solvers S, time limits t, and subsets of processors $P \subseteq \{1,\ldots,p\}$ with $|P| = n_S$. $x_{S,t,P}$ is 1 if and only if solver S is run for time t on the processors in P in the schedule.

The first constraint is again to solve all instances with the schedule or count them as not covered. There is now a time limit constraint for each processor. The third set of constraints ensures that all solvers that have a minimal solver time are included in the schedule with an appropriate time limit. The last constraint finally places a limit on the number of solvers that can be included in the schedule.

The objective is again to minimize the number of uncovered instances. The secondary criterion is to minimize the total CPU time of the schedule.

Note that this problem needs to be solved both offline to determine the static solver schedule (for this problem, $M = \emptyset$ and the solver limit is infinite) and *during the execution phase* (when M and the solver limit are determined by the static schedule computed offline). Therefore, it is absolutely necessary to solve this problem quickly, despite its huge size and its inherent symmetry caused by the multiple processors.

Note also that the parallel solver scheduling IP does not directly result in an executable solver schedule. Namely, the IP does not specify the actual start times of solvers. In the sequential case this does not matter as solvers can be sequenced in any way without affecting the total schedule time or the number of instances solved. In the parallel case, however, it is necessary to ensure that the parallel processes are in fact run in parallel. This aspect is omitted from the IP above to avoid further complicating the optimization. Instead, after solving the parallel solver IP, the solvers are heuristically scheduled in a best-effort approach, whereby solvers may be preempted and the runtime of the solvers may eventually be lowered to obtain a legal schedule. In the experiments presented later in the chapter it is shown that in practice the latter is never necessary. Hence, the quality of the schedule is

never diminished by the necessity to schedule processes that belong to the same parallel solver at the same time.

7.1.2 Solving the Parallel Solver Scheduling IP

We cannot afford to solve the parallel solver scheduling IP exactly during the execution phase. Each second spent on solving this problem is 1 s less for solving the actual SAT instance. Hence, like 3S, we revert to solving the problem heuristically by not considering variables that were never introduced during column generation.

While 3S could afford to price all columns in the IP during each iteration, fortunately it is not actually necessary to do this here. Consider the reduced costs of a variable. Denote with $\mu_i \leq 0$ the dual prices for the instance-cover constraints, with $\pi_q \leq 0$ the dual prices for the processor time limits, with $\nu_S \geq 0$ the dual prices for the minimum time solver constraints, and with $\sigma \leq 0$ the dual price for the limit on the number of solvers. Finally, let $\bar{\nu}_S = \nu_S$ when $S \in M$ and 0 otherwise. Then:

$$\bar{c}_{S,t,P} = n_S t - \sum_{i \in V_{S,t}} \mu_i - \sum_{q \in P} t\pi_q - \bar{\nu}_S - \sigma.$$

The are two important things to note here. First, the fact that only variables introduced during the column generation process are considered means that the processor symmetry is reduced in the final IP. While it is not impossible, it is unlikely that the variables that would form a symmetric solution to a schedule that can already be formed from the variables already introduced would have negative reduced costs.

Second, to find a new variable that has the most negative reduced costs, it is not necessary to iterate through all $P \subseteq \{1, \ldots, p\}$ for all solver/time pairs (S, t). Instead, the processors can be ordered by their decreasing dual prices. The next variable introduced will use the first n_S processors in this order as all other selections of processors would result in higher reduced costs.

7.1.3 Minimizing Makespan and Post-processing the Schedule

Everything is now in place to develop the parallel SAT solver portfolio. In the training phase, a static solver schedule is computed based on all training instances for 10 % of the available time. This schedule is used to determine a set M of solvers that must be run for at least the static scheduler time at runtime. During the execution phase, given a new SAT instance, its features are computed, the k closest training instances are determined, and a parallel schedule is computed that will solve as many of these k instances in the shortest amount of CPU time possible.

In these experiments a second variant of the parallel solver scheduling IP is considered where the secondary criterion is to minimize not the CPU time but the makespan of the schedule. The corresponding IP is given below, where variable m measures the minimum idle time for all processors. The reduced cost computation changes accordingly.

Parallel Solver Scheduling IP: Makespan

$$\begin{aligned}
\min \quad & (C+1)\sum_i y_i - m \\
s.t. \quad & y_i + \sum_{(S,t)\ |\ i\in V_{S,t}, P\subseteq\{1,\ldots,p\}, |P|=n_S} x_{S,t,P} \geq 1 && \forall i \\
& m + \sum_{S,t,P\subseteq\{1,\ldots,p\}\cup\{q\}, |P|=n_S} t x_{S,t,P} \leq C && \forall q \in \{1,\ldots,p\} \\
& \sum_{S,t,P\subseteq\{1,\ldots,p\}, |P|=n_S, t\geq t_S} x_{S,t,P} \geq 1 && \forall S \in M \\
& \sum_{S,t,P\subseteq\{1,\ldots,p\}, |P|=n_S} x_{S,t,P} \leq N \\
& y_i, x_{S,t,P} \in \{0,1\} && \forall i, S, t, P \subseteq \{1,\ldots,p\}, |P| = n_S
\end{aligned}$$

Regardless of whether the CPU time or makespan is minimized, as remarked earlier, the result is post-processed by assigning actual start times to solvers heuristically. The resulting solver times are also scaled to use as much of the available time as possible. For low values of k, schedules are often computed that solve all k instances in a short amount of time without utilizing all available processors. In this case, new solvers are assigned to the unused processors in the order of their ability to solve the highest number of the k neighboring instances.

7.2 Experimental Results

Using the methodology above, two parallel portfolios are built. The first is based on the 37 constituent solvers of 3S. This portfolio is referred to as p3S-37. The second portfolio built includes two additional solvers, "Cryptominisat (2.9.0)" [107] and "Plingeling (276)" [18], both executed on four cores. This portfolio is referred to as p3S-39. It is important to emphasize again that all solvers that are part of our portfolio were available *before* the 2011 SAT Competition. In the experiments, these parallel portfolios will be compared with the parallel solver portfolio "ppfolio" [98] as well as "Plingeling (587f)" [19], both executed on eight cores. Note that these competing solvers are new solvers that were introduced for the 2011 SAT Competition.

The benchmark set of SAT instances is the same as that in prior sections composed of the 5,464 instances from all SAT Competitions and Races between 2002 and 2010 [1]; the 1,200 (300 application, 300 crafted, 600 random) instances from the 2011 SAT Competition have also been added. Based on this large set of SAT instances, a number of benchmarks are created. Based on all SAT instances that can be solved by at least one of the solvers considered in p3S-39 within 5,000 s, 10 equal partitions are created. These partitions are used to conduct a tenfold cross-validation, whereby in each fold nine partitions are used as the training set (for building the respective p3S-37 and p3S-39 portfolios), and the performance is evaluated on the partition that was left out before. For this benchmark, average performance over all ten splits is reported. On top of this cross-validation benchmark, the split induced by the 2011 SAT Competition is also considered. Here, all instances prior to the competition are used as the training set, and the SAT Competition instances are used as the test set. Lastly, a competition split is created based on application instances only.

As performance measures the number of instances solved, average runtime and the PAR10 score are considered. The PAR10 is a penalized average runtime where instances that time out are penalized with 10 times the timeout. Experiments were run on dual Intel Xeon 5540 (2.53 GHz) quad-core Nehalem processors with 24 GB of DDR-3 memory.

7.2.1 Impact of the IP Formulation and Neighborhood Size

Tables 7.1 and 7.2 show the average cross-validation performance of p3S-39 when using different neighborhood sizes k and the two different IP formulations (tie

Table 7.1 Average performance comparison of parallel portfolios when optimizing CPU time and varying neighborhood size k based on tenfold cross-validation

CPU time	10	25	50	100	200
Average (σ)	320 (45)	322 (43.7)	329 (42.2)	338 (43.9)	344 (49.9)
Par 10 (σ)	776 (241)	680 (212)	694 (150)	697 (156)	711 (221)
# solved (σ)	634 (2.62)	636 (2.22)	636 (1.35)	636 (1.84)	636 (2.37)
% solved (σ)	99.0 (0.47)	99.2 (0.39)	99.2 (0.27)	99.2 (0.28)	99.2 (0.41)

Table 7.2 Average performance comparison of parallel portfolios when optimizing makespan and varying neighborhood size k based on tenfold cross-validation

Makespan	10	25	50	100	200
Average (σ)	376.1 (40.8)	369.2 (42.9)	374 (40.7)	371 (40.8)	366 (36.9)
Par 10 (σ)	917 (200)	777 (192)	782 (221)	750 (153)	661 (164)
# solved (σ)	633 (2.16)	635 (2.28)	634.9 (2.92)	635 (1.89)	637 (2.01)
% solved (σ)	98.8 (0.39)	99.1 (0.39)	99.1 (0.46)	99.2 (0.32)	99.3 (0.34)

breaking by minimum CPU time and minimizing schedule makespan). As can be seen, the size of the neighborhood k affects the most important performance measure, the number of instances solved, only very little. There is a slight trend towards larger ks working a little bit better. Moreover, there is also not a great difference between the two IP formulations, but on average it is found that the version that breaks ties by minimizing the makespan solves about one instance more per split. Based on these results, p3S in the future refers to the portfolio learned on the respective training benchmark using $k = 200$ and the IP formulation that minimizes the makespan.

7.2.2 *Impact of Parallel Solvers and the Number of Processors*

Next, the impact of employing parallel solvers in the portfolio is demonstrated. Tables 7.3 and 7.4 compare the performance of p3S-37 (without parallel solvers) and p3S-39 (which employs two four-core parallel solvers) on the cross-validation and on the competition split. A small difference is observed in the number of solved instances in the cross-validation, and a significant gap is observed in the competition split.

Two issues are noteworthy about that competition split. First, since this was the latest competition, the instances in the test set of this split are probably significantly harder than the instances from earlier years. The relatively low percentage of instances solved even by the best solvers at the 2011 SAT Competition is another indication of this. Second, in some instances of families in this test set are completely missing in the training partition. That is, for a good number of instances

Table 7.3 Performance of tenfold cross-validation on all data

Cross	p3S-37		p3S-39	
Validation	4 core	8 core	4 core	8 core
Average (σ)	420 (22.1)	**355** (31.3)	435 (48.5)	366 (36.9)
Par 10 (σ)	991 (306)	679 (176)	1116 (256)	**661** (164)
Solved (σ)	630 (4.12)	633 (2.38)	631 (2.75)	**637** (2.01)
% solved (σ)	98.3 (0.63)	98.8 (0.35)	98.5 (0.49)	**99.3** (0.34)

Results are averages over the tenfolds

Table 7.4 Performance of the solvers on all 2011 SAT competition data

	p3S-37		p3S-39		
Competition	4 cores	8 cores	4 cores	8 cores	VBS
Average	1907	1791	1787	**1640**	1317
Par 10	12,782	12,666	11,124	**10,977**	10,580
Solved	843	865	853	**892**	953
% solved	70.3	72.1	71.1	**74.3**	79.4

in the test set there may be no training instance that is very similar. These features of any competition-induced split (which is the realistic split scenario!) explain why the average cross-validation performance is often significantly better than the competition performance. Moreover, they explain why p3S-39 has a significant advantage over p3S-37: when a lot of the instances are out of reach of the sequential solvers within the competition timeout, the portfolio must necessarily include parallel solvers to perform well.

As a side remark, the presence of parallel portfolios is what makes the computation challenging in the first place. In the extreme case, we could otherwise have as many processors as parallel processors, and then a trivial portfolio would achieve the performance of the virtual best solver, that is to say; the more processors one has, the easier sequential solver selection becomes. To show what would happen when the selection is made harder than it actually is under the competition setting and reduced the number of available processors to four. For both p3S-37 and p3S-39, the cross-validation performance decreases only moderately while, under the competition split, performance decays significantly. At the same time, the advantages of p3S-39 over p3S-37 shrink a lot. As one would expect, the advantage of employing parallel solvers decays with a shrinking number of processors.

7.2.3 Parallel Solver Selection and Scheduling vs. the State of the Art

The dominating parallel portfolio to date is ppfolio [98]. In the parallel track at the 2011 SAT Competition, it won gold in the crafted and random categories and came in just shy of winning the application category as well, where it was beaten by just one instance. In the application category, the winning solver was "Plingeling (587f)" run on eight cores. Both competing approaches are compared in Figs. 7.1 and 7.2.

The top plot in Fig. 7.1 shows the scaling behavior in the form of a "cactus plot" for eight-core runs of ppfolio, p3S-37, and p3S-39,[1] for the competition split containing all 1,200 instances used in the 2011 SAT Competition. This plot shows that p3S-39 (whose curve stays the lowest as it moves to the right) can solve significantly more instances than the other two approaches for any given time limit larger than around 800 s. It is also seen that p3S-37, based solely on sequential constituent solvers, performs similarly to ppfolio for time limits up to 3,000 s, and begins to outperform it for larger time limits.

The bottom plot in Fig. 7.1 shows the per-instance performance of p3S-39 vs. ppfolio, with runtimes in log-scale on both axes. More points being below the diagonal red line signifies that p3S-39 is faster than ppfolio on a large majority

[1]The plots shown here are for the CPU time optimization variant of p3S-37 and p3S-39. The ones for makespan optimization were very similar.

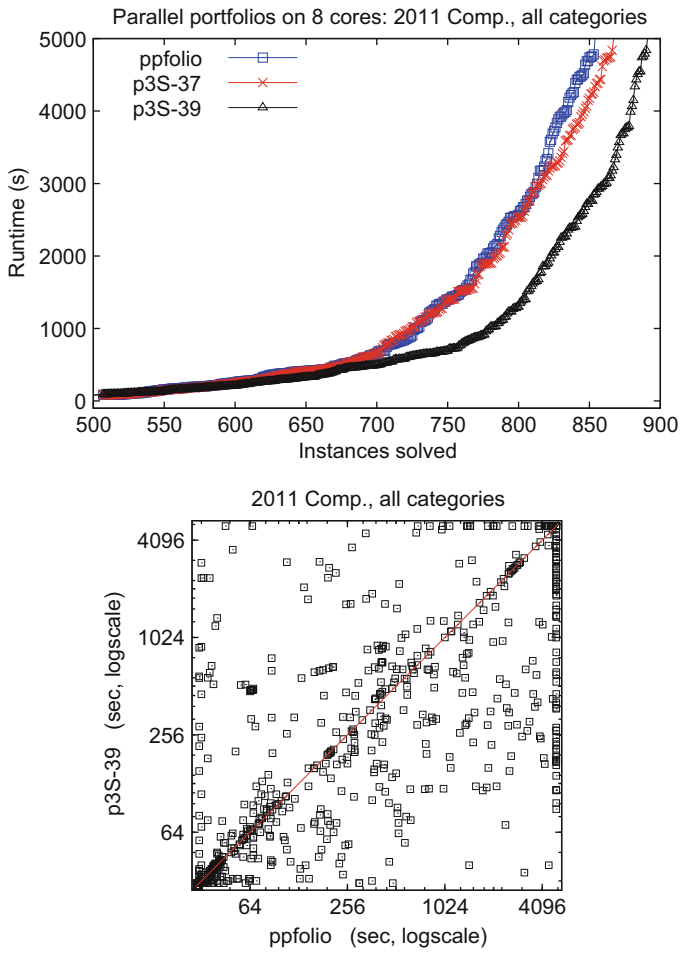

Fig. 7.1 Comparison on all 1,200 instances used in the 2011 SAT competition, across all categories. *Top* Cactus plot depicting the scaling behavior of solvers. *Bottom* Per-instance comparison between ppfolio and p3S-39

of the instances. ppfolio also times out on many instances that p3S-39 can solve, as evidenced by the large number of points on the right margin of the plot.

Overall, p3S-39 was able to solve 892 instances, 47 more than ppfolio. p3S-37 was somewhere in-between, solving 20 more than ppfolio. In fact, even with only four cores, p3S-37 and p3S-39 solved 846 and 850 instances, respectively, more than the 845 ppfolio solved on eight cores.

Figure 7.2 shows similar comparisons, but on the competition split restricted to the application category, and with Plingeling as one of the competing solvers. The cactus plot on top still shows a significantly better scaling behavior of p3S-39 than both Plingeling and ppfolio. The scatter plot shows that Plingeling, not surprisingly,

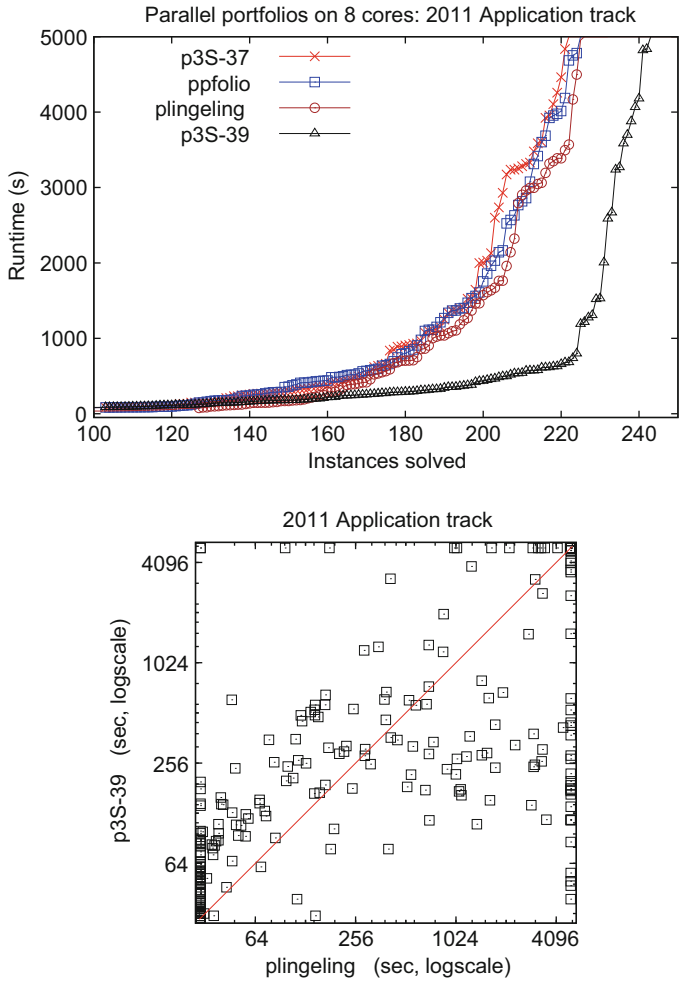

Fig. 7.2 Comparison on the 300 application category instances used in the 2011 SAT competition. *Top* Cactus plot depicting the scaling behavior of solvers. *Bottom* Per-instance comparison between Plingeling and p3S-39

is able to solve several easy instances within just a few seconds (as evidenced by the points on the bottom part of the left edge of the plot), but begins to take more time than p3S-39 on challenging instances and also times out on many more instances (shown as points on the right edge of the plot).

Overall, with eight cores, p3S-39 solved 248 application category instances, 23 more than ppfolio and 22 more than Plingeling. Moreover, p3S-37, based only on sequential constituent solvers, was only two instances shy of matching Plingeling's performance.

7.3 Chapter Summary

The chapter expanded the previously introduced 3S solver by presenting a methodology for devising dynamic parallel solver portfolios. Core methods from machine learning (nearest neighbor classification) and from optimization (integer programming and column generation) were combined to select parallel solver schedules. Different formulations of the underlying optimization problems were compared and it was found that minimizing makespan as a tie breaking rule works slightly better than minimizing CPU time. The resulting portfolio, p3S-39, was compared with the current state-of-the-art parallel solvers on instances from all SAT categories and from the application category only. It was found that p3S-39 marks a significant improvement in the ability to solve SAT instances.

Chapter 8
Dynamic Approach for Switching Heuristics

Search is an integral part of solution approaches for NP-hard combinatorial optimization and decision problems. Once the ability to reason deterministically is exhausted, state-of-the-art solvers try out alternatives that may lead to an improved (in case of optimization) or feasible (in case of satisfaction) solution. This consideration of alternatives may take place highly opportunistically, as in local search approaches, or systematically, as in backtracking-based methods.

Regardless of the scenario, efficiency could be much improved if one could effectively favor alternatives that lead to optimal or feasible solutions and a search space partition that allows short proofs of optimality or infeasibility. After all, the existence of an "oracle" is what distinguishes a non-deterministic from a deterministic Turing machine. This of course means that perfect choices are impossible to guarantee. The important insight is to realize that this is a worst-case statement. In practice, one may still hope to be able to make very good choices on average.

The view outlined above has motivated research on statistical methods to guide the search. The idea of using survey propagation in SAT [22] has led to a remarkable performance improvement of systematic solvers for random SAT instances. In stochastic offline programming [76], biased randomized search decisions are based on an offline training of the solver. A precursor of ISAC, offline training is used to associate certain features of the problem instance with specific parameter settings for the solver, whereby the latter may include the choice of branching heuristic to be used. In [99], branching heuristics for quantified Boolean formulae (QBF) were selected based on the features of the current subproblem, which led to more robust performance and solutions to formerly unsolved instances.

In this chapter, the idea of instance-specific algorithm configuration is combined with the idea of a dynamic branching scheme that bases branching decisions on the features of the current subproblem to be solved. This methodology is referred to as DASH: Dynamic Approach for Switching Heuristics.

Y. Malitsky, *Instance-Specific Algorithm Configuration*,
DOI 10.1007/978-3-319-11230-5_8

In short, the chapter follows up on the idea of choosing a branching heuristic dynamically based on certain features of the current subproblem. This idea, to adapt the search to the instance or subproblem to be solved, is by no means new. The dynamic search engine [34], for example, adapts the search heuristics based on the current state of the search. In [65], value selection heuristics for the knapsack problem were studied and it was found that accuracy of search guidance may depend heavily on the effect that decisions higher in the search tree have on the distribution of subproblems that are encountered deeper in the tree. This obviously creates a serious chicken-and-egg problem for statistical learning approaches: the distribution of instances that require search guidance affects the choice of heuristic but the latter then affects the distribution of subproblems that are encountered deeper in the tree. In [99], a method for adaptive search guidance for QBF solvers was based on logistic regression. Here, the issue of subproblem distributions was addressed by adding subproblems to the training set that were encountered during previous runs.

Inspired by the success of the approach in [99], DASH aims to boost the Cplex MIP solver. This is demonstrated on a collection of MIP problems. To this end, the branching heuristics were modified based on the features of the current subproblem to be solved. The objective of this chapter is to show that such a system can be effectively trained to improve the performance of a generalized solver for a specific application. As in previous chapters, training instances are clustered according to their features and an assignment of branching heuristics to clusters is determined that results in the best performance when the branching heuristic is dynamically chosen based on the current subproblem's nearest cluster. The approach is then examined and evaluated on the MIP solver Cplex version 12.5. These experiments show that this approach can effectively boost search performance even when trained on a rather small set of instances.

8.1 Learning Dynamic Search Heuristics

ISAC is adapted by modifying the systematic solver used to tackle the combinatorial problem in question. The following approach, highlighted in Algorithm 7, is employed.

DASH is provided with the current subproblem, the heuristic employed by the parent node, the centers of the known clusters, and the list of available heuristics. Because determining the feature can be computationally expensive and because switching heuristics at lower depths of the search tree has a smaller impact on the quality of the search, DASH chooses to switch the guiding heuristic only up to a certain depth and only at predetermined intervals, choosing the parent's heuristic in all other cases. When a decision does need to be made, the approach computes the features of the provided subproblem and determines the nearest cluster based on the Euclidean distance. In theory, any distance metric can be used here, but in practice

Algorithm 7: DASH: Branch callback

```
1: branchCallback(subproblem, parent, centers, heuristics)
2: if depth < maxDepth and depth%interval == 0 then
3:     x ← featuresComputation(subproblem)
4:     for all center c in cs do
5:         distance_i ← euclideanDistance(x, centers)
6:     end for
7:     cluster ← argmin(distance)
8:     heuristic ← heuristics_cluster
9: else
10:    heuristic ← parent.heuristic
11: end if
12: ExecuteBranching(subproblem, heuristic)
```

we found that Euclidean distance works well in the general case. In the end, DASH employs the heuristic that has been determined to be best for that cluster.

In this way, the problem has been reduced to finding a good assignment of heuristics to clusters. At this point, the problem is stated in such a way that a standard instance-oblivious algorithm configuration system can be used to find such an assignment. And as before, GGA is employed for tuning.

Note how this approach circumvents the chicken-and-egg problem mentioned in the beginning that results from the tight correlation of the distribution of subproblems encountered during search and the way branching constraints are selected. Namely, by associating heuristics and clusters *simultaneously*, it is implicitly taken into account that changes in the branching strategy result in different subproblem distributions, and that the best branching decision at a search node depends heavily on the way branching constraints will be selected further down in the tree.

That being said, the clusters themselves should reflect not only the root-node problems but also the subproblems that may be encountered during search. To this end, the training set is expanded with subproblems encountered during the runs of individual branching heuristics on the training instances to the clusters. This changes the shape of the clusters and may also create new ones. However, note that these subproblems are *not* used to learn a good assignment of heuristics to clusters, which is purely based on the original training instances. This ensures that the assignment of heuristics to clusters is not based on subproblems that will not be encountered.

8.2 Boosting Branching in Cplex for MIP

The methodology established above will now be applied to improve branching in the state-of-the-art MIP solver Cplex version 12.5 when solving MIP problems.

$$
\begin{aligned}
&\textit{Maximize} : c^T x \\
&\text{subject to} : Ax \leq b \\
&\qquad\qquad l \leq x \leq u \\
&\qquad\qquad x_j \text{ integer } \forall j \in D \text{ , where } D \subseteq \{1..n\}
\end{aligned}
$$

Here, the objective is the maximization of an objective function while maintaining the specified linear inequalities and restricting some variables to only take integer values while others are allowed to take on any real value.

To apply ISAC, instance features need to be defined for MIP problems, and various branching heuristics that our solver can choose from need to be devised.

8.2.1 MIP Features

The features have to capture as many aspects of the problems as possible without becoming too expensive to compute. To do this, we gather statistics about the problem definition of the remaining subproblem. Specifically, we compute the:

- percentage of variables in the subproblem;
- percentage of variables in the objective function of the subproblem;
- percentage of equality and inequality constraints;
- statistics (min, max, avg, std) of how many variables are in each constraint;
- statistics of the number of constraints in which each variable is used;
- depth of the branch-and-bound tree.

Wherever a feature has to do with the problem variables, we separately compute the same feature for each type of variable type, e.g., continuous, integer, and binary. Therefore, the resulting set is composed of 40 features.

8.2.2 Branching Heuristics

It is also necessary to provide a portfolio of different branching selection heuristics. The following are compared, all of which are implemented using Cplex's built-in branching methods.

8.2.2.1 Most Fractional Rounding (MF)

One of the simplest MIP branching techniques is to select the variable that has a relaxed LP solution whose fractional part is the most fractional and to round it first. The reason behind this is to make decisions on variables that deterministic analysis is least certain about. Therefore, this heuristic strives to find infeasible solutions as quickly as possible.

8.2.2.2 Less Fractional Rounding (LF)

Alternatively to MF, this technique selects the variable that has a relaxed LP solution whose fractional part is closest to an integer value and rounds it first. This is done to gently nudge the deterministic reasoning in whatever direction it is currently going, with the smallest chance of making a mistake.

8.2.2.3 Less Fractional and Highest Objective Rounding (LFHO)

This heuristic is based on the same motivation as that behind Less Fractional Branching. For each subproblem we branch on the variable for which the pair *p=(fr, obj)* is minimized (where *fr* is the fractionality and *obj* is the objective value). This means that, if we branch on a variable k in [1,n], the following property is guaranteed:

$$\forall i \in [1,n],\ fr_k < fr_i\ or\ obj_k > obj_i.$$

8.2.2.4 Most Fractional and Highest Objective Rounding (MFHO)

We use a modification of the previous approach, but this time we focus on the most fractional variables. For each subproblem we branch on the variable for which the pair *p=(fr, obj)* is maximized. In this case, the guaranteed property is:

$$\forall i \in [1,n],\ fr_k > fr_i\ or\ obj_k > obj_i.$$

8.2.2.5 Pseudocost Branching Weighted Score (PW)

This heuristic is based on the pseudocosts, numerical values that estimate the variation in objective value for rounding up or rounding down, called respectively up-pseudocost and down-pseudocost. The pseudocosts of a variable can be combined in a score function (8.2.2) that returns a numerical value. This result is used to guide the branching, for which we choose the variable that maximizes the score. Further details can be found in [3].

$$score(q^-, q^+) = (1-\mu) * min(q^-, q^+) + (\mu) * max(q^-, q^+),\quad \mu = 1/6.$$

8.2.2.6 Pseudocost Branching Product Score (P)

This approach is based on the same idea as PW. The difference lies in the score function that is now the product of the two pseudocosts.

Table 8.1 Instance distribution (percentage) in the clusterization at the root node

	1	2	3	4	5	6	7	8	9	10	11	12
Cluster 1	20	–	–	25	–	25	–	30	–	–	–	–
Cluster 2	–	45	–	–	–	14	41	–	–	–	–	–
Cluster 3	–	–	–	–	1	5	–	–	18	62	14	-
Cluster 4	–	–	49	–	–	7	–	–	–	–	–	44
Cluster 5	–	–	–	–	98	2	–	–	–	–	–	–

The problem types are: 1: airland, 2: fc, 3: GenAssignment, 4: LotSizing, 5: mik, 6: miplib2010, 7: nexp, 8: pmedcap, 9: region100, 10: region200, 11: scp, 12: SSCFLP

8.2.3 Dataset

In order to obtain a solver that works well for a generic MIP problem, instances are collected from many different datasets: *miplib2010* [63], *fc* [7], *lotSizing* [9], *mik* [8], *nexp* [6], *region* [68], and *pmedcapv*, *airland*, *genAssignment*, *scp*, and *SSCFLP* [100]. From an initial dataset of about 900 instances, those for which all our solvers timed out in 1,800 s are filtered. Also removed are the easy instances, solved entirely during the Cplex presolving or in less than 1 s by each heuristic. The final dataset is composed of 341 instances with the desired properties. From this dataset 180 are randomly selected for the training set and 161 for the testing set.

If the training data is clustered, the distribution of instances per cluster can be seen in Table 8.1. Each row is normalized to sum to 100 %. Thus, for Cluster 1, 25 % of the instances are from the *airland* dataset. From this table one can observe that there are not enough clusters to perfectly separate the different datasets into unique clusters. This, however, is not the objective. This is because we are interested in capturing similarities between instances, not in splitting benchmarks. Observe that the *region100* and *region200* instances are grouped together. Cluster 4, meanwhile, logically groups the *LotSizing* and the *SSCFLP* instances together. Finally, instances from the *miplib*, which are supposed to be an overview of all problem types, are spread across all clusters.

This clustering therefore demonstrates both that this is a diverse set of instances and that the features are representative enough to automatically notice interesting groupings.

8.3 Numerical Results

The above heuristics are embedded in the state-of-the-art MIP solver Cplex version 12.5. Note that only the branching strategy is modified through the implementation of a branch callback function. When compared with default Cplex

Table 8.2 Solving times on the testing set

Solver	Avg	Par10	% solved
BSS	315	1321	93.8
RAND 1	590	4414	77.0
RAND 2	609	5137	72.7
VBS	225	326	99.4
VBS_Rand	217	217	100

an empty branch callback is used to ensure the comparability of the approaches.[1] The empty branch callback causes Cplex to compute the branching constraint using its internal default heuristic. None of the other Cplex behavior is changed; the system uses all the standard features such as pre-solving, cutting planes, etc. Also, the search strategy, i.e., which open node to consider next, is left to Cplex. Note, however, that when Cplex dives in the tree, it considers first the first node returned by the branch callback so it does make a difference whether a variable is rounded up or down first.

With the described methodology, the main question that needs to be addressed is whether switching heuristics can indeed be beneficial to the performance of the solver. To test this, for each of the instances in the test set each of the implemented heuristics is evaluated without allowing any switching. Following this, two versions of a solver that switched between heuristics uniformly at random are also evaluated. The first solver switched between all heuristics, while the second switched only among the top four best heuristics. The results are summarized in Table 8.2.

What can be observed is that neither of the random switching heuristics performs very well by itself. However, based on the performance of the virtual best solver[2] that employs these new solvers, the performance can be further improved beyond what is possible when always sticking to the same heuristic. The question therefore now, becomes if it is possible to improve performance just by switching between heuristics randomly, can we do even better by doing so intelligently?

To answer this question, it is first necessary to set a few parameters of the solver. Particularly, to what depth should the solver be allowed to switch heuristics, and at what interval? For this, the extended dataset is clustered. This dataset includes both the original training instances and the possible observed subproblems. There are a total of 10 clusters formed. The result of projecting the feature space into two dimensions using Principal Component Analysis (PCA) [2] is presented in Fig. 8.1. Here, the cluster boundaries are represented by the solid lines, and the best heuristic for each cluster is represented by a unique symbol at its center. Also shown in these figures is the typical way in which features change as the problem is solved with a

[1] Note that Cplex switches off certain heuristics as soon as branch callbacks, even empty ones, are used, so that the entire search behavior is different.

[2] VBS is an oracle solver that for every instance always uses the strategy that results in the shortest runtime.

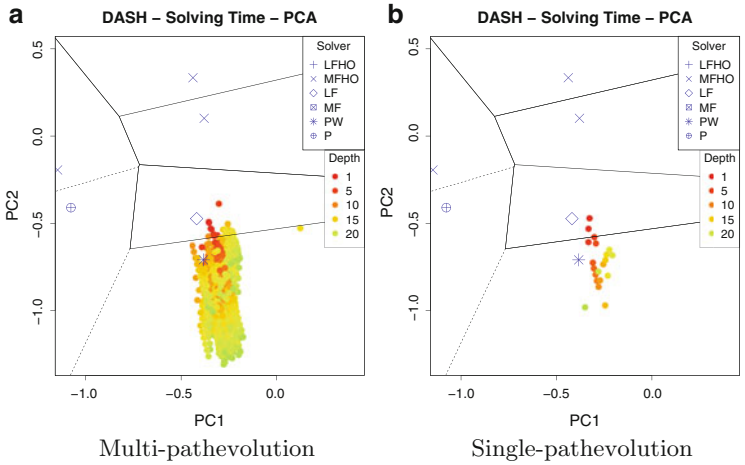

Fig. 8.1 Position of a subproblem in the feature space based on depth. (**a**) Multi-path evolution. (**b**) Single-path evolution

Table 8.3 Solving times on the testing set

Solver	Avg	Par10	% solved
BSS	315	1321	93.8
ISAC	302	1107	95.0
ISAC_filt	289	892	96.3
DASH	251	956	95.7
DASH+	255	858	96.3
DASH+filt	**241**	**643**	**98.1**
VBS	225	326	99.4
VBS_DASH	185	286	99.4

particular heuristic. The nodes are colored based on the depth of the tree, with (a) showing all the observed subproblems and (b) a single branch.

What this figure shows is that the features change gradually. This means that there is no need to check the features at every decision node. The subproblem features are therefore checked at every third node. Similarly, the figure and those like it show that using a depth of 10 is reasonable, as in most cases the nodes don't span across more than two clusters.

GGA is used to tune the parameters of DASH, computing the best heuristic for each cluster. The results are presented in Table 8.3, which compares the approach to a vanilla ISAC approach that for a given instance chooses the single best heuristic and then does not allow any switching. What can be observed is that DASH is able to perform much better than its more rigid counterpart. However, it is also necessary to allow for the possibility that switching heuristics might not be the best strategy for every instance. To cover this case, DASH+ is introduced, which first clusters the

original instances using ISAC and then allows each cluster to independently decide if it wants to use dynamic heuristic switching.

Additionally, information gain is used as a filtering technique, a method based on the calculation of entropy of the data as a whole and for each class. When applied to ISAC and DASH+ the resulting solvers are referred to respectively as ISAC_filt and DASH+filt, resulting in improvement in both cases. In particular, the resulting solver DASH+filt performs considerably better than everything else.

Table 8.3 finally shows the performance of a virtual best solver if allowed to use DASH. Observe that even though the current implementation cannot overtake VBS, future refinements to the portfolio techniques will be able to achieve performances much better than techniques that rely purely on sticking to a single heuristic.

8.4 Chapter Summary

This chapter introduced the idea of using an offline algorithm tuning tool for learning an assignment of branching heuristics to clusters of training data. During search these are used dynamically to determine a preferable branching heuristic for each encountered subproblem. This approach, named DASH, was evaluated on a set of highly diverse MIP instances. We found that the approach clearly outperforms the Cplex default and also the best pure branching heuristic considered here. While not limited by it, DASH comes very close to choosing the performance of an oracle that magically tells us which pure branching heuristic to use for each individual instance.

The chapter concluded that mixing branching heuristics can be very beneficial, yet care must be taken when learning on when to choose which heuristics, as early branching decisions determine the distribution of instances that must be dealt with deeper in the tree. This problem was solved by using the offline algorithm tuning tool GGA to determine a favorable synchronous assignment of heuristics to clusters so that instances can be solved most efficiently.

Chapter 9
Evolving Instance-Specific Algorithm Configuration

One of the main underlying themes of the previous chapters has been to demonstrate that there is no single solver that performs best across a broad set of problem types and domains. It is therefore necessary to develop algorithm portfolios, where, when confronted with a new instance, the solver selects the approach best suited for satisfying the desired objective. This process can then be further refined to intelligently create portfolios of diverse solvers through the use of instance-oblivious parameter tuners. However, all of the approaches described thus far take a static view of the learning process.

Yet there exists a class of problems where it is not appropriate to tune or train a single portfolio and then rely on it indefinitely in the future. Businesses grow, problems change, and objectives shift. Therefore, as time goes on, the problems that were used for training may no longer be representative of the types of problems being encountered. One example of such a scenario could be a business whose task it is to route a small fleet of delivery vehicles. When the business starts off, there may be only a small number of vehicles and customers, which means that the optimization problems are very small. As the business becomes increasingly successful, more customers arrive and the increased revenue allows for the purchase of more vehicles, which in turn means that a larger area can be covered. It is unlikely that a solver that has been configured to solve the original optimization problem will still be efficient at solving the new more complex problems.

This chapter, therefore, focuses on a preliminary approach that demonstrates that it is necessary to retrain a portfolio as new instances become available. This is done by identifying when the incoming instances become sufficiently different from everything observed before, thus requiring a retraining step. The chapter shows how to identify this moment and how by doing so efficiently we can create an improved portfolio over the train-once methodology.

Y. Malitsky, *Instance-Specific Algorithm Configuration*,
DOI 10.1007/978-3-319-11230-5_9

9.1 Evolving ISAC

Like other existing portfolio approaches, ISAC is a one-shot learning approach. Once the training data has been clustered and a solver is selected for each cluster, no new learning is performed. This means that even as the learned portfolio interacts with and is repeatedly used to solve new instances, the influx of new information is ultimately ignored. It also means that if the structure of the problems changes over time, the solvers that have been trained may no longer be suitable. Moreover, as time progresses, new features may be discovered that can have a dramatic impact on the quality of the clustering methods. Furthermore, the set of available solvers can change over time, which can also have a drastic impact on the current portfolio.

Of course, one solution is to re-launch ISAC from scratch after each such change. In practice, however, this can be very computationally expensive. For example, a 50 s timeout for 1,000 instances can take 72 CPU days to train a portfolio [77]. In the case of a portfolio approach, every solver would need to be run on every instance in the training set. Alternatively, for the case with a single parameterized solver, new parameterizations would need to be tuned, a process that can take days or even weeks depending on the number of training instances, parameters, and the timeout. The Evolving ISAC (EISAC) approach is aimed at tackling this by continuously evaluating the current portfolio and providing ways to continuously improve it by taking advantage of a growing set of available instances, an enriched feature set, and more up-to-date set of available solvers.

Let $\mathcal{I}_t$, $\mathcal{F}_t$, and S_t denote the set of instances, the feature set, and the set of solvers known at time t and let $\mathcal{C}_t$ denote the clustering of instances at time t. We assume that each time period is associated with one or more changes in at least one of these sets. Following this notation $\mathcal{I}_0$, $\mathcal{F}_0$, and S_0 denote the initial sets of training instances, features, and solvers, respectively.

EISAC is divided into two phases: initial phase and evolving phase. Given an initial set of instances, $\mathcal{I}_0$, and an initial set of features, $\mathcal{F}_0$, EISAC finds an initial set of clusters $\mathcal{C}_0$ and determines the best solver from set $\mathcal{S}_0$ for each cluster. This process is summarized in Fig. 9.1.

In the evolving phase, EISAC is tasked to solve an influx of new instances. As these instances come in, they are assigned to a cluster and solved by the appropriately assigned solver. This is illustrated in Fig. 9.2. Once the instance is solved, EISAC compares the original clustering with the one obtained by re-clustering all the instances. If the clusterings are sufficiently different, we accept the new clustering and retrain our portfolio approach. If the two clusterings are

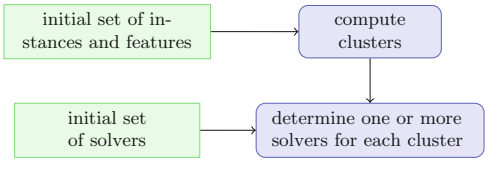

Fig. 9.1 Initial phase of EISAC which is performed offline

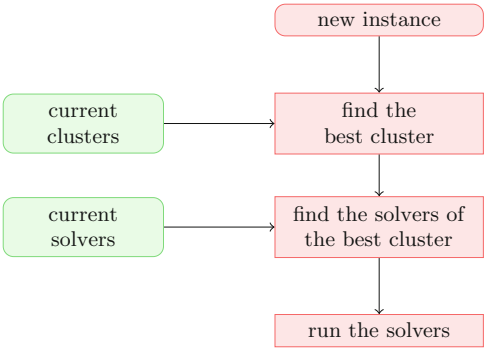

Fig. 9.2 EISAC solving new instances

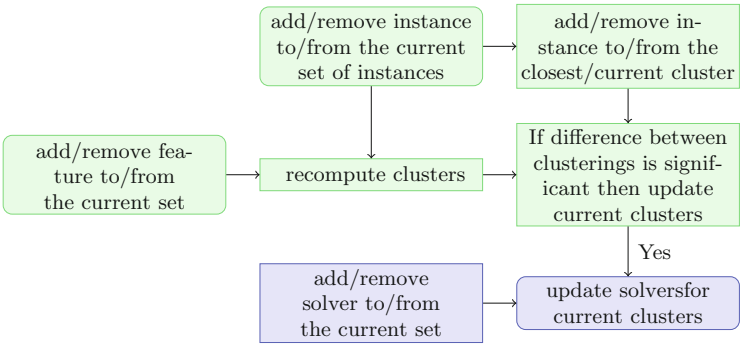

Fig. 9.3 Evolving the ISAC portfolio

relatively identical, then we stick to the portfolio we have used up to now. This process is summarized in Fig. 9.3.

We therefore see that if our original training set was representative of all the new instances we observed, then the new instances would fall neatly into the clusters we found during the first stage. In such a case, we would never need to retrain our portfolio. If, on the other hand, the instances changed over time, this approach aims to automatically identify the best time to retrain the portfolio. Depending on the time we have to retrain a portfolio or how accurate we want to be, we can modify the threshold at which we consider two clusterings to be similar enough. For the experiments presented in this chapter, we use the adjusted Rand index to make this assessment.

As presented, there are a number of decisions that govern how EISAC behaves. These include answering questions like, how many new instances need to be solved before we consider re-clustering? How do we handle the introduction of new solvers in our portfolio, or solvers no longer being available? If we have an infinite influx of new instances, we cannot keep information about all of them in memory as our

training set, so how many instances should we keep track of for re-clustering? We explain and discuss some of these issues below.

9.1.1 Updating Clusters

Updating the clusters of ISAC can potentially be very expensive if we need to train or select a new solver for each cluster. It is therefore imperative to minimize the number of times this operation is performed while still maintaining good expected performance. There are two scenarios for which we might wish to consider a new clustering:

- A new instance is made available and added to the current set of instances, and a number of instances are removed if the total number of instances exceeds the maximum number of instances that can be maintained at any time.
- A new feature is added to the current set of features or if an existing feature is removed.

In each of these cases one would like to determine whether the existing clustering is still appropriate or whether it should be modified.

Let δ be the time difference between the last time EISAC was activated and the current time. Given a value δ, EISAC recomputes the partition of the instances and compares it with the current partition. The following describes one way of comparing the similarity between two partitions.

The Rand index [52, 95] or Rand measure (named after William M. Rand) is a measure of the similarity between two data clusterings. Given a set of instances $\mathcal{I}_t$ and two partitions of $\mathcal{I}_t$ to compare, $X = \{X_1, \ldots, X_k\}$, a partition of $\mathcal{I}_t$ into k subsets, and $Y = \{Y_1, \ldots, Y_s\}$, a partition of $\mathcal{I}_k$ into s subsets, the Rand index is defined as follows:

- Let N_{11} denote the number of pairs of instances in $\mathcal{I}_t$ that are in the same set in X and in the same set in Y.
- Let N_{00} denote the number of pairs of instances in $\mathcal{I}_t$ that are in different sets in X and in different sets in Y.
- Let N_{10} denote the number of pairs of instances in $\mathcal{I}_t$ that are in the same set in X and in different sets in Y.
- Let N_{01} denote the number of pairs of instances in $\mathcal{I}_t$ that are in different sets in X and in the same set in Y.

The Rand index, denoted by R, is defined as follows:

$$R = \frac{N_{11} + N_{00}}{N_{11} + N_{00} + N_{10} + N_{01}} = \frac{2(N_{11} + N_{00})}{n * (n-1)}.$$

Intuitively, $N_{11} + N_{00}$ can be considered as the number of agreements between X and Y and $N_{10} + N_{01}$ as the number of disagreements between X and Y.

To correct for chance, the adjusted Rand index (ARI) then normalizes the Rand index to be between −1 and 1 by:

$$\texttt{ARI} = \frac{R - ExpectedIndex}{MaxIndex - ExpectedIndex}.$$

If X is the current partition and Y is the new partition and if the adjusted Rand index is less than some chosen threshold, denoted by λ, then we replace the current partition of the instances with the new partition. If the current partition is replaced by the new partition, then we may need to update the solvers for one or more clusters of the new partition. The experimental section will demonstrate results with a variety of thresholds.

9.1.2 Updating Solver Selection for a Cluster

For EISAC, the optimization problem is solved by the methods described below for finding the best solver for each cluster.

Let x_{ij} be a Boolean variable that determines if solver $j \in \mathcal{S}_t$ is chosen for instance i of some cluster C. Let T_{ij} denote the time required for solving instance i using solver $j \in S_i$. Let y_j be a Boolean that denotes whether solver j is selected for the cluster C. For each instance $i \in C$, exactly one solver is selected:

$$\forall_{i \in C} : \quad \sum_{j \in \mathcal{S}_t} x_{ij} = 1. \tag{9.1}$$

The solver j is selected if it is chosen for any instance i:

$$\forall_{i \in C} \forall_{j \in \mathcal{S}_t} : x_{ij} \Rightarrow y_j. \tag{9.2}$$

The number of selected solvers should be equal to 1:

$$\sum_{j \in S_t} y_j = 1. \tag{9.3}$$

The objective is to minimize the time required to solve all the instances of the cluster, i.e.,

$$\min \sum_{i \in C} \sum_{j \in S_t} T_{ij} \cdot x_{ij}. \tag{9.4}$$

Given the set of solvers $\mathcal{S}_t$ at time t, a cluster $C \in \mathcal{C}_t$, and the matrix T, computeBestSolvers$(C, \mathcal{S}_t, T)$ denotes the best solver obtained by solving the MIP problem composed of constraints (9.1)–(9.3) and the objective function

(9.4). In the following we describe four cases when EISAC might update the best solver for a cluster.

9.1.2.1 Removing Solvers

It may be that an existing solver is no longer available now, which could happen when a solver is no longer supported by the developers, or when a new release is made; then, one would like to discard the previous version and update it with the new version. If the removed solver is used by any cluster, then one can re-solve the above optimization problem for finding the current solvers.

9.1.2.2 Adding Solvers

When a new solver s is added to the set of solvers $\mathcal{S}_t$ at time t, it can have an impact on the current portfolio. One way is to reconstruct the portfolio by running the solver s for all the instances in $\mathcal{I}_t$, which could be time-consuming. Therefore, EISAC selects a sample from each cluster and runs the solver s for only those samples. If adding the new solver to $\mathcal{S}_t$ improves the total execution time for solving the sample instances, then EISAC computes the runtime of solver s for all the instances of the corresponding cluster. Otherwise, it avoids running the solver s for the remaining instances of the cluster. In EISAC, the sample size for a cluster C is set to $(|\mathcal{I}_0|/|\mathcal{I}_t|) * |C|$. This allows EISAC to maintain the runtimes of all the solvers in $\mathcal{S}_t$ for at least $|\mathcal{I}_0|$ number of instances.

9.1.2.3 Removing Instances

If the clustering changes because of our removing instances or because of change in the feature set, EISAC updates the current set of best solvers for each cluster by re-solving the above optimization problem.

9.1.2.4 Adding Instances

If the current clustering is replaced with the new clustering because of our adding new instances, then EISAC recomputes the best solver for one or more clusters of the new clustering. In order to do so, one approach is to compute the run-times of all the solvers for all the new instances and then recompute the best solver for each cluster. However, it is more desirable for the best solver for a cluster to be computed without our computing the run-times of all the solvers for all the newly added instances. For this, a lazy approach can be used. The idea is to predict the expected best run-time of each solver for one or more instances whose run-times are unknown and use them to compute the best solver for a cluster. If the newly

Algorithm 8: updateBestSolvers(C)

1: **loop**
2: $\mathcal{A}_t \leftarrow \{i | i \in \mathcal{I}_t \wedge |\mathcal{S}_{ti}| = |\mathcal{S}_t|\}$
3: $\forall i \in \mathcal{A}_t : T_{ir_{i1}} \leq \ldots \leq T_{ir_{i|\mathcal{S}_t|}}$
4: $\forall p < |\mathcal{S}_t| : e_p \leftarrow \frac{1}{|\mathcal{A}_t|} \sum_{i \in \mathcal{A}_t} \frac{T_{ir_{i1}}}{T_{ir_{ip}}}$
5: $C_u \leftarrow C - \mathcal{I}_{ta}$
6: **for all** $i \in C_u$ **do**
7: $p \leftarrow |\mathcal{S}_t| - |\mathcal{S}_{ti}| + 1$
8: $b \leftarrow \arg\min_{j \in \mathcal{S}_{ti}}(T_{ij})$
9: $\forall j \in \mathcal{S}_t - \mathcal{S}_{ti} : \quad T_{ij} \leftarrow e_p \times T_{ib}$
10: **end for**
11: $N_C \leftarrow$ `computeBestSolvers`$(C, \mathcal{S}_t, T)$
12: **if** $N_C = B_C$ **then**
13: **return** B_C
14: **end if**
15: **for all** $i \in C_u \wedge j \in N_C - S_{ti}$ **do**
16: $T_{ij} \leftarrow$ `computeRuntime`(i, j)
17: $S_{ti} \leftarrow S_{ti} \cup \{j\}$
18: **end for**
19: $B_C \leftarrow N_C$
20: **end loop**

computed best solver is the same as the previously known best solver for a cluster, then the task of solving new instances using many solvers is avoided. Otherwise, the current best solver is updated and the actual run-times, which are unknown for some of the instances, are computed using the newly computed best solver. This step is repeated until the newly computed best solvers are the same as the previously known best solver.

Algorithm 8 presents pseudocode for computing the best solver for a cluster C lazily. Let $\mathcal{S}_{ti}$ be the set of solvers at time t for which the run-times are known for solving an instance i. Assume that $\mathcal{S}_{ti}$ is initialized to a some solver which is used in the online phase to solve an instance i. Let $\mathcal{A}_t$ be the set of instances for which the run-times of all the solvers are known at time t (Line 2). Let r_{ip} denote the pth best solver for instance $i \in \mathcal{A}_t$ based on run-times (Line 3). Let e_p denote the average ratio between the runtimes of the best solver and the pth best solver for each instance $i \in \mathcal{A}_t$ (Line 4). Let $C_u \subseteq C$ be the set of instances for which the run-times of one or more solvers in the set $\mathcal{S}_t$ are unknown (Line 5). The expected best run-time for a solver $j \in \mathcal{S}_t - \mathcal{S}_{ti}$ for an instance $i \in C_u$ is computed as described below (Lines 6–9). If it is assumed that $\mathcal{S}_{ti}$ is the set of the $|\mathcal{S}_{ti}|$ worst solvers for instance $i \in C_u$, then the runtime of the best solver, b, of $\mathcal{S}_{ti}$ would have the pth best runtime of all the solvers, where $p = |\mathcal{S}_t| - |\mathcal{S}_{ti}| + 1$, and the expected best run-time of a solver in $j \in \mathcal{S}_t - \mathcal{S}_{ti}$ would be $T_{ib} \cdot e_p$. Notice that different assumptions on the value of p would result in different performances of EISAC.

Let N_C be the newly computed best solvers using expected best run-times (Line 10). Let B_C be the previously known best solver for the cluster C. If N_C is the

same as B_C, then the previously known solver is still the best solver for cluster C even when the known run-times are assumed to be the worst, and the computation of the run-times of $\mathcal{S}_t - \mathcal{S}_{ti}$ solvers are avoided for each instance $i \in C_u$ (Lines 11–12). If N_C is different from B_C, then the actual run-time of each solver $j \in N_C - \mathcal{S}_{ti}$ is computed for each instance $i \in C_u$, denoted by `computeRuntime`(i, j), and the best solver for cluster C is updated to N_C (Lines 13–16).

9.2 Empirical Results

In this section we demonstrate the effectiveness of using EISAC on SAT as well as MaxSAT instances.

9.2.1 SAT

For the first line of experiments we use the SAT portfolio data made available by the SATzilla team after the 2011 SAT Competition [1]. This dataset provides the runtimes of 31 top-tier SAT solvers with a 1,200 s timeout on over 3,000 instances spread across the Random, Crafted and Industrial categories. After filtering out the instances where every solver times out, we are left with 2,524 instances. For each of these instances the dataset also provides all of the known SAT features, but this chapter focuses on the 52 standard features [85] that do not rely on local search probing.

We use this dataset to simulate the scenario where instances are made available one at a time. Specifically, we start with a set of $\mathcal{I}_0$ instances for which we know the performance of every solver. Based on this initial set, we generate our initial clusters and select the solver that minimizes the PAR10 score of each cluster, PAR10 being the standard penalized average measurement in SAT where when a solver times out it is penalized as having taken 10 times the timeout to complete. We then add δ instances to our dataset, evaluate them with the current portfolio, and then determine whether we should retrain. Two thresholds for the adjusted Rand index are used, 0.5 and 0.95. Simulating the scenario where one can only keep a certain number of instances for the retraining, once we add the δ new instances, we also remove the oldest δ instances.

First, we consider the scenario where all the instances are shuffled and come randomly. We then also consider an ordering on the data, where first we iterate through the industrial instances, followed by the crafted, and finally the instances that were randomly generated. This last experiment is meant to simulate the case where instances change over time. This is also the case where traditional portfolio approaches would fail because eventually they are tasked to solve instances they have never observed during training.

Table 9.1 Comparison of performance of ISAC and EISAC on shuffled and ordered datasets using 200 or 500 training instances

Shuffled	BS		ISAC c50	EISAC c50-λ0.5	EISAC c50-λ0.95	ISAC c100	EISAC c100-λ0.5	EISAC c100-λ0.95	VBS
200	Solved	1760	1776	1753	1759	1776	1752	1752	2324
	% solved	75.7	76.0	75.4	75.7	76.0	75.4	75.4	100
	PAR10	3001	2923	3037	3006	2923	3038	3038	75.2
	# train	1	1	275	329	1	166	166	–
500	Solved	1532	1548	1548	1539	1548	1548	1544	2024
	% solved	75.7	76.4	76.4	76.0	76.4	76.4	76.3	100
	PAR10	3004	2912	2912	2962	2912	2912	2935	74.82
	# train	1	1	1	674	1	1	104	–
Ordered	BS		ISAC c50	EISAC c50-λ0.5	EISAC c50-λ0.95	ISAC c100	EISAC c100-λ0.5	EISAC c100-λ0.95	VBS
200	Solved	1078	1078	1725	1793	1078	1741	1741	2324
	% solved	46.3	46.3	74.2	77.2	46.3	74.9	74.9	100
	PAR10	6484	6484	3160	2821	6484	3084	3084	70.42
	# train	1	1	49	160	1	9	9	–
500	Solved	791	795	1261	1606	817	817	1373	2024
	% solved	39.1	39.3	62.3	79.3	40.4	40.4	67.8	100
	PAR10	7357	7334	4578	2556	7205	7205	3910	70.79
	# train	1	1	4	611	1	1	97	–

We set the minimum cluster size to be either 50 or 100 and the adjusted Rand index to either 0.5 or 0.95

Table 9.1 presents the first test case, where instances come from a shuffled dataset. This is the typical case observed in competitions, where a representative set of the test data is available for training. The table presents the performance of a portfolio which has been given 200 or 500 training instances. The single best solver (BS) chooses a single solver during training and then always uses it during the test phase. Alternatively, the virtual best solver (VBS) is an oracle portfolio that for every instance always runs the best solver. The VBS represents the limit of what can be achieved by a portfolio. We also evaluate ISAC-c50 and ISAC-c100, trained with a minimum respectively of 50 and 100 instances in each cluster. Note that in this setting ISAC is performing better than BS. It is also important to note here that in the 2012 SAT Competition, the difference between the winning and second placed single engine solver was three instances and was only eight instances between the top four solvers. Therefore the improvement of 16 instances when training on 500 instances is significant. When compared to ISAC on this shuffled data, we see that EISAC performs comparably to ISAC, although it requires significantly more training sessions. For each version of EISAC in the table we present the minimum cluster size and the adjusted Rand index threshold. So EISAC-c100-λ0.95 has clusters with at least 100 instances and retrains as soon as the adjusted Rand index drops below 0.95.

This comparable performance on shuffled data is to be expected. As the data is coming randomly, the initial training data is representative enough to capture the diversity. And even if the clusters change a little over time, the basic assignment of solvers to instances doesn't really change. Note that the slight degradation between the higher threshold in EISAC-c100 for 500 training instances can likely be attributed to overtuning (or overfitting) in the numerous re-training steps. Also note that the lower performance for 500 training instances is misleading, since by adding 300 instances to our training set, we are removing 300 instances from the test set.

The story becomes significantly different if the data is not shuffled, as is the case at the bottom of Table 9.1. Here we see that the clusters and solvers chosen by ISAC initially are ill equipped to solve the future instances. EISAC, on the other hand, is able to adapt to the changes and outperform ISAC by almost a factor of 2 in terms of the instances solved. What is also interesting is that for the case of 500 training instances and small clusters, this performance is achieved with only four re-training steps.

Table 9.2 examines the effectiveness of this training technique. Instead of computing the time of every solver on every training instance during re-tuning, we lazily fill in this data until we converge on the expected best solver for a cluster. We call this approach EISAC+. For simplicity, here we only present a comparison on the ordered dataset and for algorithms tuned with a minimum cluster size of 50. What we observe is that while performance is maintained with this lazy selection, we cut down the number of evaluations we need to 80 % and occasionally to as low as 50 %. This means that we can potentially speed up each training stage by a factor of 2 while still maintaining nearly identical performance.

Finally, Table 9.3 examines the effectiveness of EISAC and EISAC+ on a dataset where instances become progressively harder. Specifically, we order the 2,524 instances at our disposal according to the average runtime of all solvers. The first 200 (500) are used for training, and each instance is on average harder than the last. From the table we see that the regular version of ISAC struggles with this dataset, performing worse than the best single solver. This suggests that the structure of the easier instances is different from that of the harder instances. If we allow our

Table 9.2 Comparison of performance on ordered dataset using an approximation learning technique

			ISAC	EISAC	EISAC+	
		BS	c50	c50-λ0.5	c50-λ0.5	VBS
200	Solved	1078	1078	1725	1671	2324
	PAR10	6484	6484	3160	3440	70.42
	# train	1	1	49	44	–
	% eval	100	100	100	59.5	–
500	Solved	791	795	1261	1264	2024
	PAR10	7357	7334	4578	4561	70.79
	# train	1	1	4	3	–
	% eval	100	100	100	83.8	–

Table 9.3 SAT: Comparison of performance of ISAC, EISAC, and EISAC+ on data where instances become progressively harder

Easy to hard		BS	ISAC c50	EISAC c50-λ0.5	EISAC c50-λ0.95	EISAC+ c50-λ0.5	EISAC+ c50-λ0.95	VBS
200	Solved	1060	1035	1690	1823	1698	1741	2324
	% solved	45.6	44.5	72.7	78.4	73.1	74.9	100
	PAR10	6621	6746	3383	2710	3343	3122	82.51
	# train	1	1	58	155	57	185	–
	% eval	100	100	100	100	83.8	83.3	–
500	Solved	1410	1057	1485	1526	1400	1532	2024
	% solved	69.7	52.2	73.4	75.4	69.2	75.7	100
	PAR10	3753	5850	3322	3075	3811	3041	94.4
	# train	1	1	4	611	3	663	–
	% eval	100	100	100	100	87.8	83.1	–

Training set is composed of either 200 or 500 instances, the minimum cluster size is set to be 50, and the adjusted Rand index is set to either 0.5 or 0.95

portfolio to adapt to the changes, it is able to perform significantly better than the best single solver. Furthermore, EISAC+ is able to perform comparably to its regular counterpart but with fewer evaluations at each training stage.

9.2.2 MaxSAT

Additionally, we present how the Evolving ISAC approach behaves on the MaxSAT dataset, employing the instances from the 2012 MaxSAT Competition. This competition had four major categories: MaxSAT, Partial MaxSAT, Weighted MaxSAT, and Partial Weighted MaxSAT. Combining these results in 2,681 instances. What differentiates this merged dataset from the one we had for SAT, is that the solvers used for each of the different types of instances are different. So while a solver can perform very well on one category, it might completely fail to read in instances of another type. In the case presented here, whenever a solver is not able to solve an instance, it is marked as if it timed out.

For the solvers we employ 14 top-tier MaxSAT solvers with a 2,100 s timeout. Meanwhile the features used are based on the original base SAT features plus five features examining the weights of the soft clauses, namely the percentage of soft clauses, and the mean, standard deviation, minimum and maximum of the weights of these soft clauses.

The experiments explore three scenarios. First, we take a look at the standard competition, train-once, the situation where the training set is representative of the data to be seen in the test set. We can see under the rows labeled “Shuffled” in Table 9.4 that the adaptive approach, while better than the best single solver, is not better than the plain vanilla version of ISAC. This, however, quickly changes if

Table 9.4 MaxSAT: Comparison of performance of ISAC, EISAC, and EISAC+ on shuffled data (shuffled), ordered by category (ordered), and ordered by difficulty (easy to hard) datasets using 200 or 500 training instances

			ISAC	EISAC	EISAC	EISAC+	EISAC+	
Shuffled	BS		c50	c50-λ0.5	c50-λ0.95	c50-λ0.5	c50-λ0.95	VBS
200	Solved	1472	1802	1540	1544	1502	1479	2260
	% solved	59.3	72.6	62.1	62.2	60.5	59.6	91.1
	PAR10	8187	5675	7640	7609	7949	8132	1831
	# train	1	1	908	1266	872	1259	–
	% eval	100	100	100	100	95.2	93.4	–
500	Solved	1298	1724	1564	1555	1474	1475	1986
	% solved	59.5	79.0	71.7	71.3	67.6	67.6	91.1
	PAR10	8149	4252	5722	5800	6541	6532	1836
	# train	1	1	495	1811	512	1843	–
	% eval	100	100	100	100	93.2	83.8	–
Ordered by			ISAC	EISAC	EISAC	EISAC+	EISAC+	
category	BS		c50	c50-λ0.5	c50-λ0.95	c50-λ0.5	c50-λ0.95	VBS
200	Solved	1466	1087	1830	1958	1741	1796	2268
	% solved	59.1	43.8	73.8	78.9	70.2	72.4	91.4
	PAR10	8236	11275	5325	4287	6033	5584	1764
	# train	1	1	266	557	255	529	–
	% eval	100	100	100	100	50.4	48.9	–
500	Solved	1247	1131	1314	1555	1261	1422	2049
	% solved	57.1	51.9	60.2	71.3	57.8	65.2	93.9
	PAR10	8607	9656	7990	5789	8480	7006	1245
	# train	1	1	145	871	155	866	–
	% eval	100	100	100	100	58.3	56.9	–
Easy			ISAC	EISAC	EISAC	EISAC+	EISAC+	
to hard	BS		c50	c50-λ0.5	c50-λ0.95	c50-λ0.5	c50-λ0.95	VBS
200	Solved	837	875	1520	1604	1509	1544	2241
	% solved	33.7	35.2	61.2	64.6	60.8	62.2	90.3
	PAR10	13297	12987	7812	7137	7902	7539	1988
	# train	1	1	266	557	263	532	–
	% eval	100	100	100	100	83.6	82.5	–
500	Solved	539	687	1401	1403	1260	1352	1941
	% solved	24.7	31.5	64.2	64.3	57.8	62.0	89.0
	PAR10	15103	12742	7226	7207	8519	7674	2262
	# train	1	1	145	871	123	854	–
	% eval	100	100	100	100	88.0	88.4	–

We set the minimum cluster size to be 50 and the adjusted Rand index to either 0.5 or 0.95

the data is introduced first from the MaxSAT (MS) category, followed by Partial MaxSAT (PMS), then Weighted MaxSAT (WMS), and finally Partial Weighted MaxSAT (PWMS). These results are presented in the rows under "Ordered by Category" in Table 9.4.

These experiments exemplify why it is necessary to continuously refine one's portfolio approach. The regular ISAC approach performs worse than the single best solver, since it is unable to generalize the portfolio it has learned to instances that it has never seen before. The EISAC approach, however, can cope with the changes. It is also important to note that EISAC+ is able to achieve nearly the same performance as EISAC, with half the number of evaluations during the tuning stage.

The third set of rows in Table 9.4, labeled "Easy to Hard," refers to a dataset where the instances arrive in order of difficulty, from easy to hard. Here we define "easy" as an instance whose average runtime over all solvers in our portfolio is the smallest. We again see that the regular version of ISAC struggles to find a portfolio that generalizes to the harder instances. EISAC, on the other hand, can double the number of instances solved.

9.3 Chapter Summary

This chapter presented the importance of an evolving portfolio approach when handling datasets that can change over time. Specifically, it presented EISAC, a portfolio approach that solves twice as many instances as its unmodified counterpart ISAC. Further shown was how a lazy training method can help to significantly reduce the number of solver/instance combinations that need to be evaluated before the best solver for a cluster is selected.

Chapter 10
Improving Cluster-Based Algorithm Selection

There are a number of benefits to the ISAC methodology which have been touched upon in the previous chapters. In addition to those, it is also a transparent approach. Each cluster can be identified and analyzed for what makes it different from all the others. What are the features that are most different for a cluster that help differentiate it from its neighbors? Such analysis can then motivate understanding of why a particular solver performs well on one group of instances and not another. Identifying clusters where all solvers perform poorly can help direct research to developing novel solvers with a specific target benchmark. Clusters can also help identify what regions of the problem space have not yet been properly investigated.

There is of course an ever increasing number of new instance-specific algorithm portfolios being introduced that continuously outperform the previous generations. The 3S solver presented in Chap. 6 and the parallel solver from Chap. 7 are two examples addressed in this book. SATzilla 2012 [127] and CSHC [75] are solvers that did very well in the 2012 and 2013 SAT Competitions. Yet as these portfolios become better and better at predicting the best solver for an instance they also become increasingly more complex. The 2012 version of SATzilla requires multiple trees that are trained to distinguish between every pair of solvers. But such a representation makes it hard to see similarities between instances. CSHC uses trees to split instances so that each child has a better agreement on the preferred solver than its parent. This strategy can be highly informative in practice, yet in order to boost its performance, CSHC relies on a forest of such trees.

This chapter focuses on presenting how the ISAC methodology can be improved without sacrificing its transparency. Specifically, the chapter will review some of the assumptions that have been made in the previous chapters and show whether they have been justified. It will then show how taking into account the performances of the solvers in a portfolio can help guide the clustering process.

Y. Malitsky, *Instance-Specific Algorithm Configuration*,
DOI 10.1007/978-3-319-11230-5_10

10.1 Benchmark

This chapter, like many that came before it, will rely on the SAT domain for its benchmark. As before, this choice is made due to the abundance of diverse solvers and instances in the field, as well as a highly polished set of features.

10.1.1 Dataset and Features

In this chapter we take all instances from the 2002 to 2012 competitions, inclusive, which results in 2,140 in the Random categories, 735 in the Crafted categories, and 1,098 in the Industrial categories. Additionally, we add a dataset that includes all the 4,243 instances.

As mentioned in the previous chapter, algorithm selection utilizes a set of features in order to distinguish between instances. In this chapter, we use an expanded feature set that was developed by the UBC Group [126] in 2008 and now consists of the 113 continuous values listed below.

Problem Size Features:

1-2 **Number of variables and clauses in original formula**: denoted by v and c, respectively

3-4 **Number of variables and clauses after simplification with** SATElite: denoted by v' and c', respectively

5–6 **Reduction of variables and clauses by simplification**: (v-v')/v' and (c-c')/c'

7 **Ratio of variables to clauses**: v'/c'

Variable-Clause Graph Features:

8–12 **Variable node degree statistics**: mean, variation coefficient, min, max, and entropy

13–17 **Clause node degree statistics**: mean, variation coefficient, min, max, and entropy

Variable Graph Features:

18–21 **Node degree statistics**: mean, variation coefficient, min, and max

22–26 **Clustering Coefficient**: mean, variation coefficient, min, max, and entropy

Clause Graph Features:

27–31 **Node degree statistics**: mean, variation coefficient, min, max, and entropy

32–36 **Clustering Coefficient**: mean, variation coefficient, min, max, and entropy

Balance Features:

37–41 **Ratio of positive to negative literals in each clause**: mean, variation coefficient, min, max, and entropy

42–46 **Ratio of positive to negative occurrences of each variable**: mean, variation coefficient, min, max, and entropy

47–49 **Fraction of unary, binary and ternary clauses**

Proximity to Horn Formula:

50 **Fraction of Horn clauses**

51–55 **Number of occurrences in a Horn clause for each variable**: mean, variation coefficient, min, max, and entropy

Local Search Probing Features, based on 2 s of running each of SAPS and GSAT:

56–65 **Number of steps to the best local minimum in a run**: mean, median, variation coefficient, and 10th and 90th percentiles

66–69 **Average improvement to best in a run**: mean and coefficient of variation of improvement per step to best solution

70–73 **Fraction of improvement due to first local minimum**: mean and variation coefficient

74–77 **Coefficient of variation of the number of unsatisfied clauses in each local minimum**: mean and variation coefficient

Clause Learning Features (based on 2 s of running Zchaff_rand):

78–86 **Number of learned clauses**: mean, variation coefficient, min, max, and 10, 25, 50, 75, and 90 % quantiles

87–95 **Length of learned clauses**: mean, variation coefficient, min, max, and 10, 25, 50, 75, and 90 % quantiles

Survey Propagation Features:

96–104 **Confidence of survey propagation**: For each variable, compute the higher of $P(true)/P(false)$ and $P(false)/P(true)$. Then compute statistics across variables: mean, variation coefficient, min, max, and 10, 25, 50, 75, and 90 % quantiles

105–113 **Unconstrained variables**: For each variable, compute $P(unconstrained)$. Then compute statistics across variables: mean, variation coefficient, min, max, and 10, 25, 50, 75, and 90 % quantiles

These features can be broken into two categories: deterministic structural and stochastic probing. The deterministic features are those labeled 1 through 55, and focus on the facts that can be computed from the problem definition alone. This includes looking at the number of variables and clauses, the statistics about how often variables are negated, and the statistics about how many variables are in a clause. Here, the variable clause graph refers to a graph where each variable and each clause is represented as a node, with an edge signifying that a variable belongs to a particular clause. In the variable graph, the nodes represent variables, with an unweighted edge existing if two variables appear in the same clause. Similarly, a clause graph has an edge between two nodes if two clauses have the same variable.

While the deterministic features provide good information and are typically very quick to compute, the probing features are designed to give insight into how solvers might behave when tackling an instance. These features are therefore computed by running existing solvers for a short execution and gathering statistics upon completion.

Finally, the runtimes of 29 of the most current SAT solvers have been recorded; many of them have individually shown good performance in past competitions.

- `clasp-2.1.1_jumpy`
- `clasp-2.1.1_trendy`
- `ebminisat`
- `glueminisat`
- `lingeling`
- `lrglshr`
- `picosat`
- `restartsat`
- `circminisat`
- `clasp1`
- `cryptominisat_2011`
- `eagleup`
- `gnoveltyp2`
- `march_rw`
- `mphaseSAT`
- `mphaseSATm`
- `precosat`
- `qutersat`
- `sapperlot`
- `sat4j-2.3.2`
- `sattimep`
- `sparrow`
- `tnm`
- `cryptominisat295`
- `minisatPSM`
- `sattime2011`
- `ccasat`
- `glucose_21`
- `glucose_21_modified`.

Each of the solvers was run on every instance with a 5,000 s timeout. Instances that could not be solved by any solver in the allotted time were removed. Further removed were the easy instances, where at least 10 solvers could finish within 15 s. This filtering was done because, if no solver could finish, then the instance could provide no useful information, while the easy instances are trivial regardless of what is done. This way, the evaluation focused on only those instances where the decision of one solver over another had visible consequences. This meant that the final dataset comprised of 1,949 random, 363 crafted, and 805 industrial instances, i.e., 3,117 instances in total.

10.2 Motivation for Clustering

One of the underlying assumptions behind any algorithm selection approach has been that there is no single solver that can provide the best performance on every instance. Implicitly, this has been shown in the previous chapters, but Table 10.1 aims to further corroborate this point. The table shows the frequency with which a solver is deemed best over the dataset. Specifically, it utilizes the SAT datasets that were described in the previous section: RAND, HAND, INDU, and ALL. In this table, BSS is the best single solver, or the solver that provides the best performance on average. Meanwhile, VBS is the performance of an oracle approach that for every instance selects the best algorithm.

Table 10.1 clearly shows that the solver deemed best is rarely the best solver for any particular instance. This is especially evident when looking at the ALL dataset, where mPhaseSAT is never the best solver. The necessity for algorithm selection is even more pronounced when looking at the difference in the number of solved instances. For the ALL dataset, mPhaseSAT is able to solve 2,697 instances. This is not bad considering that the runner up is an alternate parameterization, mPhaseSATm, which solves 2,605 instances, and CCASat, ranked third best, can only solve 1,861 instances. However, the VBS for this dataset solves 3,676 instances. This means that without any modifications to existing solvers, it is possible to improve performance by over 20 % just by choosing the correct solver for the instance at hand.

Table 10.1 Frequencies of solver selections for VBS and the best single solver (BSS)

RAND	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
BSS	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	100	–	–
VBS	–	–	–	–	–	–	–	–	–	–	–	17	6	**19**	2	–	–	–	–	–	3	**18**	5	–	–	4	**24**	–	–
HAND	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
BSS	–	–	–	–	–	–	–	–	–	–	–	–	–	–	100	–	–	–	–	–	–	–	–	–	–	–	–	–	–
VBS	–	3	1	2	2	1	**7**	–	1	**13**	4	–	5	7	8	1	3	2	–	1	**7**	**10**	5	3	1	6	3	2	2
INDU	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
BSS	–	–	–	–	100	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
VBS	–	5	1	3	**15**	**8**	3	2	5	4	4	–	–	–	3	1	1	9	–	–	–	–	–	4	5	–	–	**20**	3
ALL	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
BSS	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	100	–	–	–	–	–	–	–	–	–	–	–	–	–
VBS	–	2	–	1	4	2	2	1	1	3	2	11	4	**13**	3	–	1	3	–	–	3	**13**	4	2	1	3	**15**	5	1

Results are expressed as percentages and entries with value < 0.5 have been reported as "–". The top three solvers for each approach are identified in bold. Solvers are: 1: clasp-2.1.1_jumpy, 2: clasp-2.1.1_trendy, 3: ebminisat, 4: glueminisat, 5: lingeling, 6: lrglshr, 7: picosat, 8: restartsat, 9: circminisat, 10: clasp1, 11: cryptominisat_2011, 12: eagleup, 13: gnoveltyp2, 14: march_rw, 15: mphaseSAT, 16: mphaseSATm, 17: precosat, 18: qutersat, 19: sapperlot, 20: sat4j-2.3.2, 21: sattimep, 22: sparrow, 23: tnm, 24: cryptominisat295, 25: minisatPSM, 26: sattime2011, 27: ccasat, 28: glucose_21, 29: glucose_21_modified

10.3 Alternate Clustering Techniques

Clustering is a typical example of unsupervised learning. Provided with a collection of instances, the task is to group the data into subsets (clusters) in a way that instances in the same subset present some kind of similarity. So far we have been predominantly focusing on g-means as the goto clustering approach and the Euclidean distance as a metric. This chapter presents two common alternatives but it will ultimately show that g-means is by far the most robust approach.

10.3.1 X-Means

One of the most used clustering methods is Lloyd's k-means [72]. The idea behind the proposed methodology (presented in Algorithm 1) is to start from k random points in the feature space. These points represent the centers of the k clusters. The algorithm alternates between two phases. First, it assigns each instance to the nearest of the k points. Second, it updates the k centers to shift to the geometric centers of the instances in each cluster.

k-means is one of the most used clustering techniques for its simplicity and ease of implementation. Unfortunately, a drawback is its need to pre-specify the expected number of clusters, a value not often known in practice. Therefore, to address this issue, X-means has been introduced in [89]. The idea is to run k-means on the input data with an increasing k (up to a chosen upper bound), and choose the best value of k based on a Bayesian Information Criterion (BIC). Intuitively, the BIC statistic used by X-means has been formulated to maximize the likelihood for spherically distributed data. Thus, in general, it overestimates the number of true clusters in non-spherical data.

10.3.2 Hierarchical Clustering

Hierarchical Clustering is an iterative approach that constructs the clusters by recursively partitioning the instances in either a top-down or a bottom-up fashion [97]. This means that there are two versions:

- *Agglomerative*: Each instance, at the beginning, represents its own cluster. Then clusters are merged until the desired cluster structure is reached;
- *Divisive*: Each instance initially belongs to one cluster. Then the cluster is divided into sub-clusters, which are successively divided into their own sub-clusters. The process continues until the desired cluster structure is reached.

The decision of which cluster to combine or split is based on a measure of dissimilarity between sets of instances, which in our case this is the Euclidean distance metric.

Regardless of the flavoring, this approach produces a dendogram[1] representing the nested grouping of the instances. A clustering of the data is obtained by cutting the dendogram at a specified level. One main drawback, however, is the computational cost as the distance between every pair of instances in the dataset needs to be repeatedly computed.

10.4 Feature Filtering

In order to succeed, ISAC relies strongly on the quality of the available features. Too few features are likely to not adequately differentiate between instances but too many features are likely to cause over fitting or even erroneous clustering due to noisy features. This section therefore presents a number of possible filtering techniques that can be used to effectively reduce the number of required features: chi squared, information gain, gain ratio, and symmetrical uncertainty.

Here, we refer to feature filtering as a function that returns a relevance index $J(S|D)$ that estimates, given the data D, how relevant a given feature subset S is for the classification task Y. Specifically, each training instance is classified by the best performing solver on that instance. Therefore, the ranking of any feature should correlate to its ability to separate instances that prefer one solver from those that prefer another. The presented experiments will focus on selecting the set of features whose ranking is statistically higher than that of the remainder.

10.4.1 Chi-Squared

The Chi-Squared filtering approach aims to measure the dependence between each feature and its classification. More specifically, it can be defined as:

$$\chi^2(t,c) = \frac{N(AD - CB)^2}{(A + C)(B + D)(A + B)(C + D)}.$$

A is the number of times feature t and category c co-occur. Conversely, B is the number of times t occurs without c and C is the number of times c occurs without t. D is the number of times neither t nor c occurs, with N being the total number of

[1] A tree diagram where the leaves are the instances and the internal nodes are the clusters to which they belong.

instances. In practice, the feature values are discretized in order to work on numeric data.

10.4.2 Information Theory-Based Methods

Utilization of information theory indices is the most frequently used approach for feature evaluation. In particular, these methods are based on the concept of information, which is the negation of entropy:

$$H(Y) = -\sum_{i=1}^{K} P(y_i) \log_2 P(y_i)$$

where $P(y_i) = m_i/m$ is the fraction of samples x from class $y_i, i = 1 \dots K$. Information contained in the joint distribution of classes and features, summed over all classes, gives an estimation of the importance of the feature, where the information contained in the joint distribution is defined as:

$$H(Y, X) = -\sum_{i} \sum_{j=1}^{K} P(y_j, x_i) \log_2 P(y_j, x_i).$$

In case of continuous features, it is again necessary to utilize discretization. The remaining three feature filtering techniques can therefore be defined as:

- *information gain:* $H(Y) + H(X) - H(Y, X)$
- *gain ratio:* $\frac{H(Y)+H(X)-H(Y,X)}{H(X)}$
- *symmetrical uncertainty:* $2 \cdot \frac{H(Y)+H(X)-H(Y,X)}{H(Y)+H(X)}$

It is commonly believed that the *gain ratio* is a stable evaluation and the *symmetrical uncertainty* has a low bias for multivalued features.

10.5 Numerical Results

Utilizing the SAT data presented in Sect. 10.1 we first perform a preliminary analysis evaluating the performances of the various feature filtering approaches and the clustering methodologies. These results are summarized in Fig. 10.1, which clearly shows that in most scenarios g-means is the preferred clustering method. In particular, since this is the best method for the dataset that is comprised of all instances from the other three, g-means seems to be the best choice for clustering.

Fixing the choice of clustering methodology, Table 10.2 demonstrates the effectiveness of instance-specific algorithm selection. Here, we compare the techniques

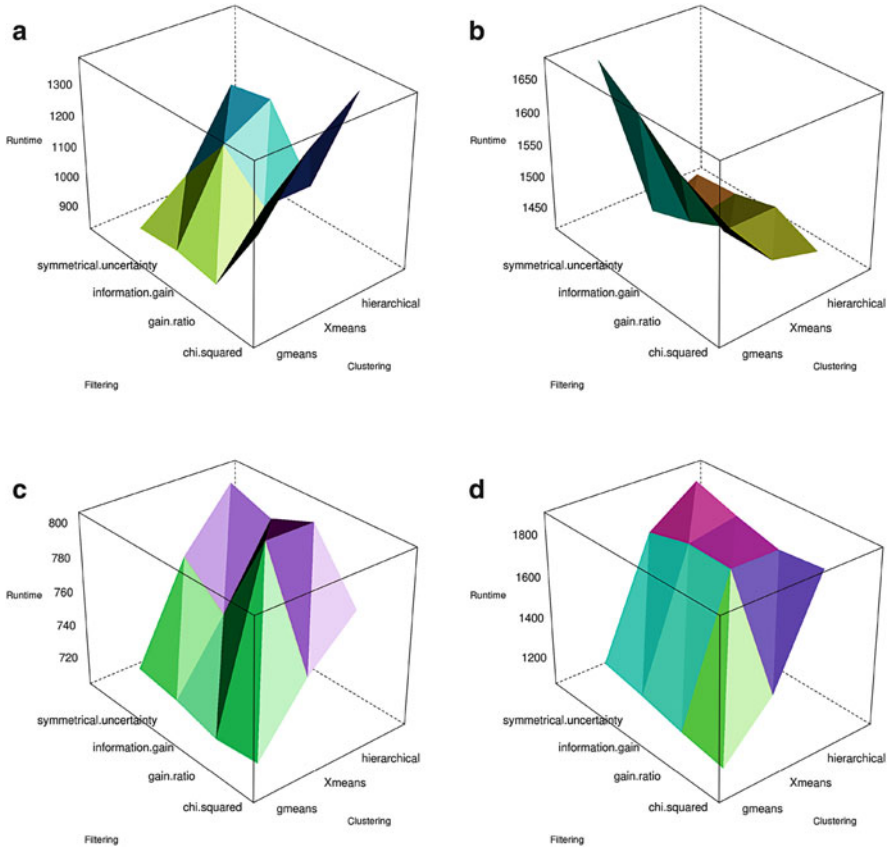

Fig. 10.1 Performances of ISAC on SAT considering different feature filtering and clustering techniques. (**a**) RAND dataset. (**b**) HAND dataset. (**c**) INDU dataset. (**d**) ALL dataset

using the usual three criteria: average runtime, PAR10, and the percentage of instances *unsolved*. To validate the performances, each approach was evaluated using tenfold cross-validation. The standard deviations in the table provide the changes over the 10 runs.

As a baseline, Table 10.2 presents the performance of the single best solver (BSS) over each dataset, as well as the oracle virtual best solver (VBS), which for each instance selects the best running solver. ISAC is the vanilla cluster-based algorithm selection approach employing no feature filtering. Conversely, "chi-squared," "information gain," "symmetrical uncertainty," and "gain ratio" specify ISAC using the appropriate feature selection methodology.

Table 10.2 Results on the SAT benchmark, comparing the virtual best solver (VBS), the best single solver (BSS), instance-specific algorithm selection (ISAC) and ISAC with different feature filtering techniques: "chi.squared," "information.gain," "symmetrical.uncertainty," and "gain.ratio"

RAND	Runtime - avg (std)	Par 10 - avg (std)	% not solved - avg (std)
BSS	1,551 (0)	13,154 (0)	25.28 (0)
ISAC	826.1(6.6)	4,584 (40.9)	8.1 (0.2)
chi.squared	1,081(42.23)	7,318 (492.7)	14 (1)
information.gain	851.5 (32.33)	5,161 (390)	8.7 (0.8)
symmetrical.uncertainty	840.2 (13.15)	4,908 (189.5)	8.76 (0.4)
gain.ratio	830.3 (21.3)	4,780 (210)	9 (0.4)
VBS	358 (0)	358 (0)	0 (0)
HAND	Runtime - avg (std)	Par 10 - avg (std)	% not solved - avg (std)
BSS	2,080 (0)	15,987 (0)	30.3 (0)
ISAC	1,743 (34.4)	13,994 (290.6)	26.5 (0.9)
chi.squared	1,544 (37.8)	11,771 (435)	23.5 (0.9)
information.gain	1,641 (38.9)	12,991 (443)	24.3 (0.9)
symmetrical.uncertainty	1,686 (27.3)	13,041 (336)	25.7 (0.7)
gain.ratio	1,588 (43.7)	12,092 (545)	22.4 (1)
VBS	400 (0)	400 (0)	0 (0)
INDU	Runtime - avg (std)	Par 10 - avg (std)	% not solved - avg (std)
BSS	871 (0)	4,727 (0)	8.4 (0)
ISAC	763.4 (4.7)	3,166 (155.6)	5.2 (0.7)
chi.squared	708.1 (25.3)	3,252 (218)	5.8 (0.4)
information.gain	712.6 (7.24)	2,578 (120)	4.3 (0.3)
symmetrical.uncertainty	716.4 (16.76)	2,737 (150)	4.4 (0.3)
gain.ratio	705.4 (19.9)	2,697 (284)	4.1 (0.6)
VBS	319 (0)	319 (0)	0 (0)
ALL	Runtime - avg (std)	Par 10 - avg (std)	% not solved
BSS	2,015 (0)	4,726 (0)	30.9 (0)
ISAC	1,015 (10.3)	6,447 (92.4)	11.8 (0.2)
chi.squared	1,078 (29.7)	7,051 (414)	11.79 (0.8)
information.gain	1,157 (18.9)	7,950 (208)	15 (0.4)
symmetrical.uncertainty	1,195 (28.7)	8,067 (341)	15.6 (0.7)
gain.ratio	1,111 (17.4)	6,678 (225)	13.39 (0.5)
VBS	353 (0)	353 (0)	0 (0)

From this data it is once more clear that using the presented methodology can provide significant improvements over using only a single solver to solve all instances. Surprisingly, however, it appears that while chi-squared filtering can often improve performance, the performance gains are not very pronounced. Yet for all these clustering approaches, we reduce the feature vector from 115 to only 15.

10.6 Extending the Feature Space

While the original version of ISAC employs Euclidean distance for clustering, there is no reason to believe that this is the best distance function. As an alternative one might learn a weighted distance metric, where the weights are tuned to match the desired similarity between two instances. For example, if two instances have the same best solver, then the distance between these two instances should be small. Alternatively, when a solver performs very well on one instance, but poorly on another, it might be desirable for these instances to be separated by a large distance.

Initial experiments trained a distance function that attempts to capture this desired behavior. Yet the resulting performance often was worse than that of the standard Euclidean distance. There are a number of reasons for this. First, while we know that some instances should be closer to or farther from each other, the ideal magnitude of the distance cannot be readily determined. Second, the effectiveness of the distance function depends on near-perfect accuracy since any mistake can distort the distance space. Third, the exact form of the distance function is not known. It is, for example, possible that even though two instances share the same best solver, they should nevertheless be allowed to be in opposite corners of the distance space. We do not necessarily want every instance preferring the same solver to be placed in the same cluster, but want to avoid contradictory preferences within the same cluster.

Due to these complications, here we present an alternate methodology for refining the feature vector. Specifically, we add the normalized performance of all solvers as a part of the features. In this setting, for each instance, the best performing solver is assigned a value of –1, while the worst performing is assigned 1. Everything in between is scaled accordingly. The clustering is then done on the set of both the normal features and the new ones. During testing, however, we do not know the performance of any of the solvers beforehand, so we set all those features to 0.

We see in Table 10.3 that the performance of this approach (called NormTimes ISAC) was really poor, never comparable with the running times of the normal ISAC. The main reason was that we were taking into consideration too many solvers during the computation of the new features.

As an alternative, one can consider that matching the performance of all solvers is too constraining. Implicitly ISAC assumes that a good cluster is one where the instances all prefer the same solver. For this reason one can choose to take into account only the performance of the best two solvers per instance. This is accomplished by extending the normal set of features with a vector of new features (one per solver), and assigning a value of 1 to the components corresponding to the best two solvers and 0 to all the others. In the testing set, since we again do not know which are the best two solvers beforehand, all the new features are set to the constant value of 0. As can be seen in Table 10.3, depending on which of the four datasets was used, we got different results (this approach is called bestTwoSolv ISAC): we observed a small improvement in the crafted and industrial datasets,

Table 10.3 Results on the SAT benchmark, comparing the best single solver (BSS), the virtual best solver (VBS), the original ISAC approach (ISAC), the ISAC approach with extra features coming from the running times: "NormTimes ISAC" has the normalized running times while "BestTwoSolv ISAC" takes into consideration just the best two solvers per each instance

RAND	Runtime - avg (std)	Par 10 - avg (std)	% not solved - avg (std)
BSS	1,551 (0)	13,154 (0)	25.28 (0)
ISAC	826.1 (6.6)	4,584 (40.9)	8.1 (0.2)
NormTimes ISAC	1,940 (–)	15,710 (–)	30 (–)
BestTwoSolv ISAC	825.6 (5.7)	4,561 (87.8)	8.1 (0.2)
VBS	358 (0)	358 (0)	0 (0)
HAND	Runtime - avg (std)	Par 10 - avg (std)	% not solved - avg (std)
BSS	2,080 (0)	15,987 (0)	30.3 (0)
ISAC	1,743 (34.4)	13,994 (290.6)	26.5 (0.9)
NormTimes ISAC	1,853 (–)	14,842 (–)	28.3 (–)
BestTwoSolv ISAC	1,725 (29.2)	13,884 (124.4)	26.5 (0.8)
VBS	400 (0)	400 (0)	0 (0)
INDU	Runtime - avg (std)	Par 10 - avg (std)	% not solved - avg (std)
BSS	871 (0)	4,727 (0)	8.4 (0)
ISAC	763.4 (4.7)	3,166 (155.6)	5.2 (0.7)
NormTimes ISAC	934.3 (–)	5,891 (–)	10.8 (–)
BestTwoSolv ISAC	750.5 (2.4)	2,917 (157.3)	4.7 (0.4)
VBS	319 (0)	319 (0)	0 (0)
ALL	Runtime - avg (std)	Par 10 - avg (std)	% not solved - avg (std)
BSS	2,015 (0)	14,727 (0)	30.9 (0)
ISAC	1,015 (10.3)	6,447 (92.4)	11.8 (0.2)
NormTimes ISAC	1,151 (–)	6,923 (–)	12.5 (–)
BestTwoSolv ISAC	1,019 (11.5)	6,484 (172.3)	11.9 (0.3)
VBS	353 (0)	353(0)	0 (0)

while for the other two datasets the results were almost the same as that of the pure ISAC methodology.

The drawbacks of directly extending the feature vector with the performance of solvers are twofold. First, the performance of the solvers is not available prior to solving a previously unseen test instance. Secondly, even if the new features are helpful in determining a better clustering, there is usually a large number of original features that might be resisting the desired clustering. Yet even though these extensions to the feature vector did not provide a compelling case for being used instead of the vanilla ISAC approach, they nonetheless support the assumption that by considering solver performances on the training data it is possible to improve the quality of the overall clustering. This therefore leads to a solver-based nearest neighbor approach which we describe in the next section.

10.7 SNNAP

There are two main takeaway messages from extending the feature vector with solver performances. First, the addition of solver performances can be helpful, but the inclusion of the original features can be disruptive for finding the desired cluster. Second, it is not necessary to find instances where the relation of every solver is similar to the current instance. It is enough to just know the best two or three solvers for an instance. Using these two ideas this section presents SNNAP, which appears as Algorithm 9.

During the training phase the algorithm is provided with a list of training instances T, their corresponding feature vectors F, and the running times R of every solver in our portfolio. We then train a single model PM for every solver to predict the expected runtime on a given instance. We have claimed previously that such models are difficult to train properly since any misclassification can result in the selection of the wrong solver. In fact, this was partly why the original version of ISAC outperformed these types of regression-based portfolios. Clusters provide better stability of the resulting prediction of which solver to choose. We are, however, not interested in using the trained model to predict the single best solver to be used on the instance. Instead, we just want to know which solvers are going to behave well on a particular instance.

For training the model, we scale the running times of the solvers on one instance so that the scaled vector will have a mean of 0 and unitary standard deviation. This kind of scaling is crucial in helping the following phase of prediction. Thus we are not training to predict runtime. We are learning to predict when a solver will perform much better than usual. When doing so, for every instance, every solver that behaves better than one standard deviation from the mean will receive a score less than –1, the solvers which behave worse than one standard deviation from the

Algorithm 9: Solver-based nearest neighbor for algorithm portfolios.

SNNAP-Train(T, F, R)
for all instances i in T **do**
    $\bar{R}_i \leftarrow Scaled(R_i)$
end for
for all solver j in the portfolio **do**
    $PM_j \leftarrow PredictionModel(T, F, \bar{R})$
end for
return PM

SNNAP-Run($x, PM, T, R, \bar{R}, A, k$)
$PR \leftarrow Predict(PM, x)$
$dist \leftarrow CalculateDistance(PR, T, \bar{R})$
$neighbors \leftarrow FindClosestInstances(dist, k)$
$j \leftarrow FindBestSolver(neighbors, R)$
return $A_j(x)$

mean will receive a score greater than 1, and the other solvers will lie in between. Here, random forests [26] were used as the prediction model.

In the prediction phase, the procedure is presented with a previously unseen instance x, the prediction models PM (one per solver), the training instances T, their running times R (and the scaled version $\bar{R}$), the portfolio of solvers A, and the size of the desired neighborhood k. The procedure first uses the prediction models to infer the performances PR of the solvers on the instance x, using its originally known features. SNNAP then continues to use these performances to compute a distance between the new instance and every training instance, selecting the k nearest ones among them. The distance calculation takes into account only the scaled running time of the instances of the training set and the predicted performances PR of the different solvers on the instance x. At the end the instance x will be solved using the solver that behaves best (measured as the average running time) on the k neighbors previously chosen.

It is worth highlighting again that we are not trying to predict the running times of the solvers on the instances but, after scaling, to predict a ranking amongst the solvers on a particular instance: which will be the best, the second best, and so on. Moreover, as shown in the next section, we are interested in learning a ranking among not all the solvers, but just a small subset of them, specifically for each instance which will be the best n solvers.

10.7.1 Choosing the Distance Metric

The k-nearest-neighbors approach is usually used in conjunction with the weighted Euclidean distance; unfortunately the Euclidean distance does not take into account the performances of the solvers in a way that is helpful to us. What is needed is a distance metric that takes into account the performances of the solvers and that allows the possibility of making some mistakes in the prediction phase without too much prejudice on the performances. Thus the metric should be trained with the goal that the k nearest neighbors always prefer to be solved by the same solver while instances that prefer different solvers are separated by a large margin.

Given two instances a and b and the running times of the m algorithms in the portfolio A on both of them, $R_{a_1}, \ldots R_{a_m}$ and $R_{b_1}, \ldots R_{b_m}$, we identify the best n solvers on each $(A_{a_1}, \ldots A_{a_n})$ and $(A_{b_1}, \ldots A_{b_n})$ and define their distance as a Jaccard distance:

$$1 - \frac{|intersection((A_{a_1}, \ldots A_{a_n}), (A_{b_1}, \ldots A_{b_n}))|}{|union((A_{a_1}, \ldots A_{a_n}), (A_{b_1}, \ldots A_{b_n}))|}.$$

Using this definition, two instances that will prefer the exact same n solvers will have a distance of 0, while instances which prefer completely different solvers will have a distance of 1. Moreover, using this kind of distance metric we are no longer concerned with making small mistakes in the prediction phase: even if we switch the

ranking between the best n solvers, the distance between two instances will remain the same. In the presented experiments, we focus on setting $n = 3$, as with higher values the performances degrades.

10.7.2 Numerical Results

In SNNAP, with the Jaccard distance metric, for each instance we are interested in knowing the n best solvers. In the prediction phase, we used random forests, which achieved high levels of accuracy: as stated in Table 10.4 we correctly made 91, 89, 91, and 91 % (respectively for RAND, HAND, INDU and ALL datasets) of the predictions. We compute these percentages in the following manner. There are 29 predictions made (one per solver) for each instance, giving us a total of 5,626, 1,044, and 2,320 predictions per category. We define accuracy as the percentage of matches between the predicted best n solvers and the true best n.

Having tried different parameters we use here the performance of just the $n = 3$ best solvers in the calculation of the distance metric and a neighborhood size of 60. Choosing a larger number of solvers degrades the results. This is most likely due to scenarios where one instance is solved well by a limited number of solvers, while all the others time out.

As can be seen in Table 10.5, the best improvement, as compared to the standard ISAC, is achieved in the crafted dataset. Not only are the performances improved by 60 %, but also the number of unsolved instances is halved; this also has a great impact on the PAR10 evaluation. It is interesting to note that the crafted dataset is the one that proves to be most difficult in terms of solving time, while being the setting in which we achieved the most improvement.

We also achieve a significant improvement, although lower than that with the crafted dataset, on the industrial and ALL ($\sim$ 25 %) datasets. Here the number of unsolved instances was also halved. In the random dataset we achieved the lowest improvement but we were able to overtake significantly the standard ISAC approach.

We have also applied feature filtering to SNNAP and the results are shown in Table 10.5. Feature filtering is again proved beneficial, significantly improving the results for all our datasets and giving us a clue that not all 115 features are essential. Results in the table have been reported only for the more successful ranking function

Table 10.4 Statistics of the four datasets used: Instances generated at random "RAND," crafted instances "HAND," industrial instances "INDU," and the union of them "ALL"

	RAND	HAND	INDU	ALL
Number of instances considered	1,949	363	805	3,117
Number of predictions	5,626	1,044	2,320	9,019
Accuracy in the prediction phase (%)	91	89	91	91

Table 10.5 Results on the SAT benchmark, comparing the best single solver (BSS), the original ISAC approach (ISAC), our SNNAP approach (SNNAP) (also with feature filtering), and the virtual best solver (VBS)

RAND	Runtime - avg (std)	Par 10 - avg (std)	% not solved - avg (std)
BSS	1,551 (0)	13,154 (0)	25.28 (0)
ISAC	826.1 (6.6)	4,584 (40.9)	8.1 (0.2)
SNNAP	791.4 (15.7)	4,119 (207)	7.3 (0.2)
SNNAP + Filtering	723 (9.27)	3,138 (76.9)	5.28 (0.1)
VBS	358 (0)	358 (0)	0 (0)
HAND	Runtime - avg (std)	Par 10 - avg (std)	% not solved - avg (std)
BSS	2,080 (0)	15,987 (0)	30.3 (0)
ISAC	1,743 (34.4)	13,994 (290.6)	26.5 (0.9)
SNNAP	1,063 (33.86)	6,741 (405.5)	12.4 (0.4)
SNNAP + Filtering	995.5 (18.23)	6,036 (449)	10.5 (0.4)
VBS	400 (0)	400 (0)	0 (0)
INDU	Runtime - avg (std)	Par 10 - avg (std)	% not solved - avg (std)
BSS	871 (0)	4,727 (0)	8.4 (0)
ISAC	763.4 (4.7)	3,166 (155.6)	5.2 (0.7)
SNNAP	577.6 (21.5)	1,776 (220.8)	2.6 (0.4)
SNNAP + Filtering	540 (15.52)	1,630 (149)	2.4 (0.4)
VBS	319 (0)	319 (0)	0 (0)
ALL	Runtime - avg (std)	Par 10 - avg (std)	% not solved - avg (std)
BSS	2,015 (0)	14,727 (0)	30.9 (0)
ISAC	1,015 (10.3)	6,447 (92.4)	11.8 (0.2)
SNNAP	744.2 (14)	3,428 (141.2)	5.8 (0.2)
SNNAP + Filtering	692.9 (7.2)	2,741 (211.9)	4.5 (0.1)
VBS	353 (0)	353 (0)	0 (0)

(gain ratio for the random dataset, chi-squared for the crafted, industrial, and ALL datasets).

It is clear that the dramatic decrease in the number of unsolved instances is highly important, as they are key to lowering the average and the PAR10 scores. This result can also be observed in Table 10.6, where we can see the percentage of instances solved/not solved by each approach. In particular, the most significant result is achieved, again, in the HAND dataset, where the number of instances not solved by ISAC but solved by our approach is 17.4 % of the overall instances, while the number of instances not solved by our approach but solved by ISAC is only 3.3 %. As we can see, this difference is also considerable in the other three datasets. Deliberately, we chose to show this matrix only for the version of ISAC and SNNAP without feature filtering as it offers an unbiased comparison between the two approaches, as we have shown that ISAC does not improve after feature filtering.

Table 10.6 Matrix for comparing instances solved and not solved using SNNAP and ISAC for the four datasets: RAND, HAND, INDU, and ALL

RAND		
SNNAP \ISAC	Solved	Not solved
Solved	89.4	3.3
Not solved	2.5	4.8

HAND		
SNNAP \ISAC	Solved	Not solved
Solved	70.2	17.4
Not solved	3.3	9.1

INDU		
SNNAP \ISAC	Solved	Not solved
Solved	93.9	3.5
Not solved	0.9	1.7

ALL		
SNNAP \ISAC	Solved	Not solved
Solved	85.8	8.4
Not solved	2.4	3.4

Values are in percentages

10.8 Chapter Summary

Instance-Specific Algorithm Configuration (ISAC) is a successful approach to tuning a wide range of solvers for SAT, MIP, set covering, and others. Yet, as has been described in previous chapters, this approach assumes that the features describing an instance are enough to group instances so that all instances in the cluster prefer the same solver. Yet there is no fundamental reason why this hypothesis should hold. This chapter shows that the assumptions that ISAC makes can be strengthened. Shown is the fact that not all employed features are useful and that it is possible to achieve similar performance with only a fraction of the features that are available. Further shown is the fact that it is possible to extend the feature vector to include the past performances of solvers to help guide the clustering process. Finally, an alternative view of ISAC is presented which uses the existing features to predict the best three solvers for a particular instance. Using the k nearest neighbors, the approach then scans the training data to find other instances that preferred the same solvers, and uses them as a dynamically formed training set to select the best solver to use. This methodology is referred to as Solver-based Nearest Neighbors for Algorithm Portfolios (SNNAP).

Chapter 11
Conclusion

This book introduces the new methodology of instance-specific algorithm configuration or ISAC. Although there has recently been a surge of research in the area of automatic algorithm configuration, ISAC enhances the existing work by merging the strengths of two powerful techniques: instance-oblivious tuning and instance-specific regression. When used in isolation, these two methodologies have major drawbacks. Existing instance-oblivious parameter tuners assume that there is a single parameter set that will provide optimal performance over all instances, an assumption that is not provably true for NP-hard problems. Instance-specific regression, on the other hand, depends on accurately fitting a model to map from features to a parameter, which is a challenging task requiring a lot of training data when the features and parameters have non-linear interactions. ISAC resolves these issues by relying on machine learning techniques.

This approach has been shown to be beneficial on a variety of problem types and solvers. This book has presented a number of possible configurations and consistently expanded the possible applications of ISAC. The main idea behind this methodology is the assumption that although solvers have varied performance on different instances, instances that are similar in structure result in similar performance for a particular solver. The objective then becomes to identify these clusters of similar instances and then tune a solver for each cluster. To find such clusters, ISAC identifies each instance as a vector of descriptive features. When a new instance needs to be solved, its computed features are used to assign it to a cluster, which in turn determines the parameterization of the solver used to solve it. Based on this methodology, the book began by using a parameterized solver, a normalized vector of all the supplied instance features, g-means [44] for clustering, and GGA [5] for training. This was shown to be highly effective for set covering problems, mixed integer problems, satisfiability problems, and machine reassignment.

The approach was then extended to be applicable to portfolios of solvers by changing the training methodology. There are oftentimes many solvers that can be

Y. Malitsky, *Instance-Specific Algorithm Configuration*,
DOI 10.1007/978-3-319-11230-5_11

tuned, so instead of choosing and relying on only tuning a single parameterized solver, we saw how to create a meta solver whose parameters could determine not only which solver should be employed but also the best parameters for that solver. When applied to satisfiability problems, this portfolio-based approach was empirically shown to outperform the existing state-of-the-art regression-based algorithm portfolio solvers like SATzilla [126] and Hydra [124].

The book then showed how ISAC can be trained dynamically for each new test instance by changing the clustering methodology to k nearest neighbor and further improved by training sequential schedules of solvers in a portfolio. Although similar to the previously existing constraint satisfaction solver CPHydra [87], this book showed how to use particular integer programming and column generation to create a more efficient scheduling algorithm which was able to efficiently handle over 60 solvers and hundreds of instances in an online setting. This last implementation was the basis of a 3S SAT solver that won seven medals in the 2011 SAT Competition.

ISAC was then further expanded in three orthogonal ways. First, the book showed how to expand the methodology behind 3S to create a parallel schedule of solvers dynamically. The book compared the resulting portfolio with the current state-of-the-art parallel solvers on instances from all SAT categories and showed that creating these parallel schedules marks a very significant improvement in the ability to solve SAT instances. The new portfolio generator was then used to generate a parallel portfolio for application instances based on the latest parallel and sequential SAT solvers available. This portfolio performed well in the 2012 SAT Challenge.

Next, this book showed how the ISAC methodology can be used to create an adaptive tree search-based solver that dynamically chooses the branching heuristic that is most appropriate for the current sub-problem it observes. Here, the book showed that in many cases in optimization when performing a complete search, each subtree is still a problem of the same type as the original but with a slightly different structure. By identifying how this structure changes during search, the solver can dynamically change its guiding heuristics to best accommodate the new sub-tree. Tested on the general MIP problem, this adaptive technique was shown to be highly effective.

The book also touched on scenarios where a train-once methodology was not appropriate. Scenarios were discussed where new instances come over time and gradually become different than those used to initially train the portfolio. The corresponding chapter discussed how to identify moments when the current portfolio was no longer applicable and needed to be updated. The chapter then further showed how this retuning could be done efficiently.

Finally, the book discussed the assumptions that have been made by ISAC to that point and demonstrated their validity. Further discussed were ways how these assumptions could be strengthened and refined to make an increasingly accurate portfolio.

In its entirety, the book showed that ISAC is a highly configurable and effective methodology, demonstrating that it is possible to train a solver or algorithm automatically for enhanced performance while requiring minimal expertise and involvement on the part of the user.

Having laid out the groundwork, there are a number of future directions that can be pursued to push the state-of-the-art further. One such direction involves a deeper analysis of the base assumption that instances with similar features tend to yield to the same algorithm. One way to do this is by creating instances that have a specific feature vector. This way, arbitrarily tight clusters can be generated and trained on automatically. An additional benefit to this line of research will be the ability to expand the small existing benchmarks. As was noted in this book, certain problem types have limited numbers of instances that cannot be used for training effectively. This is the case for the set partitioning problems, the standard MIP benchmark, and the industrial SAT instances.

Furthermore, by being able to generate instances with a specific feature vector automatically, the entirety of the problem space can be explored. Such an overview of the search space can give insight into how smooth the transitions are between two clusters, how wide the clusters are, and how numerous the hard or easy clusters are. Additionally, solvers won't need to be created for the general case, but instead research can be focused on each separate cluster where state-of-the-art solvers will be struggling.

Exploring the entire problem space has an added bonus of identifying clusters of easier and harder instances. Such knowledge can then be exploited by studying instance perturbation techniques. In such a scenario, when an instance is identified as belonging to a hard cluster, it might be possible to apply some efficient modifications to the instance, changing its features and thereby shifting it into an easier cluster. Observing these areas of easier and harder instances in the problem space would also allow for a better understanding of the type of structure that leads to difficult problems.

Alternatively, it would be interesting to explore the relations between problem types. It is well known that any NP-complete problem can be reduced to another NP-complete problem in polynomial time. This fact could be used to avoid coming up with new features for each new problem type. Instead it might be enough to convert the instance to SAT and use the 48 well-established features. Some preliminary testing with cryptography instances suggests that regardless of the conversion technique used to get the SAT instance, the resulting clusters are usually very similar. Additionally for CP problems, a domain where a vector of features has already been proposed, converting the problems to SAT results in similar clusters regardless of whether the CP or the SAT features are employed.

Transitions to SAT are always for NP-complete problems but not always easy or straightforward, so additional research needs to be done in automating the feature generation. This can be done by converting the problem into black box optimization (BBO). These problems arise in numerous applications, especially in scientific and engineering contexts in problems that are too incomplete for developing effective problem-specific heuristics. So the feature computation can be done through sampling of the instance's solution space to get a sense of the search terrain. The question that will need to be answered is how much critical information is needed when converting a problem like SAT into a BBO in order to compute its features.

Further research should also be done for the instance-oblivious tuning. In the experiments presented in this book, GGA performed well, but in all of the tuned experiments had a short cutoff time: under 20 min. Tuning problems where each instance can take more time or that have more instances becomes computationally infeasible. For example, simulations are frequently relied on for disaster recovery, sustainability research, etc., and tuning these simulation algorithms to work quickly and efficiently would be of immense immediate benefit. These simulations, however, tend to run for extended periods of time in order for us to get accurate results. It is therefore important to investigate new techniques that can find useable parameterizations within a reasonable timeframe.

In summary, this book lays out the groundwork for a highly configurable and effective instance-specific algorithm configuration methodology, and hopefully further research will enhance and expand its applicability.

References

1. SAT Competition. http://www.satcomptition.org, 2013
2. H. Abdi, L.J. Williams, Principal component analysis. Wiley Interdiscip. Rev. Comput. Stat. **2**, 433–459 (2010)
3. T. Achterberg, T. Koch, A. Martin, Branching rules revisited. Oper. Res. Lett. **33**(1), 42–54 (2004)
4. B. Adenso-Diaz, M. Laguna, Fine-tuning of algorithms using fractional experimental designs and local search. Oper. Res. **54**(1), 99–114 (2006)
5. C. Ansótegui, M. Sellmann, K. Tierney, A gender-based genetic algorithm for the automatic configuration of algorithms, in *Proceedings of Principles and Practice of Constraint Programming (CP)*, vol. 5732, ed. by I. Gent (Springer, Berlin, 2009), pp. 142–157
6. A. Atamtürk, G.L. Nemhauser, M.W.P. Savelsbergh, Valid inequalities for problems with additive variable upper bounds. Math. Program. **91**(1), 145–162 (2001)
7. A. Atamtürk, Flow pack facets of the single node fixed-charge flow polytope. Oper. Res. Lett. **29**(3), 107–114 (2001)
8. A. Atamtürk, On the facets of the mixed-integer knapsack polyhedron. Math. Program. **98**(1–3), 145–175 (2003)
9. A. Atamtürk, J.C. Munoz, A study of the lot-sizing polytope. Math. Program. **99**(3), 443–465 (2004)
10. G. Audemard, L. Simon, GLUCOSE: a solver that predicts learnt clauses quality. SAT Competition (2009)
11. C. Audet, D. Orban, Finding optimal algorithmic parameters using derivative-free optimization. SIAM J. Optim. **17**(3), 642–664 (2006)
12. A. Balint, M. Henn, O. Gableske, hybridGM. Solver description. SAT Competition (2009)
13. R. Battiti, G. Tecchiolli, I. Nazionale, F. Nucleare, The reactive tabu search. INFORMS J. Comput. **6**(2), 126–140 (1993)
14. P. Berkhin, Survey of clustering data mining techniques, in *Grouping Multidimensional Data*, ed. by J. Kogan, C. Nicholas, M. Teboulle (Springer, Berlin, Heidelberg, 2002), pp. 25–71
15. A. Biere, Picosat version 846. Solver description. SAT Competition (2007)
16. A. Biere, P{re,i}coSATSC'09. SAT Competition (2009)
17. A. Biere, Lingeling. SAT Race (2010)
18. A. Biere, PLingeling. SAT Race (2010)
19. A. Biere, Lingeling and Friends at the SAT Competition 2011. Technical report, 4040 Linz (2011)

Y. Malitsky, *Instance-Specific Algorithm Configuration*,
DOI 10.1007/978-3-319-11230-5

20. M. Birattari, T. Stützle, L. Paquete, K. Varrentrapp, A racing algorithm for configuring metaheuristics, in *Proceedings of the Genetic and Evolutionary Computation Conference* (Morgan Kaufmann publishers, San Francisco, 2002), pp. 11–18
21. J. Boyan, A.W. Moore, P. Kaelbling, Learning evaluation functions to improve optimization by local search. J. Mach. Learn. Res. **1**, 77–112 (2000)
22. A. Braunstein, M. Mézard, R. Zecchina, Survey propagation: an algorithm for Satisfiability. Random Struct. Algorithm. **27**(2), 201–226 (2005)
23. D.R. Bregman, The SAT Solver MXC, Version 0.99. SAT Competition (2009)
24. D.R. Bregman, D.G. Mitchell, The SAT Solver MXC, version 0.75. Solver description. SAT Race (2008)
25. L. Breiman, Bagging predictors. Mach. Learn. **24**(2), 123–140 (1996)
26. L. Breiman, Random forests. Mach. Learn. **45**(1), 5–32 (2001)
27. S. Cai, K. Su, Configuration checking with aspiration in local search for SAT, in *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)* (AAAI Press, Toronto, 2012), pp. 434–440
28. A. Caprara, M. Fischetti, P. Toth, D. Vigo, P.L. Guida, Algorithms for railway crew management. Math. Program. **79**, 125–141 (1997)
29. S.P. Coy, B.L. Golden, G.C. Runger, E.A. Wasil, Using experimental design to find effective parameter settings for heuristics. J. Heuristics **7**(1), 77–97 (2001)
30. G.B. Dantzig, P. Wolfe, The decomposition algorithm for linear programs. Econometrica **29**(4), 767–778 (1961)
31. M. Davis, G. Logemann, D. Loveland, A machine program for theorem-proving. Commun. ACM **5**(7), 394–397 (1962)
32. G. Dequen, O. Dubois, kcnfs. Solver description. SAT Competition (2007)
33. N. Een, N. Sörensson, MiniSAT (2010). http://minisat.se
34. S.L. Epstein, E.C. Freuder, R.J. Wallace, A. Morozov, B. Samuels, The adaptive constraint engine, in *Principles and Practice of Constraint Programming (CP)*, vol. 2470, ed. by P.V. Hentenryck (Springer, Berlin, Heidelberg, 2002), pp. 525–542
35. A.S. Fukunaga, Automated discovery of local search heuristics for satisfiability testing. Evol. Comput. **16**(1), 31–61 (2008)
36. M. Gagliolo, J. Schmidhuber, Dynamic algorithm portfolios. Ann. Math. Artif. Intell. **47**, 3–4 (2006)
37. M. Gebser, B. Kaufmann, T. Schaub, Solution enumeration for projected boolean search problems, in *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR)*, vol. 5547, ed. by W.-J. van Hoeve, J.N. Hooker (Springer, Berlin Heidelberg, 2009), pp. 71–86
38. I.P. Gent, H.H. Hoos, P. Prosser, T. Walsh, Morphing: combining structure and randomness, in *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, vol. 9 (AAAI Press, Orlando, 1999), pp. 849–859
39. C.P. Gomes, B. Selman, Problem structure in the presence of perturbations, in *Proceedings of the National Conference on Artificial Intelligence (AAAI)* (AAAI Press, New Providence, 1997), pp. 221–226
40. C.P. Gomes, B. Selman, Algorithm portfolios. Artif. Intell. **126**(1–2), 43–62 (2001)
41. Google, Google ROADEF Challenge (2012). http://challenge.roadef.org/2012/en/index.php
42. M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, I.H. Witten, The WEKA data mining software: an update. SIGKDD Explor. Newsl. **11**(1), 10–18 (2009)
43. Y. Hamadi, S. Jabbour, L. Sais, LySAT: solver description. SAT Competition (2009)
44. G. Hamerly, C. Elkan, Learning the K in K-means, in *Neural Information Processing Systems* (MIT Press, Cambridge, 2003)
45. M. Heule, H. van Marren, march hi: Solver description. SAT Competition (2009)
46. M. Heule, H. van Marren, march nn (2009). http://www.st.ewi.tudelft.nl/sat/download.php

47. M. Heule, M. Dufour, J. Van Zwieten, H. Van Maaren, March eq: implementing additional reasoning into an efficient lookahead SAT solver, in *Proceedings of the International Conference on Theory and Application of Satisfiability Testing (SAT)*, vol. 3542 (Springer, Berlin, 2005), pp. 345–359
48. K.L. Hoffman, M. Padberg, Solving airline crew scheduling problems by branch-and-cut. Manag. Sci. **39**(6), 657–682 (1993)
49. H.H. Hoos, An adaptive noise mechanism for WalkSAT, in *Proceedings of the National Conference on Artificial Intelligence (AAAI)* (AAAI Press, Menlo Park, 2002), pp. 655–660
50. E. Housos, T. Elmroth, Automatic optimization of subproblems in scheduling airline crews. Interfaces **27**(5), 68–77 (1997)
51. B.A. Huberman, R.M. Lukose, T. Hogg, An economic approach to hard computational problems. Science **275**(5296), 51–53 (1997)
52. L. Hubert, P. Arabie, Comparing partitions. J. Classif. **2**(1), 193–218 (1985)
53. F. Hutter, Y. Hamadi, Parameter adjustment based on performance prediction: towards an instance-aware problem solver. Technical Report MSR-TR-2005-125, Microsoft Research (2005)
54. F. Hutter, Y. Hamadi, H.H. Hoos, K. Leyton-Brown, Performance prediction and automated tuning of randomized and parametric algorithms, in *Proceedings of Principles and Practice of Constraint Programming (CP)*, vol. 4204, ed. by F. Benhamou (Springer, Berlin, Heidelberg, 2006), pp. 213–228
55. F. Hutter, H.H. Hoos, K. Leyton-Brown, K. Murphy, Time-bounded sequential parameter pptimization, in *Learning and Intelligent Optimization (LION)*, vol. 6073, ed. by C. Blum, R. Battiti (Springer, Berlin, Heidelberg, 2010), pp. 281–298
56. F. Hutter, H.H. Hoos, K. Leyton-Brown, T. Stützle, ParamILS: an automatic algorithm configuration framework. J. Artif. Intell. Res. **36**(1), 267–306 (2009)
57. F. Hutter, D.A.D. Tompkins, H.H. Hoos, Scaling and probabilistic smoothing: efficient dynamic local search for SAT, in *Principles and Practice of Constraint Programming (CP)*, vol. 2470, ed. by P.V. Hentenryck (Springer, Berlin Heidelberg, 2002), pp. 233–248
58. F. Hutter, D.A.D. Tompkins, H.H. Hoos, Scaling and probabilistic smoothing: efficient dynamic local search for SAT, in *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)* (Springer, Berlin, 2002), pp. 233–248
59. IBM, Reference manual and user manual. V12.5 (2013)
60. S. Kadioglu, Y. Malitsky, M. Sellmann, K. Tierney, ISAC – instance-specific algorithm configuration, in *Proceedings of the European Conference on Artificial Intelligence (ECAI)* (IOS Press, Amsterdam, 2010), pp. 751–756
61. S. Kadioglu, M. Sellmann, Dialectic search, in *Proceedings of Principles and Practice of Constraint Programming (CP)*, vol. 5732, ed. by I. Gent (Springer, Berlin Heidelberg, 2009), pp. 486–500
62. A.R. Khudabukhsh, L. Xu, H.H. Hoos, K. Leyton-Brown, SATenstein: automatically building local search SAT solvers from components, in *International Joint Conference on Artificial Intelligence (IJCAI)* (AAAI Press, Menlo Park, 2009), pp. 517–524
63. T. Koch, T. Achterberg, E. Andersen, O. Bastert, T. Berthold, R.E. Bixby, E. Danna, G. Gamrath, A.M. Gleixner, S. Heinz, A. Lodi, H. Mittelmann, T. Ralphs, D. Salvagnin, D.E. Steffy, K. Wolter, MIPLIB 2010 - Mixed Integer Programming Library version 5. Math. Program. Comput. **3**(2), 103–163 (2011)
64. M.G. Lagoudakis, M.L. Littman, Learning to select branching rules in the DPLL procedure for satisfiability, in *LICS 2001 Workshop on Theory and Applications of Satisfiability Testing (SAT)*, vol. 9 (2001), pp. 344–359
65. D.H. Leventhal, M. Sellmann, The accuracy of search heuristics: an empirical study on knapsack problems, in *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR)*, vol. 5015, ed. by L. Perron, M.A. Trick (Springer, Berlin, Heidelberg, 2008), pp. 142–157

66. K. Leyton-Brown, E. Nudelman, G. Andrew, J. Mcfadden, Y. Shoham, A portfolio approach to algorithm selection, in *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)* (Morgan Kaufmann Publishers Inc., San Francisco, 2003), pp. 1542–1543
67. K. Leyton-Brown, E. Nudelman, G. Andrew, J. Mcfadden, Y. Shoham, Boosting as a Metaphor for Algorithm Design, in *Proceedings of Principles and Practice of Constraint Programming (CP)*, vol. 2833, ed. by F. Rossi (Springer, Berlin, Heidelberg, 2003), pp. 899–903
68. K. Leyton-Brown, M. Pearson, Y. Shoham, Towards a universal test suite for combinatorial auction algorithms, in *Proceedings of the ACM Conference on Electronic Commerce* (ACM Press, New York, 2000), pp. 66–76
69. C.M. Li, W.Q. Huang, G2WSAT: gradient-based greedy WalkSAT, in *Proceedings of the International Conference on Theory and Application of Satisfiability Testing (SAT)*, vol. 3569 (Springer, Berlin, 2005), pp. 158–172
70. C.M. Li and W. We. Combining Adaptive Noise and Promising Decreasing Variables in Local Search for SAT. Solver description. SAT Competition (2009)
71. X. Li, M.J. Garzarán, D. Padua, Optimizing sorting with genetic algorithms. *International Symposium on Code Generation and Optimization (CGO)* (2005), pp. 99–110
72. S.P. Lloyd, Least squares quantization in PCM. Trans. Inf. Theory **28**(2), 129–137 (1982)
73. L. Lobjois, M. Lemaître, Branch and bound algorithm selection by performance prediction, in *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)* (AAAI Press., Madison, 1998), pp. 353–358
74. Y. Malitsky, D. Mehta, B. O'Sullivan, H. Simonis, Tuning parameters of large neighborhood search for the machine reassignment problem, in *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR)*, vol. 7874, ed. by C. Gomes, M. Sellmann (Springer, Berlin, Heidelberg, 2013), pp. 176–192
75. Y. Malitsky, A. Sabharwal, H. Samulowitz, M. Sellmann, Algorithm portfolios based on cost-sensitive hierarchical clustering, in *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)* (AAAI Press, Beijing, 2013), pp. 608–614
76. Y. Malitsky, M. Sellmann, Stochastic offline programming, in *International Conference on Tools with Artificial Intelligence (ICTAI)* (IEEE Press, New Yark, 2009), pp. 784–791
77. Y. Malitsky, M. Sellmann, Instance-specific algorithm configuration as a method for non-model-based portfolio generation, in *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR)*, vol. 7298, ed. by N. Beldiceanu, N. Jussien, E. Pinson (Springer, Berlin, Heidelberg, 2012), pp. 244–259
78. D. McAllester, B. Selman, H. Kautz, Evidence for invariants in local search, in *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)* (AAAI Press, Providence, 1997), pp. 321–326
79. D. Mehta, B. O'Sullivan, H. Simonis, Comparing solution methods for the machine reassignment problem, in *Principles and Practice of Constraint Programming (CP)*, vol. 7514, ed. by M. Milano (Springer, Berlin, Heidelberg, 2012), pp. 782–797
80. S. Minton, Automatically configuring constraint satisfaction programs: a case study. Constraints **1**(1–2), 7–43 (1996)
81. M. Motoki, R. Uehara, Unique solution instance generation for the 3-Satisfiability (3SAT) problem. Technical Report COMP98-54, IEICE (1998)
82. M. Muja, D.G. Lowe, Fast approximate nearest neighbors with automatic algorithm configuration, in *VISAPP International Conference on Computer Vision Theory and Applications* (2009), pp. 331–340
83. N. Musliu, Local search algorithm for unicost set covering problem, in *Proceedings of the International Conference on Advances in Applied Artificial Intelligence: Industrial, Engineering and Other Applications of Applied Intelligent Systems (IEA/AIE)* (Springer, Berlin, 2006), pp. 302–311

84. M. Nikolic, F. Maric, P. Janici, Instance based selection of policies for SAT solvers, in *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing (SAT)* (Springer, Berlin, 2009), pp. 326–340
85. E. Nudelman, A. Devkar, Y. Shoham, K. Leyton-Brown, Understanding random SAT: beyond the clauses-to-variables ratio, in *Principles and Practice of Constraint Programming (CP)*, vol. 3258, ed. by M. Wallace (Springer, Berlin, Heidelberg, 2004), pp. 438–452
86. M. Oltean, Evolving evolutionary algorithms using linear genetic programming. Evol. Comput. **13**(3), 387–410 (2005)
87. E. O'Mahony, E. Hebrard, A. Holland, C. Nugent, B. O'ullivan, Using case-based reasoning in an algorithm portfolio for constraint solvingm, in *Irish Conference on Artificial Intelligence and Cognitive Science* (2008)
88. D.J. Patterson, H. Kautz, Auto-walksat: a self-tuning implementation of walksat, in *LICS Workshop on Theory and Applications of Satisfiability Testing (SAT)*, vol. 9 (2001), pp. 360–368
89. D. Pelleg, A.W. Moore, X-means: extending K-means with efficient estimation of the number of clusters, in *Proceedings of the Seventeenth International Conference on Machine Learning (ICML)* (Morgan Kaufmann Publishers Inc., San Francisco, 2000), pp. 727–734
90. M.P. Petrik, S. Zilberstein, Learning static parallel portfolios of algorithms, in *International Symposium on Artificial Intelligence and Mathematics (ISAIM)* (2006)
91. D.N. Pham, A. Anbulagan, Resolution enhanced SLS solver: R+AdaptNovelty+. solver description. SAT Competition (2007)
92. D.N. Pham, C. Gretton, gnovelty+. Solver description. SAT Competition (2007)
93. D.N. Pham, C. Gretton, gnovelty+ (v.2). Solver description. SAT Competition (2009)
94. S. Prestwich, Random walk with continuously smoothed variable weights, in *Theory and Applications of Satisfiability Testing (SAT)*, vol. 3569, ed. by F. Bacchus, T. Walsh (Springer, Berlin, Heidelberg, 2005), pp. 203–215
95. W.M. Rand, Objective criteria for the evaluation of clustering methods. J. Am. Stat. Assoc. **66**(336), 846–850 (1971)
96. P. Refalo, Impact-based search strategies for constraint programming, in *Principles and Practice of Constraint Programming (CP)*, vol. 3258, M. Wallace (Springer, Berlin, Heidelberg, 2004), pp. 557–571
97. L. Rokach, O. Maimon, Clustering methods. *The Data Mining and Knowledge Discovery Handbook* (Springer, New York, 2005), pp. 321–352
98. O. Roussel, Description of ppfolio (2011). http://www.cril.univ-artois.fr/~roussel/ppfolio/solver1.pdf
99. H. Samulowitz, R. Memisevic, Learning to solve QBF, in *Proceedings of the National Conference on Artificial Intelligence (AAAI)* (AAAI Press, Menlo Park, 2007), pp. 255–260
100. A. Saxena, MIP Benchmark Instances (2003). http://www.andrew.cmu.edu/user/anureets/mpsInstances.htm
101. M. Sellmann, Disco-novo-gogo: integrating local search and complete search with restarts, in *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)* (AAAI Press, Boston, 2006), pp. 1051–1056
102. B. Selman, H. Kautz, Domain-independent extensions to GSAT: solving large structured satisfiability problems, in *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)* (Morgan Kaufmann Publishers Inc., San Francisco, 1993.), pp. 290–295
103. B. Silverthorn, R. Miikkulainen, Latent class models for algorithm portfolio methods, in *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)* (AAAI Press, Atlanta, 2010)
104. A. Slater, Modeling more realistic SAT problems, in *Advances in Artificial Intelligence*, vol. 2557, ed. by B. McKay, J. Slaney (Springer, Berlin, Heidelberg, 2002), pp. 291–602
105. K. Smith-Miles, Cross-disciplinary perspectives on meta-learning for algorithm selection. ACM Comput. Surv. **41**(1), 6:1–6:25 (2009)
106. M. Soos, CryptoMiniSat 2.5.0. Solver description. SAT Race (2010)
107. M. Soos, Cryptominisat 2.9.0 (2011)

108. N. Sorensson, N. Een, MiniSAT 2.2.0 (2010). http://minisat.se
109. D. Stern, H. Samulowitz, R. Herbrich, T. Graepel, L. Pulina, A. Tacchella, Collaborative expert portfolio management, in *Proceedings of the National Conference on Artificial Intelligence (AAAI)* (AAAI Press, Atlanta, 2010)
110. M. Streeter, D. Golovin, S.F. Smith, Combining multiple heuristics online, in *Proceedings of the National Conference on Artificial Intelligence (AAAI)* (AAAI Press, Vancouver, 2007), pp. 1197–1203
111. M.J. Streeter, S.F. Smith, New techniques for algorithm portfolio design, in *Proceedings of the Conference in Uncertainty in Artificial Intelligence (UAI)*, ed. by D.A. McAllester, P. Myllymäki (AUAI Press, Helsinki, 2008), pp. 519–527
112. H. Terashima-Marín, P. Ross, Evolution of constraint satisfaction strategies in examination timetabling, in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, vol. 1, ed. by W. Banzhaf, J. Daida, A.E. Eiben, M.H. Garzon, V. Honavar, M. Jakiela, R.E. Smith (Morgan Kaufmann, San Francisco, 1999), pp. 635–642
113. J. Thornton, D.N. Pham, S. Bain, V. Ferreira, Additive versus multiplicative clause weighting for SAT, in *Proceedings of the National Conference on Artificial Intelligence (AAAI)* (AAAI Press, San Jose, 2004), pp. 191–196
114. D.A.D Tompkins, F. Hutter, H.H. Hoos, saps. Solver description. SAT Competition (2007)
115. C. Toregas, R. Swain, C. ReVelle, L. Bergman, The location of emergency service facilities. Oper. Res. **19**(6), 1363–1373 (1971)
116. T. Uchida, O. Watanabe, Hard SAT instance generation based on the factorization problem (2010). http://www.is.titech.ac.jp/~watanabe/gensat/a2/index.html
117. F.J. Vasko, F.E. Wolf, K.L. Stott, Optimal selection of ingot sizes via set covering. Oper. Res. **35**(3), 346–353 (1987)
118. W. Wei, C.M. Li, Switching between two adaptive noise mechanisms in local search for SAT. Solver description. SAT Competition (2009)
119. W. Wei, C.M. Li, H. Zhang, adaptg2wsatp. Solver description. SAT Competition (2007)
120. W. Wei, C.M. Li, H. Zhang, Combining adaptive noise and promising decreasing variables in local search for SAT. Solver description. SAT Competition (2007)
121. W. Wei, C.M. Li, H. Zhang, Deterministic and random selection of variables in local search for SAT. Solver description. SAT Competition (2007)
122. D.H. Wolpert, W.G. Macready, No free lunch theorems for optimization. Evol. Comput. **1**(1), 67–82 (1997)
123. L. Xu, F. Hutter, H.H. Hoos, K. Leyton-Brown, SATzilla2009: an automatic algorithm portfolio for SAT. Solver description. SAT Competition (2009)
124. L. Xu, H.H. Hoos, K. Leyton-Brown. Hydra: automatically configuring algorithms for portfolio-based selection, in *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)* (AAAI Press, Atlanta, 2010)
125. L. Xu, F. Hutter, H.H. Hoos, K. Leyton-Brown, SATzilla-07: the design and analysis of an algorithm portfolio for SAT, in *Proceedings of Principles and Practice of Constraint Programming (CP)*, vol. 4741, ed. by C. Bessiere (Springer, Berlin, Heidelberg, 2007), pp. 712–727.
126. L. Xu, F. Hutter, H.H. Hoos, K. Leyton-Brown, SATzilla: portfolio-based algorithm selection for SAT. J. Artif. Intell. Res. **32**(1), 565–606 (2008)
127. L. Xu, F. Hutter, J. Shen, H.H. Hoos, K. Leyton-Brown, SATzilla2012: improved algorithm selection based on cost-sensitive classification models. SAT Competition (2012)