Lecture Notes in Computer Science 6060

*Commenced Publication in 1973*
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Tapio Elomaa  Heikki Mannila
Pekka Orponen (Eds.)

# Algorithms and Applications

Essays Dedicated to Esko Ukkonen on the Occasion of His 60th Birthday

Springer

Volume Editors

Tapio Elomaa
Tampere University of Technology
Department of Software Systems
P. O. Box 553, 33101 Tampere, Finland
E-mail: elomaa@cs.tut.fi

Heikki Mannila
Aalto University School of Science and Technology
Department of Information and Computer Science
P.O. Box 17800, 00076 Aalto, Finland
E-mail: heikki.mannila@aaltouniversity.fi

Pekka Orponen
Aalto University School of Science and Technology
Department of Information and Computer Science
P.O. Box 15400, 00076 Aalto, Finland
E-mail: pekka.orponen@tkk.fi

Cover illustration:
Artwork by Jussi Ukkonen, Finland (2010)

Library of Congress Control Number: 2010924186

CR Subject Classification (1998): I.2, H.3, J.3, I.5, H.4-5, F.2

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues

ISSN 0302-9743
ISBN-10 3-642-12475-5 Springer Berlin Heidelberg New York
ISBN-13 978-3-642-12475-4 Springer Berlin Heidelberg New York



Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper 06/3180

**Esko Ukkonen**
(The photograph was taken by Joma Marstio 2010)

# Preface

This Festschrift is dedicated to Esko Ukkonen on the occasion of his 60th birthday on January 26, 2010. It contains contributions by his former PhD students and colleagues with whom he cooperated closely within his career. The Festschrift was presented to Esko during a festive symposium organized at the University of Helsinki to celebrate his birthday.

Esko Ukkonen has worked on many areas of computer science, including numerical methods, complexity theory, theoretical aspects of compiler construction, and logic programming. However, his main research interest over the years has been algorithms, with applications. Esko's style of work has been to collaborate closely with scientists from other areas and to study their computational needs. From an understanding of available data the work progresses to the formulation of computational concepts, i.e., finding out what should be computed. The properties of the concepts are then analyzed, algorithms are designed, their behavior is analyzed, the methods are implemented and taken to real applications. This style of work has been very successful throughout his career: Esko has formulated and analyzed many central concepts in computational data analysis. Combining applications and algorithms is also the central theme in the Center of Excellence, Algodan, directed by Esko.

Perhaps the most important scientific areas of Esko Ukkonen are computational pattern matching and string algorithms. He has contributed significantly to the development of these overlapping fields and has helped them to find their own identity. Most of the contributions in this volume concern computational pattern matching or string algorithms.

Esko Ukkonen has had a major role in the development of Finnish computer science. He was the key person in the development of the school of algorithmic research in Finland, and he has had a major role in PhD education. The editors of this volume are grateful to Esko for the insightful guidance that they received from him when they were his PhD students.

January 2010

Tapio Elomaa
Heikki Mannila
Pekka Orponen

**Acknowledgements**

We would like to thank everybody who contributed to this Festschrift: the authors for their interesting articles, the colleagues and PhD students who helped proofread the contributions, Greger Lindén for technical assisstance, and Veli Mäkinen for organizing the seminar to honor Esko's birthday.

# Table of Contents

# String Rearrangement Metrics: A Survey

Amihood Amir[1,2,⋆] and Avivit Levy[3,4]

[1] Department of Computer Science, Bar Ilan University, Ramat Gan 52900, Israel
amir@cs.biu.ac.il
[2] Department of Computer Science, Johns Hopkins University, Baltimore, MD 21218
[3] Shenkar College, Anna Frank 12, Ramat Gan 52526, Israel
avivitlevy@shenkar.ac.il
[4] CRI, University of Haifa, Mount Carmel, Haifa 31905, Israel

**Abstract.** A basic assumption in traditional pattern matching is that the order of the elements in the given input strings is correct, while the description of the content, i.e. the description of the elements, may be erroneous. Motivated by questions that arise in Text Editing, Computational Biology, Bit Torrent and Video on Demand, and Computer Architecture, a new pattern matching paradigm was recently proposed by [2]. In this model, the pattern *content* remains intact, but the relative positions may change. Several papers followed the initial definition of the new paradigm. Each paper revealed new aspects in the world of *string rearrangement metrics*. This new unified view has already proven itself by enabling the solution of an open problem of the mathematician Cayley from 1849. It also gave better insight to problems that were already studied in different and limited situations, such as the behavior of different cost functions, and enabled deriving results for cost functions that were not yet sufficiently analyzed by previous research. At this stage, a general understanding of this new model is beginning to coalesce. The aim of this survey is to present an overview of this recent new direction of research, the problems, the methodologies, and the state-of-the-art.

## 1 Introduction

### 1.1 Motivation

Consider a text $T = t_0 \cdots t_{n-1}$ and pattern $P = p_0 \cdots p_{m-1}$, both over an alphabet $\Sigma$. Traditional pattern matching regards $T$ and $P$ as *sequential* strings, provided and stored in sequence (e.g. from left to right). Therefore, implicit in the conventional approximate pattern matching is the assumption that there may indeed be errors in the **content** of the data, but the **order** of the data is inviolate. However, some non-conforming problems have been gnawing at the walls of this assumption. Selected examples are:

⋆ Partly supported by NSF grant CCR-09-04581 and ISF grant 347/09.

T. Elomaa et al. (Eds.): Ukkonen Festschrift, LNCS 6060, pp. 1–33, 2010.

**Text Editing:** The *swap* error, motivated by the common typing error where two adjacent symbols are exchanged [34,9], does not assume error in the content of the data, but rather, in the order. The data content is, in fact, assumed to be correct. The swap error seemed initially to be akin to the other Levenshtein errors, in that it could be added to the other edit operations and solved with the same dynamic programming [34]. However, when isolated, it turned out to be surprisingly simple to handle [13]. This scarcely seems to be the case for indels or mismatch errors.

**Computational Biology:** During the course of evolution areas of the genome may be shifted from one location to another. Considering the genome as a string over the alphabet of genes, these cases represent a situation where the difference between the original string and resulting one is in the locations rather than contents of the different elements. Several works have considered specific versions of this biological setting, primarily focusing on the sorting problem (*sorting by reversals* [18,19], *sorting by transpositions* [15], and *sorting by block interchanges* [21]).

**Bit Torrent and Video on Demand:** The inherently distributed nature of the web is already causing the phenomenon of transmission of a stream of data in tiny pieces from different sources. This creates the problem of putting scrambled data back together again.

**Computer Architecture:** In computer architecture, it is by no means taken for granted that when seeking a word from a given address, no errors will occur in the address bits [28]. This problem is relevant even when reading a buffer of consecutive words since these words are not necessarily consecutive in the disk or in an interleaved cache[1].

Motivated by these questions a new pattern matching paradigm – *pattern matching with address errors* – was proposed by [2]. In this model, the pattern *content* remains intact, but the relative positions (addresses) may change. The advantages of suggesting and studying a unified general model for all the above examples are:

1. By providing a unified general framework, the relationships between the different problems can be better understood.
2. General techniques can be developed, rather than ad-hoc solutions.
3. Future problems can be more readily analyzed.

Indeed, this unified view has already proven itself by enabling the solution of an open problem of the mathematician Cayley from 1849. It also gave better insight to problems that were already studied in different and limited situations, such as the behavior of different cost functions, and enabled deriving results for cost functions that were not yet sufficiently analyzed by previous research.

Several papers ([1,2,5,7,11,10,30]) followed the initial definition of the new paradigm. Each paper revealed new aspects in the world of *string rearrangement*

[1] Practically, these problems are solved by means of redundancy bits, checksum bits, error detection and correction codes, and communication protocols.

*metrics*. At this stage, a general understanding of this new model is beginning to coalesce. The aim of this survey is to present an overview of this recent new direction of research, the problems, the methodologies, and the state-of-the-art.

### 1.2 The String Rearrangement Model

The novel paradigm being considered, of errors in the *location* of the input elements, rather than their *contents*, raises a plethora of new questions. To better understand the nature of the research directions undertaken so far, as well as to map the possible future paths open to further research, we identify three different thrusts:

1. *What caused the error?* Different phenomena that occur in various diverse applications, cause different types of errors. Interesting such types need to be addressed in the context of approximate pattern matching. Examples of different types of errors in the traditional pattern matching models are the Hamming distance and the edit distance.
2. *What is the error cost?* Even for a given type of error, there can be different error costs. As an example, consider the Hamming distance in the traditional pattern matching model. It assigns the cost of "1" to every mismatch. Nevertheless, different applications make different assignments. If one considers typing errors, then the cost of mismatch in letters that appear in proximity on the keyboard should be less than the cost of distant letters. In a black-and-white image, mismatch in pixels with a close grey-scale level should be less expensive that large distances. Interesting cost measures should be identified and explored.
3. *What set of tools is useful to solve problems in the model?* Various areas develop traditional techniques that lend themselves to cracking the mysteries of the field. Using traditional pattern matching as an example once again, one can point to automata methods, dueling, subword structures, the FFT, or embeddings, as tools to be considered when a problem in the field is addressed. Perusal of the work so far on the string rearrangements model, reveals that these methods have generally not proven useful. Even at this relatively early stage of research in the new model it is interesting to stop and consider if any new methods or data structures seem to be developing.

**Error Causes.** Three types of causes can be identified from the literature for rearrangement errors.

1. *Independent Individual Moves.* In this model every element can independently be shifted and placed in every possible other location. This model is capable of considering situations where elements are objects with independent control. Unlike other models, the positions of the elements are fixed and can be viewed like boxes that should be filled with elements[2]. Indeed it has been studied in some of the early papers [2,5].

[2] In external process models the positions are just the relative order, and therefore, a change in the position of some elements may affect the positions of other elements.

2. *External Process.* The first papers in the literature that dealt with rearrangement operators before the model has been formally identified and defined, were motivated by applications where such rearrangements occur due to an external process [15,18,22,17]. For example, the evolution or mutation process causes genes to relocate, or reverse. The typing process causes adjacent letters to be swapped. In each of these applications, the external process defines the possible or "legal" operators that may cause a symbol to change its location, and only those moves need to be considered for that application.
3. *Internal Process.* We have mentioned applications where the location of a symbol is explicitly given as an address, for example in architecture or bit torrent. In such cases, an internal error may cause the address to be corrupted effecting a relocation of the symbol thus addressed. The types of relocation in this case, then, are the ones that can be caused by "legal" kinds of errors that can plague the address register. Such errors were addressed in [7,10].

**Cost Models:** Three types of cost models have been considered in the context of rearrangement problems.

1. *The Unit-cost model (UCM).* In this model, each operation is given a unit cost, and the problem is to transform $S$ into $T$ with a minimum number of operations. In comparison to traditional pattern matching, this is akin to Hamming distance, or edit distance, where every operation has a cost of "1".
2. *Length-cost model (LCM).* Here, the cost of an operation depends on its length characteristic. For example, moving an element a short distance may cost less than moving an element far away. Such cost models have been considered in other context, and it was noted that their behavior may be different from the UCM [17,16].
3. *Element Cost Model (ECM).* It may be the case that some elements may be "heavier" than other elements. In such cases, moving light elements is preferable to moving heavy elements. In [25], Gupta and Kumar considered the problem of sorting and selection in the comparison model for structured costs. In their work, they assumed that every element has a weight and that the cost of a comparison is defined by a function applied to the weight of the elements that participate in the comparison. Recently, [14] addressed the same problem of sorting and selection for random costs. It is natural to consider rearrangement problems in the ECM [30].

**Tools and Techniques.** Traditional pattern matching is one of the earliest areas of Computer Science, and thus has created a considerable tool kit. The string rearrangement model has been explicitly researched for only half a decade, and thus has not amassed a substantial set of new tools. However, its history does suggest that some of the powerful techniques of traditional pattern matching are inadequate. This is perhaps not surprising since these tools assume that the elements don't "move". They do not perform well on a "moving target". Nevertheless, a trend is already being established, of the tools that have proven successful.

The papers dealing with rearrangements have made massive use of graph theoretic techniques, results, and data structures. In turn, they have been able to reciprocate by shedding new light on some venerable problems in Graph theory. In particular, it has afforded an answer to a question posed by Cayley in 1849 [20] by proving that finding a maximum cardinality decomposition of directed graph into directed cycles is an $\mathcal{NP}$-hard problem.

Another interesting tool seen for the first time in pattern matching, is the use of other types of convolutions that the ubiquitous FFT over the complex field. For example, in [10] use was made of the FFT over $\mathbb{Z}_2$.

### 1.3 Formal Model Definition

*Notations.* Let $x$ be a $m$-long string over alphabet $\Sigma$. The $i$-th character of $x$ is denoted $x[i]$. If $x$ has $m$ distinct elements, it can be viewed as a permutation of $1, \ldots, m$. In this case, it defines a function $\pi : \{1, \ldots, m\} \mapsto \{1, \ldots, m\}$ in the following way: $\pi(i) = j$ if and only if $x[j] = i$. The function $\pi$ is a permutation of $1, \ldots, m$. Thus, in the case that $x$ has $m$ distinct elements, we refer to the permutation $\pi$ as a string, and therefore use the notation $\pi[i]$ for the $i$-th element in $\pi$. Given a permutation $\pi$, we may also use the notation $\pi^{-1}$ for the inverse permutation, i.e., $\pi^{-1}(j) = i$ if and only if $\pi(i) = j$. If $\pi$ is also viewed as a string then $\pi^{-1}(i)$ is exactly $\pi[i]$. The notation of $\pi^{-1}$ is only used when its meaning as an inverse permutation is needed, for example, when we refer to a permutation of the indices of a string with repeating symbols.

*Problem Definition.* Consider a set $\Sigma$ and let $x$ and $y$ be two $m$-tuples over $\Sigma$. We wish to formally define the process of converting $x$ to $y$ through a sequence of *rearrangement* operations.

**Definition 1.** *Let $x, y \in \Sigma^m$, we say that $x$* can be converted to *$y$ if for each $\sigma \in \Sigma$, the number of appearances of $\sigma$ in $x$ equals the number of appearances of $\sigma$ in $y$.*

**Definition 2.** *A* rearrangement operator *$\pi$ is a function $\pi : [0..m-1] \to [0..m-1]$, with the meaning being that for each $i$, $\pi$ moves the element currently at location $i$ to location $\pi(i)$. Let $s = (\pi_1, \pi_2, \ldots, \pi_k)$ be a sequence of rearrangement operators, and let $\pi_s = \pi_1 \circ \pi_2 \circ \cdots \circ \pi_k$ be the composition of the $\pi_j$'s. We say that $s$* converts *$x$ into $y$ if for any $i \in [0..m-1]$, $x_i = y_{\pi_s(i)}$. That is, $y$ is obtained from $x$ by moving elements according to the designated sequence of rearrangement operations.*

*Let $\Pi$ be a set of rearrangement operators, we say that $\Pi$* can convert *$x$ to $y$, if there exists a sequence $s$ of operators from $\Pi$ that converts $x$ to $y$. Given a set $\Pi$ of rearrangement operators, we associate a non-negative* cost *with each sequence from $\Pi$, $cost : \Pi^* \to R^+$. We call the pair $(\Pi, cost)$ a* rearrangement system. *For two vectors $x, y \in A^m$ and a rearrangement system $\mathcal{R} = (\Pi, cost)$, we define the distance from $x$ to $y$ under $\mathcal{R}$ to be:*

$$d_{\mathcal{R}}(x, y) = \min\{cost(s) | s \in \Pi^* \text{ that converts } x \text{ into } y\}$$

*If there is no sequence in $\Pi^*$ that converts $x$ to $y$ then the distance is $\infty$.*

If all elements in $x$ are distinct, a unique bijection $f : x \rightarrow \{1, \ldots, m\}$ can be defined such that $f(x_i)$ equals the position of the element $x_i$ in $y$. Thus $x$ can be represented by $\pi = f(x_1), f(x_2), \ldots, f(x_m)$ and $y$ by $1, \ldots, m$. For this case the term *permutation string* is used. The input string is then assumed to be $\pi$, i.e, a permutation of $1, \ldots, m$. Under this assumption the rearrangement problem is simply a sorting problem, i.e. the distance is the minimum cost for sorting $\pi$. Problems of sorting a *permutation string* have been studied extensively(e.g. [15,18,19,21,26,27]). For the general case in which $x$ may have repetitions of elements, the term *general string* is used.

*The String Matching Problem.* Let $\mathcal{R}$ be a rearrangement system and let $d_{\mathcal{R}}$ be the induced distance function. Consider a text $T = T[0], \ldots, T[n-1]$ and pattern $P = P[0], \ldots, P[m-1]$ $(m \leq n)$. For $0 \leq i \leq n-m$ denote by $T^{(i)}$ the $m$-long substring of $T$ starting at location $i$. Given a text $T$ and pattern $P$, we wish to find the index $i$ such that $d_{\mathcal{R}}(P, T^{(i)})$ is minimal.

*Paper Organization.* The rest of the paper is organized as follows. In Sect. 2 we discuss rearrangement systems where the operators are independent individual moves. In Sect. 3 we discuss rearrangement systems where the operator is defined by an external process. Specifically, we discuss the interchange operator under the various cost models: UCM, LCM and ECM, but we also mention other operators. Finally, in Sect. 4 we discuss rearrangement systems where the operator is defined by an internal process.

## 2 Independent Individual Moves

In this section we consider the independent individual move model, which allows any element to be inserted at any other location. However, we now consider various length-cost metrics.

Under the $\ell_1$ *Rearrangement System*, the cost of such a rearrangement is the sum of the distances the individual elements have been moved. Formally, let $x$ and $y$ be strings of length $m$. A rearrangement under the $\ell_1$ distance is a permutation $\pi : [0..m-1] \rightarrow [0..m-1]$, where the cost is $cost(\pi) = \sum_{j=0}^{m-1} |j - \pi(j)|$. We call the resulting distance the $\ell_1$ *Rearrangement Distance.*

In the $\ell_2$ *Rearrangement System* we use the same set of operators, with the cost being the sum of squares of the distances the individual elements have moved.[3] Formally, let $x$ and $y$ be strings of length $m$. A rearrangement under

[3] For simplicity of exposition we omit the square root usually used in the $\ell_2$ distance. This does not change the complexity, since the square root operation is monotone, and can be computed at the end.

the $\ell_2$ distance is a permutation $\pi : [0..m-1] \to [0..m-1]$, where the cost is $cost(\pi) = \sum_{j=0}^{m-1} |j - \pi(j)|^2$. We call the resulting distance the $\ell_2$ *Rearrangement Distance*.

In the $\ell_\infty$ *Rearrangement System*, the cost is the maximum of the distances the individual elements have been moved. Formally, let $x$ and $y$ be strings of length $m$. A rearrangement under the $\ell_\infty$ distance is a permutation $\pi : [0..m-1] \to [0..m-1]$, where the cost is $cost(\pi) = \max_{j \in \{0,\ldots,m-1\}} |j - \pi(j)|$. We call the resulting distance the $\ell_\infty$ *Rearrangement Distance*.

In the rest of the section we describe the basic solutions for the $\ell_1$ and $\ell_2$ rearrangement systems. We also briefly mention the results for the $\ell_\infty$ rearrangement system.

### 2.1 The $\ell_1$ Rearrangement Distance

Let $x$ and $y$ be strings of length $m$. Clearly, if $x$ contains distinct elements then only one permutation can convert $x$ to $y$. However, there if $x$ and $y$ contains elements that appear multiple times, then there can be several different permutations that can convert $x$ to $y$. Intuitively, of these permutations, the least cost is obtained by the one that does not change the order among identical letters (see Fig. 1). The following lemma proves that this is indeed the case.

**Lemma 1.** *Let $x, y \in \Sigma^m$ be two strings such that $d_{\ell_1}(x, y) < \infty$. Let $\pi_o$ be the permutation that for each $a \in \Sigma$ and each $k$, moves the $k$-th $\sigma$ of $x$ to the location of the $k$-th $\sigma$ of $y$. Then,*

$$d_{\ell_1}(x, y) = cost(\pi_o).$$

*i.e. $\pi_o$ is a permutation of the least cost.*

*Proof.* For a permutation $\pi$, and $i < j$ such that $x[i] = x[j]$, say that $\pi$ *reverses $i$ and $j$* if $\pi(i) > \pi(j)$. Note that $\pi_o$ is characterized by having no *reversals*. We show that it has the least cost. Let $\tau$ be a permutation converting $x$ to $y$ of minimal cost that has the minimal number of reversals. If there are no reversals

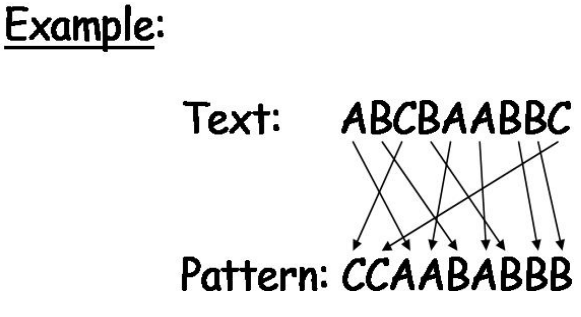



**Fig. 1.** The minimal cost pairing permutation $\pi_o$

in $\tau$, then there is nothing to prove, since it is exactly the permutation $\pi_o$. Otherwise, suppose $\tau$ reverses $j$ and $k$ ($j < k$). Let $\tau'$ be the permutation which is identical to $\tau$, except that $\tau'(j) = \tau(k)$ and $\tau'(k) = \tau(j)$. Then, clearly $\tau'$ also converts $x$ to $y$. We show that $cost(\tau') \leq cost(\tau)$. There are two cases:

**Case 1:** $\tau(j) \geq k$ or $\tau(k) \leq j$. Consider the case $\tau(j) \geq k$. Then clearly $\tau(j) > j$, hence:

$cost(\tau) - cost(\tau') =$
$= |\tau(j) - j| + |\tau(k) - k| - |\tau'(j) - j| - |\tau'(k) - k|$
$= |\tau(j) - j| + |\tau(k) - k| - |\tau(k) - j| - |\tau(j) - k|$
$= (\tau(j) - j) + |\tau(k) - k| - |\tau(k) - j| - (\tau(j) - k)$
$= \tau(j) - j + |\tau(k) - k| - |\tau(k) - j| - (\tau(j) - k)$
$= (k - j) + |\tau(k) - k| - |\tau(k) - j|$
$\geq |(k - j) + (\tau(k) - k)| - |\tau(k) - j| = 0.$

The argument for $\tau(k) \leq j$ is symmetrical.

**Case 2:** $j < \tau(k) < \tau(j) < k$. Then,

$cost(\tau) - cost(\tau') =$
$= |\tau(j) - j| + |\tau(k) - k| - |\tau'(j) - j| - |\tau'(k) - k|$
$= |\tau(j) - j| + |\tau(k) - k| - |\tau(k) - j| - |\tau(j) - k|$
$= (\tau(j) - j) + (k - \tau(k)) - (\tau(k) - j) - (k - \tau(j))$
$= 2(\tau(j) - \tau(k)) > 0.$

Thus, the cost of $\tau'$ is at most that of $\tau$, and there is one reversal less in $\tau'$, in contradiction. □

Thus, in order to compute the $\ell_1$ distance of $x$ and $y$, we create for each symbol $a$ two lists, $\psi_a(x)$ and $\psi_a(y)$, the first being the list of locations of $a$ in $x$, and the other – the locations of $a$ in $y$. Both lists are sorted. These lists can be created in linear time[4]. Clearly, if there exists an $a$ for which the lists are of different lengths then $d_{\ell_1}(x, y) = \infty$. Otherwise, for each $a$, compute the $\ell_1$ distance between the corresponding lists, and sum over all $a$'s. This provides a linear time algorithm for strings of identical lengths, and an $O(m(n-m+1))$ algorithm for the general case.

**Theorem 1.** [2] *For $T$ and $P$ of sizes $n$ and $m$ respectively ($m \leq n$), the $\ell_1$ Rearrangement Distance can be computed in time $O(m(n-m+1))$. If all entries of $P$ are distinct, then the distance can be computed in time $O(n)$.*

The $\ell_1$ rearrangement distance can be approximated efficiently with high probability.

**Theorem 2.** [5] *For $T$ and $P$ of sizes $n$ and $m$ respectively ($m \leq n$), then for any $\epsilon > 0$ and $0 < \delta < 1$ there exists a constant $c = c(\epsilon, \delta)$ such that the $\ell_1$ Rearrangement Distance can be approximated to $\pm\epsilon$ in time $O(n \cdot c/\epsilon^2 \log 1/\delta)$ with probability $1 - \delta$.*

[4] Clearly, we can consider only the letters in $P$, and hence sorting can be completed in linear time.

### 2.2 The $\ell_2$ Rearrangement Distance

Interestingly, the $\ell_2$ distance can be computed much more efficiently.

Consider first the case of equal length sequences. Let $x$ and $y$ be strings of length $m$. The following lemma, characterizing the minimal cost permutation converting $x$ to $y$, is the $\ell_2$ analogue of Lemma 1. The proof is analogous to the proof of Lemma 1.

**Lemma 2.** [2] *Let $x, y \in \Sigma^m$ be two strings such that $d_{\ell_2}(x, y) < \infty$. Let $\pi_o$ be the permutation that for all $a$ and $k$, moves the $k$-th $a$ in $x$ to the location of the $k$-th $a$ in $y$. Then,*

$$d_{\ell_2}(x, y) = cost(\pi_o).$$

*I.e. $\pi_o$ is a permutation of the least cost.*

Now that we are guaranteed that $\pi_o$ provides the minimum distance, we need to compute $cost(\pi_o)$. In the case that $x$ and $y$ are of the same length, the cost can be computed in the following manner. Consider an element $a \in \Sigma$, and let $\mathrm{occ}_a(x)$ be the number of occurrences of $a$ in $x$. Note that if $x$ can be converted to $y$ then necessarily $\mathrm{occ}_a(x) = \mathrm{occ}_a(y)$. Let $\psi_x(a)$ be the sorted sequence (of length $\mathrm{occ}_a(x)$) of locations of $a$ in $x$. Similarly $\psi_a(y)$ is this sequence for $y$. Then,

$$cost(\pi_o) = \sum_{a \in x} \sum_{j=0}^{\mathrm{occ}_a(x)-1} (\psi_a(x)[j] - \psi_a(y)[j])^2. \tag{1}$$

Since $\sum_{a \in x} \mathrm{occ}_a(x) = m$, the above sum can be computed in linear time.

Consider, now, a text $T$ of length $n$ and a pattern $P$ of length $m$. We wish to compute the $\ell_2$ distance of $P$ to each text substring $T^{(i)}$ (where $T^{(i)}$ is the $m$-long substring of $T$ starting at position $i$). First note that by simple counting we can find all locations for which the distance is $\infty$, i.e. the locations for which there is no way to convert the one string to the other. Thus, we need only consider the substrings $T^{(i)}$ that are a permutation of $P$. For these substrings, $\mathrm{occ}_a(P) = \mathrm{occ}_a(T^{(i)})$ for all $a \in P$.

We can certainly compute the distances by repeatedly applying the algorithm for equal lengths strings presented above, but the total time would be $O(nm)$. However, we can obtain a much faster algorithm as follows. Consider a symbol $a$, and let $\psi_a(P)$ and $\psi_a(T)$ be the sorted lists of locations of $a$ in $P$ and $T$, respectively. Note that these two lists need not be of the same length. Similarly, let $\psi_a(T^{(i)})$ be the list of locations of $a$ in $T^{(i)}$. Then, by Equation (1), for any $T^{(i)}$ (which is a permutation of $P$):

$$d_{\ell_2}(P, T^{(i)}) = \sum_{a \in P} \sum_{j=0}^{\mathrm{occ}_a(P)-1} (\psi_a(P)[j] - \psi_a(T^{(i)})[j])^2 \tag{2}$$

We now wish to express the above sum using $\psi_a(T)$ instead of the individual $\psi_a(T^{(i)})$'s. Note that all the $a$'s referred to in $\psi_a(T^{(i)})$ are also referred to in

$\psi_a(T)$. However, $\psi_a(T)$ gives the locations with respect to the beginning of $T$, whereas $\psi_a(T^{(i)})$ gives the locations with respect to the beginning of $T^{(i)}$ – which is $i$ positions ahead.

For each $i$ and $a$, let $match_a(i)$ be the index of the smallest entry in $\psi_a(T)$ with value at least $i$. Then, $match_a(i)$ is the first entry in $\psi_a(T)$ also referenced by $\psi_a(T^{(i)})$ (assuming the $a$ appears in $T^{(i)}$). For any $a, i$ and $j \leq \text{occ}_a(P)$, we have:

$$\psi_a(T^{(i)})[j] = \psi_a(T)[match_a(i) + j] - i.$$

Thus, Equation (2) can be rewritten as:

$$d_{\ell_2}(P, T^{(i)}) = \sum_{a \in P} \sum_{j=0}^{\text{occ}_a(P)-1} \left(\psi_a(P)[j] - (\psi_a(T)[match_a(i) + j] - i)\right)^2 \quad (3)$$

We wish to compute this sum for all $i$. We do so by a combination of convolution and polynomial interpolation, as follows.

*The Values of $match_a(i)$.* We first show how to efficiently compute $match_a(i)$ for all $a$ and $i$. Consider two consecutive locations $i$ and $i+1$. Let $T[i]$ be the symbol at the $i$-th location in $T$. Then,

$$match_a(i+1) = \begin{cases} match_a(i) + 1 & a = T[i] \\ match_a(i) & a \neq T[i] \end{cases} \quad (4)$$

Equation (4) allows us to incrementally compute $match_a(i)$ for all $i$. That is, if we know $match_a(i)$ for all $a$, then we can also know $match_a(i+1)$, for all $a$, in $O(1)$ steps.

*The Functions $G_x$ and $F_x$.* Fix a number $x$, and suppose that instead of the computing the sum in Equation (3), we want to compute the sum:

$$G_x(i) = \sum_{a \in P} \sum_{j=0}^{\text{occ}_a(P)-1} \left(\psi_a(P)[j] - (\psi_a(T)[match_a(i) + j] - x)\right)^2$$

This is the same sum as in Equation (3), but instead of subtracting $i$ in the parenthesis, we subtract the fixed $x$. The important difference is that now $x$ is independent of $i$. Note that by Equation (3) $d_{\ell_2}(P, T^{(i)}) = G_i(i)$.

For $a, k$ let

$$F_x(a, k) = \sum_{j=0}^{\text{occ}_a(P)-1} \left(\psi_a(P)[j] - (\psi_a(T)[k + j] - x)\right)^2$$

Then,

$$G_x(i) = \sum_{a \in P} F_x(a, match_a(i)) \quad (5)$$

Suppose that for a fixed $x$ we have pre-computed $F_x(a,k)$ for all $a$ and $k$. We show how to compute $G_x(i)$ for all $i$ (for the fixed $x$). We do so by induction. For $i=0$ we compute $G_x(i)$ using Equation 5 in $O(m)$ steps. Suppose we have computed $G_x(i)$ and now wish to compute $G_x(i+1)$. Then,

$$G_x(i) = \sum_{a \in P} F_x(a, match_a(i))$$

while

$$G_x(i+1) = \sum_{a \in P} F_x(a, match_a(i+1))$$

However, by Equation (4), for most of the $a$'s $match_a(i+1) = match_a(i)$ and for $a = T[i]$, $match_a(i+1) = match_a(i) + 1$. Thus,

$$G_x(i+1) - G_x(i) = -F_x(T[i], match_{T[i]}(i)) + F_x(T[i], match_{T[i]}(i)+1)$$

Thus, assuming that $G_x(i)$ is known, and that all $F_x(a,k)$'s have been pre-computed, $G_x(i+1)$ can be computed in $O(1)$ steps. (The values of $match_a(i)$ are incrementally computed as we advance from $i$ to $i+1$.)

*Computing $F_x(a,k)$.* We now show how to compute $F_x(a,k)$ for all $a$ and $k$. We do so using the following general lemma:

**Lemma 3.** [3] *Let $Q$ and $W$ be two sequences of real numbers, with lengths $|Q|$ and $|W|$, respectively ($|Q| \leq |W|$). Let $p(q,w)$ be a polynomial in two variables, and $t$ an integer ($t \leq |Q|$). For $i = 0, \ldots, |W| - |Q|$, let $P_{Q,W}(i) = \sum_{j=0}^{t-1} p(Q[j], W[i+j])$. Then, $P_{Q,W}(i)$ can be computed for all $i$'s together, in $O(|W| \log |Q|)$ steps.*

Applying the lemma to our setting let $p(q,w) = (q-w+x)^2$, $t = \mathrm{occ}_a(P)$, $Q = \psi_a(P)$ and $W = \psi_a(T)$. Then, $F_x(a,k) = \sum_{j=0}^{t-1} p(Q[j], W[k+j])$. Thus, $F_x(a,k)$ can be computed for all $k$'s together in $O(\mathrm{occ}_a(T) \log(\mathrm{occ}_a(P)))$ steps. Combining for all $a$'s, the computation takes:

$$\sum_{a \in P} O(\mathrm{occ}_a(T) \log(\mathrm{occ}_a(P))) = O(n \log m)$$

(since $\sum_{a \in P} \mathrm{occ}_a(T) \leq n$ and $\sum_{a \in P} \mathrm{occ}_a(P) = m$).

*From $G_x(i)$ to $d_{\ell_2}(P, T^{(i)})$.* We have so far seen that for any fixed $x$, we can compute $G_x(i)$ for all $i$ in $O(n \log m)$ steps. Recall that $d_{\ell_2}(P, T^{(i)}) = G_i(i)$. Thus, we wish to compute $G_i(i)$ for all $i$. For any fixed $i$, considering $x$ as a variable, $G_x(i)$ is a polynomial in $x$ of degree $\leq 2$. Thus, if we know the value of $G_x(i)$ for three different values of $x$, we can then compute its value for any other $x$ in a constant number of steps using polynomial interpolation. Thus, in order to compute $G_i(i)$ we need only know the value of $G_x(i)$ for three arbitrary values of $x$, say $0, 1$ and $2$. Accordingly, we first compute $G_0(i), G_1(i)$ and $G_2(i)$,

for all $i$ in $O(n \log m)$ time, as explained above. Then, using interpolation, we compute $G_i(i)$ for each $i$ separately, in $O(1)$ steps per $i$. The total complexity is thus $O(n \log m)$.

We summarize:

**Theorem 3.** *For $T$ and $P$ of sizes $n$ and $m$ respectively ($m \leq n$) the $\ell_2$ Rearrangement Distance can be computed in time $O(n \log m)$.*

### 2.3 The $\ell_\infty$ Rearrangement Distance

The $\ell_\infty$ case, as the $\ell_1$ case, has not yielded a deterministically efficient algorithm. However, using the techniques developed for the computation of the $\ell_1$ distance and a property of the $\ell_\infty$ distance that is similar to Lemma 1, we get the following result.

**Theorem 4.** [5] *For $T$ and $P$ of sizes $n$ and $m$ respectively ($m \leq n$) the $\ell_\infty$ Rearrangement Distance can be computed in time $O(m(n-m+1))$. If all entries of $P$ are distinct then the $\ell_\infty$ Rearrangement Distance can be computed in time $O(n \log m)$.*

However, there is an efficient approximation algorithm. This approximation algorithm utilizes a connection between the $\ell_\infty$ distance and a generalization of the $\ell_2$ distance. As a side effect it generalizes the technique use to compute the $\ell_2$ distance.

**Theorem 5.** [5] *For $T$ and $P$ of sizes $n$ and $m$ respectively ($m \leq n$), the $\ell_\infty$ Rearrangement Distance can be approximated to a factor of $1+\varepsilon$ in time $O(\frac{1}{\varepsilon^2} n \log^3 m)$.*

### 2.4 Independent Individual Moves Rearrangements: Summary

Table 1 summarizes the results on independent individual moves rearrangement systems as studied in [2,5]. The table refers to the string matching version of the problem, where the text is assumed to be longer than the pattern. In the case that the text and pattern are of equal size, computations of the $\ell_1$, $\ell_2$ and $\ell_\infty$ distances can be trivially done in linear time.

**Table 1.** Independent Individual Moves Rearrangement Systems: A Summary of Results

| The System | Patterns with Distinct Elements | General Patterns | Approximation |
|---|---|---|---|
| $\ell_1$ | $O(n)$ [2] | $O(nm)$ [2] | $O(n \cdot c/\epsilon^2 \log 1/\delta)$ $\pm\epsilon$-approx. $\star$ [6] |
| $\ell_2$ | $O(n)$ [3] | $O(n \log m)$ [2] | – |
| $\ell_\infty$ | $O(n \log m)$ [6] | $O(nm)$ [6] | $O(\frac{1}{\varepsilon^2} n \log^3 m)$ $1+\varepsilon$-approx. [6] |

$\star$ $c = c(\epsilon, \delta)$ is a constant. The approximation is within the given range with probability $1-\delta$.

## 3 External Process Rearrangement Systems

In this section we consider rearrangement systems where the rearrangement operators are defined by an external process. Specifically, we discuss the interchange rearrangement problem under various cost models. We also discuss the related parallel-interchange rearrangement system. The *interchange rearrangement problem* is the following: Given two strings $x, y$ over alphabet $\Sigma$ such that $x, y$ have the same quantity of each symbol, the goal is to transform $x$ (called the input string) to $y$ (called the target string) using a succession of interchange operations. An *interchange* of two elements, $a$ in position $i$ and $b$ in position $j$, puts element $a$ in position $j$ and element $b$ in position $i$. The *interchange distance problem* is to find the minimum number of interchanges needed to transform the input string $x$ to the target string $y$.

The interchange distance problem defined by [2], is actually a classical problem mentioned back in 1849 by Cayley [20]. Cayley mainly studied permutation strings, in which all elements are distinct. In that case, strings can be viewed as permutations of $1, \ldots, m$, where $m$ is the length of the string. This classical setting was well studied (e.g., [20,29]). Cayley [20] gives a characteristic theorem for the distance in this case, from which a simple linear time algorithm for computing it on permutation strings can be immediately derived, as described in [2]. However, these results do not apply for the general strings case, posed as an open problem by Cayley. [11] studied a generalization of this classical and well-studied problem on permutations by considering general strings as input and examining various cost models.

*Formal Definition of the Problem.* In the sequel, we give some formal definitions, including a formal definition of an *interchange* and the *interchange distance problem in w-cost model*.

The *interchange operator* is a special rearrangement operator.

**Definition 3.** *The* interchange *rearrangement operator is a function* $op : \Sigma^m \times \{1, \ldots, m\} \times \{1, \ldots, m\} \mapsto \Sigma^m$*, such that*

$$op(x, i, j) = \begin{cases} x[1, \ldots, i-1] \cdot x[j] \cdot x[i+1, \ldots, j-1] \cdot x[i] \cdot x[j+1, \ldots, m], & \text{if } i < j; \\ x[1, \ldots, j-1] \cdot x[i] \cdot x[j+1, \ldots, i-1] \cdot x[j] \cdot x[i+1, \ldots, m], & \text{if } i > j; \\ x, & \text{if } i = j. \end{cases}$$

**Definition 4.** *Let* $w : \mathbb{N} \mapsto \mathbb{R}$ *be a cost function such that* $w(0) = 0$*,* $x, y \in \Sigma^m$ *be two strings such that* $x$ *can be converted to* $y$*, and let* $s = s_1, \ldots, s_k$ *be a sequence of interchanges that converts* $x$ *to* $y$*, where* $s_j$ *interchanges elements in positions* $i_j, i'_j$*, then* $cost(s) = \sum_j w(|i_j - i'_j|)$*. The* interchange distance problem in $w$ cost model *(or the w-interchange distance problem) is to compute* $d_{WI(w)}(x, y) = \min\{cost(s) \mid s \text{ converts } x \text{ to } y\}$[5]*. The* interchange distance problem *is simply the interchange distance problem in the* unit cost model*, i.e.,* $w(\ell) = 1$ *for every* $\ell > 0$*.*

[5] If $x$ cannot be converted to $y$, define $d_{WI(w)}(x, y) = \infty$. In this paper, it is always assumed that $x$ can be converted to $y$, thus, $d_{WI(w)}(x, y) < \infty$.

A special case of the interchange distance problem is the *sorting by interchanges problem* defined as follows.

**Definition 5.** *Let $w : \mathbb{N} \mapsto \mathbb{R}$ be a cost function. Let $\Sigma$ be an alphabet and $R$ a total order defined on the elements of $\Sigma$. Let $x$ be a string over alphabet $\Sigma$ and let $s = s_1, \ldots, s_k$ be a sequence of interchanges that sorts $x$ according to $R$ (resulting in the string $x_R$), where $s_j$ interchanges elements in positions $i_j, i'_j$, then cost$(s) = \sum_j w(|i_j - i'_j|)$. The* sorting by interchanges problem in $w$ cost model *is to compute $d_{WI(w)}(x, x_R) = \min\{cost(s) \mid s$ sorts $x$ according to $R\}$.*

### 3.1 The Unit Cost Model

We begin by discussing the interchange and parallel-interchange rearrangement systems in the unit cost model.

**The Interchange Distance Problem.** We show that this problem is equivalent to the problem of finding the cardinality of the maximum edge-disjoint cycle decomposition of Eulerian directed graphs (denoted by maxDCD). The later problem has been shown to be $\mathcal{NP}$-hard [11].

*Equivalence to maxDCD.* Given two strings $x, y \in \Sigma^m$ such that $x$ can be converted to $y$, it is possible to derive two permutations of $1, \ldots, m$, as follows. Let $\mathcal{S}_m$ be the set of all permutations of $1, \ldots, m$. A *labelling* of an $m$-long string over alphabet $\Sigma$ is a function $L : \Sigma^m \mapsto \mathcal{S}_m$. We now formally define a *legal* labelling $L_{x,y}$ for the strings $x$ and $y$. Let $I \in \mathcal{S}m$ be the identity permutation (i.e., the string $1, \ldots, m$). The target string $y$ is labelled as the identity permutation, i.e., $L_{x,y}(y) = I$. For every $\sigma \in \Sigma$ denote by $L_y^\sigma$ the set $\{i \in \{1, \ldots, m\} \mid y[i] = \sigma\}$. Similarly, $L_x^\sigma = \{i \in \{1, \ldots, m\} \mid x[i] = \sigma\}$. For the definition of a legal labelling of the input string $x$, consider the sets $L_x^\sigma$ and $L_y^\sigma$ for each $\sigma$. Since $x$ can be converted to $y$, we have $|L_x^\sigma| = |L_y^\sigma|$. Let $f_\sigma$ be a bijection, $f_\sigma : L_x^\sigma \mapsto L_y^\sigma$. Now, a legal labelling of $x$ is $L_{x,y}(x) = f_{x[1]}(1) f_{x[2]}(2) \ldots f_{x[m]}(m)$. Note that $L_{x,y}(x)$ is a permutation of $1, \ldots, m$. A legal labelling can also be viewed as a *pairing* between the positions of elements in the input string and the target string.

The distance of the permutation $L_{x,y}(x)$ (from the identity permutation) is characterized by Fact 6. Of course, different choices for bijections $f_\sigma$ yield a different permutation of $1, \ldots, m$, $L_{x,y}(x)$. The interchange distance of the strings $x$ and $y$ is achieved by moving each misplaced element in the input string $x$ to one of the positions in the target string $y$ where this element appear. It can, therefore, be viewed as defining a legal labelling of the input string as a permutation of $1, \ldots, m$. Observation 1 specifies the connection between the interchange distance of a permutation resulting from a legal labelling and the interchange distance of the original strings $x$ and $y$.

**Theorem 6.** [Cayley] [20] *The interchange distance of an $m$-length permutation $\pi$ is $m - c(\pi)$, where $c(\pi)$ is the number of permutation cycles in $\pi$.*

**Observation 1.** *There exists a legal labelling for which the interchange distance between the resulting permutation of $1, \ldots, m$ is exactly the interchange distance*

*between the given m-length strings. Moreover, the permutation resulting from this labelling has the minimum interchange distance over every permutation resulting from any other labelling.*

**Definition 6.** *maxDCD is the following problem: Given an Eulerian directed graph $G = (V, E)$, find maximum-cardinality edge-disjoint directed cycle decomposition of G, i.e., partition of E into the maximum number of mutually edge-disjoint directed cycles.*

**Lemma 4.** *The interchange distance problem and the maxDCD problem can be transformed to each other in linear time.*

*Proof.* We describe a linear-time transformation from the interchange distance problem to maxDCD. Let $x, y$ be two $m$-length strings over alphabet $\Sigma$, such that $|\Sigma| \leq m$, and $x$ can be converted to $y$. We construct the directed graph $G = (V, E)$, where $V = \Sigma$ and $E = \{e_i = (a, b) \mid 1 \leq i \leq m, x[i] = b, y[i] = a\}$. $G$ is an Eulerian directed graph, since $x$ can be converted to $y$, thus for every vertex in $G$ the in- and out-degree are equal. The maximum edge-disjoint cycle decomposition of $G$ includes only cycles with distinct vertices, since, if a vertex appears twice in a cycle, break it into two different cycles: one for each appearance of the repeating vertex ($G$ is Eulerian). Thus, in the rest of the proof we only consider cycles with distinct vertices. Every edge-disjoint cycle decomposition of $G$ into cycles with distinct vertices defines a labeling of the strings symbols (with different labels to repeating symbols), resulting in permutations of $1, \ldots, m$, where $y$ defines the identity permutation. For the permutation derived from $x$ the graph cycles represent the permutation cycles. Denote the number of permutation cycles in a decomposition $D_G$ by $c(D_G)$. By Fact 6 the interchange distance between the permutations derived from $x$ and $y$ is exactly $m - c(D_G)$. Finally, denote by $|maxDCD(G)|$ the cardinality of the maximum cycle decomposition of the directed graph $G$. Then, by Observation 1 the interchange distance between $x$ and $y$ is exactly $m - |maxDCD(G)|$.

The inverse transformation from maxDCD to the interchange distance problem for general strings is similar. Given an Eulerian directed graph $G = (V, E)$, we construct $x, y$, two $m$-length general strings of symbols from alphabet $\Sigma$, where $m = |E|$ and $|\Sigma| = |V|$, such that $x$ can be converted to $y$. Let $e_1, e_2, \ldots, e_{|E|}$ be any order of the edges in $G$. For all $1 \leq i \leq m$, define $x[i] = b, y[i] = a$ if $e_i = (a, b)$. Since this is the same transformation, only inversely built, we get as above that the interchange distance between $x$ and $y$ is exactly $m - |maxDCD(G)|$. □

In order to complete the $\mathcal{NP}$-hardness proof we need the following theorem.

**Theorem 7.** [11] *The maxDCD problem is $\mathcal{NP}$-hard.*

**The Parallel-Interchanges Distance Problem.** Next we consider a rearrangement system where multiple pairs can be interchanged in parallel, i.e. in any given step any number of pairs can be interchanged but an element can participate in at most one interchange. The cost of a sequence is the number of such

parallel steps. We call the resulting distance the *parallel interchange distance*, denoted by $d_{p\text{-}interchange}(\cdot,\cdot)$ . [2] prove:

**Theorem 8.** *For any two strings $x$ and $y$, either $d_{p\text{-}interchange}(x,y) = \infty$ or $d_{p\text{-}interchange}(x,y) \leq 2$.*

This means that if it is altogether possible to convert $x$ to $y$, then it is possible to do so in at most two parallel steps of interchange operations!

With regards to computing the distance [2] prove:

**Theorem 9.** *For $T$ and $P$ of sizes $n$ and $m$ respectively ($m \leq n$), if there are $k$ distinct entries in $P$, then the parallel interchange distance can be computed deterministically in time $O(k^2 n \log m)$.*

**Theorem 10.** *For $T$ and $P$ of sizes $m$ and $n$ respectively ($m \leq n$), the parallel interchange distance can be computed randomly in expected time $O(n \log m)$.*

Below we describe how these results were obtained.

*Bounding the Parallel Interchange Distance.* Previously we saw that a cycle of length $\ell$ can be sorted by $\ell - 1$ interchanges. We now ask what is the minimal number of parallel interchange steps required for this sorting. Surprisingly, the next lemma shows that with a careful choice of the interchanges, we can always sort with at most two parallel steps.

**Lemma 5.** *Let $\sigma$ be a cycle of length $\ell > 2$. It is possible to sort $\sigma$ in two parallel interchanges steps.*

*Proof.* W.l.o.g. the string is $(1, 2, 3, \ldots, \ell-2, \ell-1, 0)$ and has to be converted to $(0, 1, \ldots, \ell - 1)$. In the first parallel step we invert the segment $(1, 2, \ldots, \ell - 1)$, namely perform the $(\ell-1)/2$ interchanges $(1, \ell-1)$, $(2, \ell-2)$, etc. The resulting string is $(\ell - 1, \ell - 2, \ell - 3, \ldots, 3, 2, 1, 0)$, from which the sorted string can be obtained in one additional parallel step (containing $\ell/2$ interchanges): $(0, \ell - 1)$, $(1, \ell - 2), \ldots$ (see Fig. 2). □

Since different cycles can be sorted in parallel, we obtain Theorem 8.

*Computing the Parallel Interchange Distance.* By theorem 8 there are only four different possibilities for the parallel-interchange distance between a pattern and a text, namely: 0, 1, 2 or $\infty$. Thus, in order to compute the distance, we need only check which of the four is the correct one. Distance 0 signifies an exact match, and can be found in $O(n)$ steps using standard techniques. Distance $\infty$ means that at each text location $i$, the strings $P$ and $T^{(i)}$ either contain different symbols, or with different multiplicity. This can again be computed in $O(n)$ steps by simple counting. Thus, it remains to be able distinguish between distances 1 and 2. We show how to check for distance 1.

We start by describing a deterministic algorithm. If two strings have distance 1, then we say that one is a *parallel interchange* of the other. For each $i$ and pair

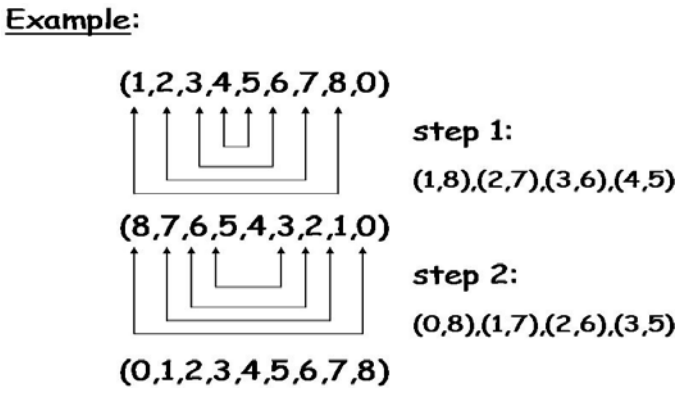



**Fig. 2.** The structure of the parallel interchanges sorting a permutation cycle

of alphabet symbols $(a, b)$, we count the number of times that $a$ appears in the pattern and $b$ appears in the corresponding location in the text $T^{(i)}$. Then, $P$ is a parallel interchange of $T^{(i)}$ if and only if for all $a, b$, the count for $(a, b)$ equals that for $(b, a)$. This count can be implemented by convolutions in the following manner.

Let $S$ be a string over alphabet $\Sigma$ and let $a \in \Sigma$. Denote by $\chi_a(S)$ the binary string of length $|S|$ where every occurrence of $a$ is replaced by 1 and every occurrence of any other symbol is replaced by 0. The dot product of $\chi_a(T^{(i)})$ with $\chi_b(P)$ gives precisely the number of times an $a$ in $T^{(i)}$ is aligned with a $b$ in $P$. This number can be computed for all alignments of the pattern with the text in time $O(n \log m)$ using convolutions [24]. Clearly, it is sufficient to consider only symbols from $\Sigma_P$. Thereby we obtain that the parallel interchange distance can be computed deterministically in time $O(|\Sigma_P|^2 n \log m)$ (Theorem 9).

For unbounded alphabets this is not very helpful. So, we seek a further speedup via randomization. The idea is to view the symbols of the alphabet as symbolic variables, and use the Schwartz-Zippel Lemma [35,37], as follows. For variables $a, b$, let $h(a, b) = a^2b - b^2a$. Note that $h(a, a) = 0$ and $h(a, b) = -h(b, a)$. Given two strings $x, y \in A^m$ define the polynomial:

$$H_{x,y} = \sum_{j=0}^{m-1} h(x_j, y_j)$$

Then:

**Lemma 6.** [3] *Given two strings $x, y \in A^m$, $H_{x,y} \equiv 0$ (i.e. $H_{x,y}$ is the all zeros polynomial) iff $x$ is a parallel interchange of $y$.*

Thus, for each text location $i$, we wish to check if $H_{P,T^{(i)}} \equiv 0$. We do so by randomly assigning numeric values to the symbolic variables, and using the Schwartz-Zippel Lemma. Specifically, each variable is assigned a random value chosen uniformly and independently at random from the set $\{1, \ldots, 3m\}$. Let

$r$ be the random assignment. Then by the Schwartz-Zippel lemma, for any $x$ and $y$, $\Pr[H_{x,y}(r) = 0 | H_{x,y} \not\equiv 0] \leq \frac{deg(H_{x,y})}{3m} = \frac{1}{m}$. Clearly, if $H_{x,y} \equiv 0$ then $H_{x,y}(r) = 0$ for all $r$. Accordingly, given the random assignment $r$, we compute the value of $H_{P,T^{(i)}}(r)$, for all $i$. If the value is different from 0 for all $i$, then clearly there is no parallel interchange of the pattern in the text, and the distance cannot be 1. Otherwise, we check one by one each location $i$ for which $H_{P,T^{(i)}}(r) = 0$. For each such $i$, we check if $H_{P,T^{(i)}} \equiv 0$ (as a symbolic polynomial). For each specific location $i$, this can be performed in time $O(m)$. Once the first location for which $H_{P,T^{(i)}} \equiv 0$ is found, we conclude that the distance is 1, and no further locations are checked.

It remains to explain how to compute $H_{P,T^{(i)}}$, for all $i$. We do so using convolutions. Specifically, from the string $P$, we create a string $P'$ of length $2m$, by replacing each entry $a$, by the pair $r(a)^2, r(a)$ (where $r(a)$ is the value given to the symbolic variable $a$ under the random assignment $r$). Similarly, from $T$ we create a string $T'$ of length $2n$, by replacing each $b$ with the pair $-r(b), r(b)^2$. Then, if $C$ is the convolution of $T'$ and $P'$, then for all $i$, $C(2i) = H_{P,T^{(i)}}(r)$. We obtain:

**Lemma 7.** [2] *The above algorithm determines if there is a parallel interchange of $P$ in $T$ in expected time $O(n \log m)$.*

*Example:* Consider the text $T = abcbaabbc$ and pattern $P = ccaababbb$, and suppose we assign $a = 1$, $b = 2$, $c = 3$. Then,

$$T' = 1, -1, 4, -2, 9, -3, 4, -2, 1, -1, 1, -1, 4, -2, 4, -2, 9, -3$$
$$P' = 3, 9, 3, 9, 1, 1, 1, 1, 2, 4, 1, 1, 2, 4, 2, 4, 2, 4$$

The convolution of $T'$ and $P'$ gives the sum of the following differences:

$$3 - 9, 12 - 18, 9 - 3, 4 - 2, 2 - 4, 1 - 1, 8 - 8, 8 - 8, 18 - 12.$$

Note that match positions contribute 0 to the convolution sum and positions of parallel interchanges cancel themselves. We thus have a randomized algorithm that computes the parallel interchange distance in expected $O(n \log m)$ steps.

We may now be tempted to try and extend this method to obtain a more efficient deterministic algorithm, in the following method. Suppose that we could find a small number of polynomials, $H^{(1)}, H^{(2)}, \ldots, H^{(k)}$, such that for a *given* assignment, computing their values at each text location $i$ would provide a deterministic indication of a parallel interchange. For example, suppose we could find a "good" set of polynomials such that for any assignment they vanish iff there is a parallel interchange. Then, if we could compute their values using convolutions, we could hope for an efficient algorithm. The next lemma, which is based on communication complexity arguments, proves that such an approach cannot provide better performance than $\widetilde{\Omega}(nm)$. To this end we use the *convolution model*, which is a specialized model of computation that solves a subset of pattern matching problems, defined in [4] as follows:

**Definition 7.** *Given a pattern matching problem whose input is a text $T$ and a pattern $P$, a solution in the* convolutions model *has the following form. Let $g_i$, $i = 1, \ldots, h(n)$ be pattern preprocessing functions, and let $f_{g_i}$, $i = 1, \ldots, h(n)$ be the corresponding local text preprocessing functions. Let $b$ be a parameter for size in bits.*

1. *Compute $h(n)$ convolutions $C_i \leftarrow f_{g_i}(T) \otimes g_i(P)$, $i = 1, \ldots, h(n)$, with $b$-bit inputs and outputs.*
2. *Compute the matches as follows. The decision of whether location $j$ of the text is a match is made by a computation whose inputs are a subset of $\{C_i[j] | i = 1, \ldots, h(n)\}$.*

**Lemma 8.** [3] *Any algorithm in the convolution model for determining if there is a parallel interchange requires $(m(n - m + 1))$ bit operations.*

### 3.2 The Length Cost Model

The popular cost model used in the study of rearrangement distances is the unit cost model. In this model, every rearrangement operation is given a unit cost. This is the model used in the definition of the interchange distance problem [2]. Recently, Bender et al. [16] initiated the study of length-weighted cost models for rearrangement operators. Their basic claim is that, there is no real reason to assume that all operations always have equal cost. On the contrary, in some situations rearranging closer elements may be cheaper than rearranging distant elements. Bender et al. also give biological justification for length-weighted cost models of the reversal rearrangement operator. Following their basic observation, [11,12] studied the interchange rearrangement under a variety of length cost models.

A cost function of an interchange of elements in positions $i, j$ where $i < j$, can be viewed as a cost function on the segment $[i, j]$. Following Bender et al., this paper considers increasing monotone cost functions on the *length* of the segment, i.e. $|i - j|$. Such cost functions seem more natural in situations where the distance that objects are moved contributes to the cost. Cost functions $w$ of the form $w(\ell) = \ell^\alpha$ for all $\alpha \geq 0$, where $\ell = |i - j|$, are specifically studied. These cost functions are referred to as $\ell_\alpha$-cost functions. The study is also broadened to include various cost functions (e.g. $\log(\ell)$) classified by their characteristic behavior with regards to the marginal cost. Two types of cost functions are considered: the $\mathcal{I}$*-type* and the $\mathcal{D}$*-type* defined as follows.

**Definition 8.** *Let $w : \mathbb{N} \mapsto \mathbb{R}$ be a cost function. We say that:*
• *$w \in \mathcal{I}$-type if for every $a, b, c \in \mathbb{N}$ such that $a < b$, $w(a + c) - w(a) < w(b + c) - w(b)$. We call this property* ***the law of increasing marginal cost****.*
• *$w \in \mathcal{D}$-type if for every $a, b, c \in \mathbb{N}$ such that $a < b$, $w(a + c) - w(a) > w(b + c) - w(b)$. We call this property* ***the law of decreasing marginal cost****.*

Though this study of length-weighted cost models may include many cost functions that are hardly meaningful in practical situations, we prefer the generalized perspective. A comparison with the results of Bender et al. [16], who studied the

**Table 2.** $\ell_\alpha$-interchange Distance Problem: A Summary of Results

| $\alpha$ Value | Binary Alphabet | Permutations | General Strings |
|---|---|---|---|
| $\alpha = 0$ | $O(m)$ | $O(m)$ | $\mathcal{NP}$-hard<br>$O(m \log \lvert\Sigma\rvert)$ 1.5-approximation |
| $0 < \alpha \leq \frac{1}{\log m}$ | $O(m)$ $^\star$ | $O(m)$ 2-approximation | $O(m \log \lvert\Sigma\rvert)$ 3-approximation |
| $\frac{1}{\log m} < \alpha < 1$ | $O(m)$ $^\star$ | $O(m)$ 2-approximation | $O(m^3)$ $\lvert\Sigma\rvert$-approximation |
| $\alpha = 1$ | $O(m)$ | $O(m)$ | $O(m)$ |
| $1 < \alpha \leq \log 3$ | $O(m)$ | $O(m)$ 2-approximation | $O(m)$ 2-approximation |
| $\alpha > \log 3$ | $O(m)$ | $O(m)$ | $O(m)$ |

$\star$ Only for the sorting problem. For the general rearrangement problem it is $O(m^3)$.

reversal rearrangement operator on permutation strings in the $\ell_\alpha$-cost models for different values of $\alpha$, supports our approach. They showed that the sorting by reversals problem, which is known to be $\mathcal{NP}$-hard even on permutations, is polynomial time computable for some length-weighted cost models. The results of [11] and [16] together might indicate a general phenomenon about length-weighted distances that should be further studied. Moreover, [11] proposed the classification of cost functions by *laws of increasing/decreasing marginal cost*. This classification gives insight to the behavior of the different cost functions and enables deriving results for cost functions that were not yet sufficiently analyzed by Bender et al. [16].

In Subsubsection 3.2, the $\ell_1$-cost model is studied. A characterization of the distance is given, and it is proven to be polynomial time computable. In [11], the problems for $\mathcal{I}$-type and $\mathcal{D}$-type cost functions are studied, and optimal and approximation algorithms for the problem under these cost models are given. The results apply specifically to $\ell_\alpha$-interchange distance problem for every $\alpha > 1$ and $0 < \alpha < 1$, but apply to other functions as well (e.g. $\log(\ell)$). A summary of the results of [11] for the $\ell_\alpha$-interchange distance problem is given in Table 2. The table presents running times for finding the distance of $m$-length strings.

**The $\ell_1$-Cost Model.** We now describe a characterization of the $\ell_1$-interchange distance (denoted by $\mathrm{WI}(\ell_1)$) and a polynomial time algorithm for the interchange distance problem in the $\ell_1$-cost model. We deal with permutations (strings with distinct elements) first, and then show how to handle general strings.

*Permutation Strings.* Let $x, y$ be two strings with the same $m$ distinct symbols. Since the rearrangement of $x$ to $y$ can be viewed as sorting $x$ by assuming $y$ is the permutation $1, \ldots, m$, while making the appropriate changes for symbols names in $x$ as explained in Subsection 3.1, in the sequel we assume that we sort $x$ to the identity permutation $I = 1 \ldots m$. Let $\pi$ be a permutation of $1, \ldots, m$. For an element $i$ in $\pi$, define its *critical* segment to be $[\pi(i), i]$ if $\pi(i) < i$ or $[i, \pi(i)]$ if $i < \pi(i)$. Lemma 9 is a first step in the characterization of the $\ell_1$-interchange distance for permutations. We will show the connection to the $\ell_1$-distance, defined in Section 2.

```
SORT(π)
Begin
    While there are unsorted pairs in π
        Find a good pair i,j.
        Interchange elements i and j.
End
```

**Fig. 3.** Algorithm Sort

**Lemma 9.** *Let $\pi$ be a permutation of $1,\ldots,m$, and let $I$ denote the identity permutation. Then, $d_{WI(\ell_1)}(\pi, I) \geq \frac{d_{\ell_1}(\pi, I)}{2}$.*

*Proof.* In order to be sorted, every element $i$ must pass its critical segment. Note that in the $\ell_1$-cost model, the distance is not diminished if element $i$ passes the critical segment using more than one interchange in this segment. Thus, for every element we must pay its critical segment in the $\ell_1$-cost model. Therefore, the best situation is where each interchange we perform sorts the two participating elements, since in this case we pay for each interchange exactly the cost that every element must pay. In this case, each interchange costs half the cost paid in the $\ell_1$ distance, since in the $\ell_1$ distance each element is charged independently. The lemma follows. □

Given a permutation $\pi$ of $1,\ldots,m$, we provide a polynomial time algorithm that sorts the permutation by interchanges (see Fig. 3). Our algorithm performs only interchanges that advance both elements towards their final positions. The following definition is a formalization of this requirement.

**Definition 9.** *Let $\pi$ be a permutation of $1,\ldots,m$. A pair of elements $i,j$ is a* good pair *if $j \leq \pi(i) < \pi(j) \leq i$.*

The next lemma states that it is possible to sort a permutation using only good pairs.

**Lemma 10.** [11] *Every non-identity permutation has a good pair.*

Note that by interchanging good pairs, elements move along their *critical* segment only. Since the cost paid for this movement is its total length, and every interchange cost can be divided between the two participating elements, the sum of costs of good-pairs interchanges never exceeds $d_{\ell_1}(\pi, I)/2$, Lemma 11 follows.

**Lemma 11.** *Let $\pi$ be a permutation of $1,\ldots,m$. Then, $d_{WI(\ell_1)}(\pi, I) = \frac{d_{\ell_1}(\pi, I)}{2}$.*

*Remark.* Algorithm Sort requires $O(m^2)$ time. However, the WI($\ell_1$)-distance can be computed in $O(m)$ time, if an actual rearrangement sequence need not be produced.

*General Strings.* The main difficulty in the case of general strings is that repeated symbols have multiple choices for their desired destination. Let $x$ and $y$ be strings of length $m$. Our goal is to pair the locations in $x$ to destination locations in $y$, so that repeating symbols can be labeled in $x$ and $y$ to get strings with the same $m$ distinct symbols (permutation strings). Such a labeling can be viewed as a permutation of the indices of $x$. Clearly, if $x$ contains distinct elements then only one labeling permutation can convert $x$ to $y$. However, there can be many labeling permutations if $x$ contains multiple occurrences of elements. Trying all labeling permutations $\pi$ and choosing the one that gives the minimal $w$-interchange distance is impractical. Fortunately, Lemma 1 characterize a labeling permutation of indices that gives the minimum $\ell_1$-distance. This will be enough to derive a polynomial time algorithm for the $\ell_1$-interchange distance problem in the general strings case as well.

**Theorem 11.** *Let $x$ and $y$ be $m$-length strings. Then, $d_{WI(\ell_1)}(x,y) = \frac{d_{\ell_1}(x,y)}{2}$. Moreover, $d_{WI(\ell_1)}(x,y)$ is computable in $O(m)$ time and the actual rearrangement sequence is computable in $O(m^2)$ time.*

*Proof.* An algorithm for finding an actual rearrangement sequence is the following: transform $x$ and $y$ into permutation strings $x'$ and $y'$ by giving different labels to repeating symbols in $x$ and give the labels in $y$ according to $\pi_o$, i.e. $y'[\pi_o(i)] = x'[i]$. This transformation can be done in time $O(m)$. Now, run the Sort algorithm (see Fig. 3) for permutation strings on $x'$ and $y'$, and return its result. The cost of the optimal algorithm can be viewed as the minimum over all labelling permutations (pairing) $\pi$ of the distances returned by the optimal algorithm on the permutation strings $x_\pi$ and $y_\pi$ resulting from a labelling according to $\pi$. By Lemma 11, for every $\pi$, $d_{WI(\ell_1)}(x_\pi, y_\pi) = \frac{d_{\ell_1}(x_\pi, y_\pi)}{2}$. Finally, by Lemma 1, $d_{\ell_1}(x', y')$ is minimal, so our algorithm provides the $\ell_1$-interchange distance of $x$ and $y$. The overall time for computing the rearrangement sequence is determined by the time of the Sort algorithm, which is $O(m^2)$. Computing $d_{WI(\ell_1)}(x,y)$ requires only the computation of $d_{\ell_1}(x', y')$, which can be done in $O(m)$ time. □

### 3.3 The Element Cost Model

[30] is the first paper that explicitly considered the *ECM* for dealing with rearrangement problems. A formal definition of this cost model is given below.

**Definition 10.** *Let $w : \Sigma \to \mathbb{R}^+$ be a weight function, which assigns a non-negative weight to every element in $\Sigma$. Let $g : \Sigma \times \Sigma \to \mathbb{R}^+$ be a function defining the interchange cost. The function $g$ is called a* general function *if it satisfies the following conditions:*

1. $\forall x, y \in \Sigma : g(x,y) = g(y,x)$.
2. $\forall x, y, z \in \Sigma : w(y) \leq w(z) \Leftrightarrow g(x,y) \leq g(x,z)$.

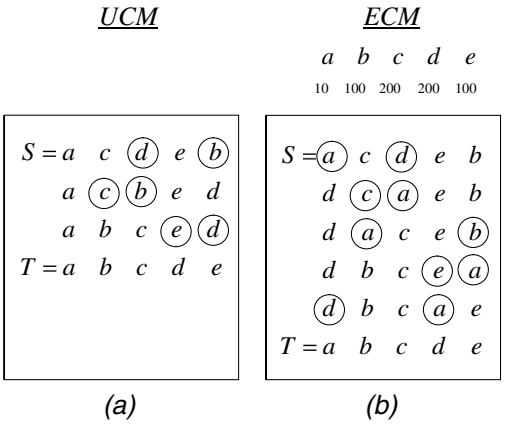



**Fig. 4.** In both (a) and (b), every row represents a stage in the rearrangement. The elements marked with circles are the elements interchanged to establish the next stage. In (a), the goal is to transform $S$ into $T$ with a minimum number of interchanges ($UCM$). This is done by applying 3 interchanges. In (b), the $ECM$ is used. Every element has a weight and the cost of an interchange is the sum of the weights. The sequence of interchanges applied in (a) costs 900, whereas the sequence of 5 interchanges applied in (b) costs 850.

The summation function $g(x, y) = w(x) + w(y)$ and the multiplication function $g(x, y) = w(x) \cdot w(y)$ are two examples of intuitive general functions.

The technique used in the *interchange rearrangement problem* under the $ECM$ is different than the one used under the $UCM$. Consider the example shown in Fig. 4. In this example, an optimal rearrangement is given when the $UCM$ is used - an input string $S$ is transformed into a same length target string $T$ over the same alphabet $\Sigma$, using 3 interchanges (Fig. 4(a)). When the $ECM$ is used, the same sequence of interchanges costs 900, whereas the alternative sequence of interchanges suggested performs 5 interchanges and costs only 850 (Fig. 4(b)).

The main results presented in [30] are:

1. $O(m)$ time algorithm for the *interchange rearrangement problem* for *permutation strings* for any general function.
2. $\mathcal{NP}$-hardness for the *interchange rearrangement problem* for *general strings*:
   (a) $O(m)$ time 3-approximation algorithm for any general function.
   (b) $O(m \cdot \lg |\Sigma|)$ time 1.72-approximation algorithm for the summation function.

For the interchange distance problem under the $UCM$, Cayley [20] showed that given a permutation $\pi$, the minimum number of interchanges needed for sorting $\pi$, is $m - c(\pi)$. This is achieved by interchanging only elements that share a cycle until there are no such elements (the permutation is sorted). When the $ECM$ is used, one might also be inclined to apply a minimum number of interchanges. This inclination implies that one would be making interchanges only within cycles. Any interchange between elements of different cycles would result in an

increase in the number of interchanges needed for sorting $\pi$ and probably in the total cost for sorting $\pi$. However, this inclination is incorrect. Moreover, there might be cases in which the optimal solution would be to increase the number of interchanges needed for sorting $\pi$ in order to decrease the total cost.

[30] describe an algorithm for sorting a *permutation string* by *interchanges* under *ECM*, and prove that it yields the optimal cost. The basic idea of this algorithm is quite simple. In order to sort the permutation $\pi$ at a minimum cost, either the cheapest element in some cycle is used to sort all the other elements including itself, or (if the cheapest element in the cycle is not cheap enough) the cost for introducing the cycle to the cheapest element in $\pi$ is "paid" by interchanging it with the cheapest element of the cycle. Doing so unites the cycle with the cycle of the minimum cost element of $\pi$. Then the cheapest element of $\pi$ can be used to sort all the other elements in the cycle. We call this algorithm "*The Cheapest Employee Algorithm" (CEA)*. This optimal algorithm for permutation strings is then combined with suitable cycle decomposition heuristics on the graph constructions in order to achieve the approximation results.

[31] broaden the study to include the *single elements transposition rearrangement problem* (denoted se-transposition) under the *ECM*, *UCM* and the *LCM* for *general strings* and *permutation strings*.

### 3.4 External Process Rearrangements: Summary

Table 3 summarizes the results on external process rearrangement systems as studied in [2,11,30,31].

**Table 3.** External Process Rearrangement Systems: A Summary of Results

| | UCM | ECM | LCM $\star$ |
|---|---|---|---|
| **Interchanges** | | | |
| *Permutation Strings* | $O(m)$ [20] | $O(m)$ (general function) [30] | $O(m)$ [11] |
| *General Strings* | $\mathcal{NP}$-hard [11] $O(m \cdot \lg \lvert\Sigma\rvert)$ 1.5-approx. [11] | $\mathcal{NP}$-hard $O(m)$ 3-approx. (general function) [30] $O(m \cdot \lg \lvert\Sigma\rvert)$ 1.72-approx. (summation function) [30] | $O(m)$ [11] |
| **P-Interchanges** | | | |
| *Permutation Strings* | $O(m)$ [2] $\star\star$ | – | – |
| *General Strings* | $O(\lvert\Sigma\rvert^2 n \log m)$ [2] $\star\star\star$ | – | – |
| **SE-Transpositions** | | | |
| *Permutation Strings* | $O(m \lg m)$ [26] | $O(m \lg m)$ [31] | $O(m \lg m)$ [31] |
| *General Strings* | $O(m^2)$ [31] | $O(m^2)$ [31] | $O(m \lg m)$ [31] |

$\star$ The results for LCM presented in this table refer only to the $\ell_1$ cost model of the given operators.
$\star\star$ Implicit from theorems proven in [2].
$\star\star\star$ For the string matching problem. The randomized algorithm has expected $O(n \log m)$ time.

The results for LCM presented in this table refer only to the $\ell_1$cost model of the given operators. This is the simplest of the LCMs. For a detailed examination of other LCMs for the interchange operator see Table 2. The behavior of the other operators under other LCMs was not studied and is open for future research. It is important to note that even for the interchange operator that was extensively studied under a variety of LCMs, the true complexity of some classes of LCMs is still open. This can be seen from Table 2, as we give approximation algorithms in cases that we have neither a polynomial time optimal algorithm nor an $\mathcal{NP}$-hardness proof. Note that the parallel-interchange operator was studied only for UCM and its behavior under LCM or ECM is still an open question.

Another important note is that, except for the parallel-interchanges operator, the string matching problem was generally not studied. The current state of the art is basically to use algorithms for equal length text and pattern at each text position. How to use information gathered in checked positions to accelerate the string matching is still open for almost all the studied rearrangement operators.

## 4 Internal Process Rearrangement Systems

Another broad class of rearrangement systems inspired by computer architecture was studied by [7]. They consider address errors which arise from a process of flipping some or all of the bits in the binary representation of $[1, m]$. Such errors represent situations where the text and the pattern are generated by two different systems, which may use different naming conventions. The error processes are inspired by address errors resulting from failures in the wires of the address bus, the wires connecting the CPU and the memory which are used to transmit the address of operands (see Fig. 5), or failure in the transmitted address bits.

### 4.1 The Bit Errors Definition

Consider a string $S \in \Sigma^m$. Using an alternative view of strings we write $S = \{(\sigma, i) : i \in \{0,1\}^{\log m}\}$. Four types of errors in the bits of the $i$ entries were considered in [7,10]:

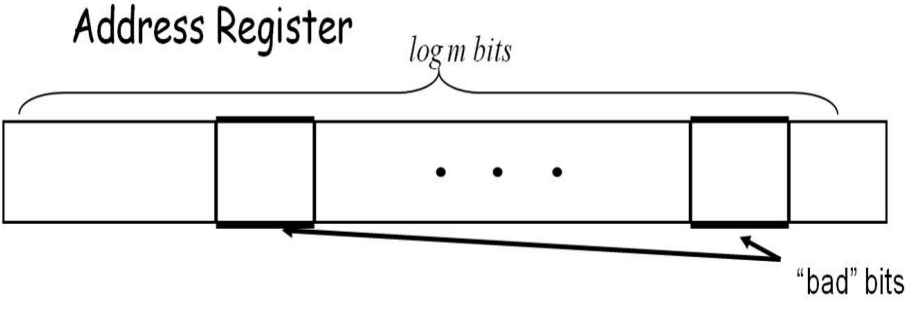



**Fig. 5.** Failures in the address bus due to 'bad' bits cause wrong addresses to be stored in the address register

**Flipped bits:** There exists a subset of bit positions $F \subseteq \{0, \ldots, \log m - 1\}$, such that in each $i$, all bits in positions $f \in F$ are flipped (i.e. 1 is turned into a 0 and visa versa).

For example, for the string $S = 1234 = \{(1, 00), (2, 01), (3, 10), (4, 11)\}$ and $F = \{1\}$, the resulting string is $S' = 3412 = \{(1, 10), (2, 11), (3, 00), (4, 01)\}$.

**Faulty bits:** There exists a subset of bit positions $F \subseteq \{0, \ldots, \log m - 1\}$, such that in each $i$, the bits in positions $f \in F$ *may* be flipped, and may not.

For example, for the string $S = 1234 = \{(1, 00), (2, 01), (3, 10), (4, 11)\}$ and $F = \{1\}$, the resulting string may be $S' = \{(1, 10), (2, 01), (3, 10), (4, 01)\}$ (the bit was flipped for 1 and 4 but not for 2 and 3).

Note that in this case the resulting set is actually a multi-set, and may not represent a valid string, as some locations may appear multiple times, while others not at all.

**Stuck bits:** There exists a subset of bit positions $F \subseteq \{0, \ldots, \log m - 1\}$, such that in each $i$, all bits in positions $f \in F$ are either always changed to zero (i.e. 1 is turned into a 0 and 0 remains 0) or always changed to one (i.e. 0 is turned into a 1 and 1 remains 1).

For example, for the string $S = 1234 = \{(1, 00), (2, 01), (3, 10), (4, 11)\}$ and $F = \{1\}$, a resulting string is $S' = \{(1, 00), (2, 01), (3, 00), (4, 01)\}$.

**Transient stuck bits:** There exists a subset of bit positions $F \subseteq \{0, \ldots, \log m - 1\}$, such that in each $i$, the bits in positions $f \in F$ *may* remain unchanged, or may be changed to a "1" (of course the original string changes only if the intention was to output a "0").

As an example, for the string $S = 1234 = \{(1, 00), (2, 01), (3, 10), (4, 11)\}$ and $F = \{1\}$, the resulting string may be $S' = \{(1, 10), (2, 01), (3, 10), (4, 11)\}$ (the bit was changed to one for address 1 but not for address 2).

[7,10] consider approximate pattern matching problems associated with each of the above types of errors. Specifically, given a pattern $P$ and text $T$, the goal is to find:

- the smallest set $F$ such that if the bits of $F$ are consistently flipped, then $P$ has a match in $T$. We call this problem the *flipped bits* problem.
- the smallest set $F$ such that if the bits of $F$ may be transiently flipped, then $P$ has a match in $T$. We call this problem the *faulty bits* problem.
- the smallest set $F$ such that if the bits of $F$ are consistently stuck, then $P$ has a match in $T$. We call this problem the *stuck bits* problem.
- the smallest set $F$ such that if the bits of $F$ may be transiently stuck, then $P$ has a match in $T$. We call this problem the *transient stuck bits* problem.

In [7] the following results were proved:

- For pattern and text of size $m$, the flipped bits problem can be solved in $O(m \log m)$ steps.
- For pattern and text of size $m$, the faulty bits problem can be solved deterministically in $O(m^{\log_2 3}|\Sigma|)$ steps and randomly in $O(m \log m)$ steps.
- For pattern and text of size $m$, the faulty bits problem can be deterministically approximated to a constant $c > 1$ in $O(|\Sigma| \frac{m^{\log 3}}{\log^{c-1} m})$.

– For text and pattern of sizes $n$ and $m$, respectively, $m$ power of 2, the faulty bits problem can be solved deterministically in $O(|\Sigma|nm\log m)$ steps.

In addition, [10] show:

– An $O(m \log m)$ time algorithm for the stuck bits problem, which also reports the stuck bits positions.
– A simple $O(m^{2.5})$ time algorithm for the transient stuck bits problem, which also reports the stuck bits positions. This algorithm is based on a reduction to finding perfect matching in a bipartite graph.
– A flow-based $O(m^{2.2156} \log^2 m)$ time algorithm for the transient stuck bits problem, which also reports the stuck bits positions.

In the rest of the section we will describe the solutions for the flipped bits and faulty bits problems as appear in [8].

### 4.2 Flipped Bits Errors

In this section we consider the flipped bits problem. In this setting, one or more of the bit positions may exhibit a faulty behavior whereby the bit at this position is consistently flipped. Given two strings $P, T \in \Sigma^m$, the distance between the two is the least number of *flipped bits* positions that can explain the differences between the two, and $\infty$ if no such set of position can explain the difference. Formally,

**Definition 11.** *For an index $k \in [0..m-1]$,[6] we view $k$ as a binary string, i.e. $k = k[0] \cdots k[\log m - 1] \in \{0,1\}^{\log m}$ (w.l.o.g. $m$ is a power of 2). Consider $F \subseteq [0..\log m - 1]$. The* bit flip transformation induced by $F$*, denoted $f_F$, is a function $f_F : \{0,1\}^{\log m} \to \{0,1\}^{\log m}$, such that for any $k$ and $i$*

$$f_F(k)[i] = \begin{cases} 1 - k[i] & i \in F \\ k[i] & i \notin F \end{cases}$$

*i.e. the value of $f_F(k)$ is flipped at bits of $F$ and identical on other bits.*

*For strings $P, T \in \Sigma^m$ we say that $T$ is a $F$-*flip-bits match of $P$ *if for all $k \in \{0,1\}^{\log m}$, $T[k] = P[f_F(k)]$. The* flip-bit distance between $P$ and $T$ *is the cardinality of the smallest $F$ such that $T$ is an $F$-flip-bits match of $P$. If no such $F$ exists, then the distance is $\infty$.*

Note that there are $2^{\log m}$ possible faulty sets $F$. Checking each possibility separately takes $O(m)$, so a naive algorithm takes time $O(m^2)$ per position. We show how to reduce this to $O(m \log m)$. We begin with an efficient solution for the case $\Sigma = \{0,1\}$, and then use it to obtain an efficient solution for general alphabets.

Let $k, j \in \{0,1\}^{\log m}$, denote $k \oplus j$ to be the result of the bitwise XOR of the two, i.e. for each $i$, $(k \oplus j)[i] = k[i] \oplus j[i]$ (where $\oplus$ is the XOR operation, i.e.

[6] For integers $i$, $j$, we denote by $[i..j]$ the set of integers from $i$ to $j$. Thus, $[0..m-1]$ is the set $\{0, 1, \ldots, m-1\}$.

addition over $Z_2$). For strings $T, P \in \mathbb{Z}^m$, define the *binary convolution* of the two to be a vector, also of size $m$, $T \otimes P \in \mathbb{Z}^m$, such that for all $k \in \{0,1\}^{\log m}$: $(T \otimes P)[k] = \sum_{j \in \{0,1\}^{\log m}} T[j] \cdot P[k \oplus j]$.

**Lemma 12.** *For a set $F \subseteq 0..\log m - 1$, let $\chi_F \in \{0,1\}^{\log m}$ be the characteristic vector of $F$. Consider binary $P$ and $T$, both of size $m$, and let $\alpha_P$ be the number of ones in $P$ and $\alpha_T$ be the number of ones in $T$. Then, $T$ is an $F$-flip-bits match of $P$ iff $\alpha_T = \alpha_P$ and $(T \otimes P)[\chi_F] = \alpha_T$.*

*Proof.* For any index $j$, $T[j] \cdot P[\chi_F \oplus j] = 1$ iff both $T[j] = 1$ and $P[\chi_F \oplus j] = P[f_F(j)] = 1$. Thus, $(T \otimes P)[\chi_F]$ counts the number of ones in $T$ that are mapped to ones in $P$ under the transformation $f_F$. Since, $(T \otimes P)[\chi_F] = \alpha_T$, then all ones in $T$ are mapped to ones in $P$. But, $\alpha_T = \alpha_P$, so also all zeros in $T$ are mapped to zeros in $P$. □

Thus, in order to find the *flip-bit distance between $P$ and $T$* we compute the entire vectors $T \otimes P$. We then seek all locations $k$ for which $(T \otimes P)[k] = \alpha_T$, and among these $k$'s, find the one with the minimum weight (i.e. least number of 1's).

It thus remains to explain how to efficiently compute the binary convolution. The convolution can easily be computed in $O(m^2)$ time. We explain how to compute it in $O(m \log m)$ time.

For a vector $v \in \mathbb{Z}^t$ ($t$ power of 2), define two vectors $v^+, v^- \in \mathbb{Z}^{t/2}$, as follows. For each $k \in \{0,1\}^{\log t - 1}$, $v^+[k] = v[0k] + v[1k]$ and $v^-[k] = v[0k] - v[1k]$. The key lemma for the computation is:

**Lemma 13.** [7] *For any $v, w \in \{0,1\}^t$, and $k \in \{0,1\}^{\log t - 1}$:*
$(v \otimes w)[0k] = \frac{(v^+ \otimes w^+)[k] + (v^- \otimes w^-)[k]}{2}$, $(v \otimes w)[1k] = \frac{(v^+ \otimes w^+)[k] - (v^- \otimes w^-)[k]}{2}$.

Thus, in order to compute $T \otimes P$, our algorithm recursively computes $T^+ \otimes P^+$ and $T^- \otimes P^-$, and then uses Lemma 13 in order to compute the convolution $T \otimes P$. In each recursion level we need to compute $O(m)$ values, each taking $O(1)$ time. Thus, we get a recursive recurrence $time(m) = 2 \cdot time(m/2) + cm$, for a total $time(m) = O(m \log m)$. We obtain:

**Theorem 12.** *The flipped bit problem can be solved in $O(m \log m)$ time for binary text and pattern of size $m$.*

For a general alphabet, the same techniques as in [23] can be used to handle with only one convolution. Hence,

**Theorem 13.** *The flipped bit problem can be solved in $O(m \log m)$ time for text and pattern of size $m$ and alphabet $\Sigma$.*

*Remark.* The above algorithm can also be viewed as a form of Fast Fourier Transform over $\mathbb{Z}_2$ (rather than over the complexes). We omit the details.

### 4.3 The Faulty Bits Problem

This section studies the faulty bits problem. In this model a faulty position inconsistently produces errors. It may sometimes hold the correct value and sometimes the wrong one. Given two strings, the objective is to find the least number of faulty positions that explain the differences between the two. We begin by formally defining the faulty bits distance problem.

**Problem Definition.** Let $\Sigma$ be a finite alphabet. Let $P, T \in \Sigma^m$ be two strings of length $m$, such that $P$ is the *query string* and $T$ is the *stored string*. Denote $P = p[0]p[1]\cdots p[m-1]$ and similarly for $T$. Consider $F \subseteq \{0, \ldots, \log m - 1\}$, and suppose that the address bits carrying bits in the set $F$ are faulty. We now formulate the criterion that determines if the stored string $T$ matches the query string $P$, assuming that the bits of $F$ are faulty.

Consider an address $k$, and let $k = k[0]k[1]\cdots k[\log m - 1]$ be the binary representation of $k$. Let $[k]_F$ be the set of all the addresses $\ell$ such $k[i] = \ell[i]$ for all $i \notin F$, i.e. $k$ and $\ell$ agree on all bits not in $F$. Note that $[k]_F$ is an equivalence class, so $[\ell]_F = [k]_F$ if $\ell \in [k]_F$. Then, if the address bits in $F$ are faulty, a value intended to location $k$ can end up in any location $\ell \in [k]_F$. Thus, we obtain the following criterion for a match of $T$ to the query string $P$ while using the faulty bits of $F$:

**Definition 12.** *For strings $P$ and $T$ and set $F \subseteq \{0, \ldots, \log m - 1\}$ we say that $T$ is an* $F$-faulty-bit match *of $P$ if for each equivalence class $[k]_F$ and for each $\sigma \in \Sigma$*

$$|\{\ell : \ell \in [k]_F, P[\ell] = \sigma\}| = |\{\ell : \ell \in [k]_F, T[\ell] = \sigma\}|$$

*The Optimization Problem.* Given any of the above match conditions and strings $P$ and $T$, we wish to find the set $F$ of minimal cardinality such that $T$ is an $F$-faulty-bit match of $P$. We call this the *faulty-bits problem.*

**A Deterministic Algorithm.** For each equivalence class $[k]_F$ and $\sigma \in \Sigma$ let

$$\textsc{bucket}(P, [k]_F, \sigma) = \{\ell : \ell \in [k]_F, P[k] = \sigma\}$$

the elements of $P$ with locations in $[k]_F$ that have value $\sigma$. Similarly,

$$\textsc{bucket}(T, [k]_F, \sigma) = \{\ell : \ell \in [k]_F, T[\ell] = \sigma\}$$

the elements of $T$ with locations in $[k]_F$ that have value $\sigma$. The criteria for an $F$-faulty-bit match is that for all $k$ :

$$|\textsc{bucket}(P, [k]_F, \sigma)| = |\textsc{bucket}(T, [k]_F, \sigma)|$$

for all $\sigma$. Thus, it remains to explain how to compute the sizes of the buckets.

For any fixed $F$, all buckets can be computed in a total of $O(m)$ steps, with a single pass over the strings $T$ and $P$. Thus, for a given $F$, the condition can be tested in $O(m)$ steps. There are $2^{\log m} = m$ different possible sets $F$,

which provides a naive $O(m^2)$ algorithm. We now show how to reduce this to $O(m^{\log 3}|\Sigma|)$.

For an address $k$ and index $i \in \{0, \ldots, \log m - 1\}$, let $k^{(i)}$ be the address which has the same representation as $k$ except for the $i$-th bit which is flipped. Then, it is easy to see that for any $i$, $\sigma$ and $X \in \{T, P\}$, $\text{BUCKET}(X, [k]_F, \sigma) = \text{BUCKET}(X, [k]_{F-\{i\}}, \sigma) \cup \text{BUCKET}(X, [k^{(i)}]_{F-\{i\}}, \sigma)$. That is, the bucket with faults at $F$ can be obtained as the union of buckets with one less fault, and fixing the two possible values for this bits. In particular,

$$|\text{BUCKET}(X, [k]_F, \sigma)| = |\text{BUCKET}(X, [k]_{F-\{i\}}, \sigma)| + |\text{BUCKET}(X, [k^{(i)}]_{F-\{i\}}, \sigma)| \tag{6}$$

Note that for $F = \emptyset$,

$$|\text{BUCKET}(X, [k]_F, \sigma)| = \begin{cases} 1 & X[k] = \sigma \\ 0 & X[k] \neq \sigma \end{cases} \tag{7}$$

Thus, combining (7) and (6), we obtain that for any $\sigma$ all sizes of all buckets can be computed in an inductive fashion, with $O(1)$ steps per bucket.

For a given $\sigma$, the overall total number of buckets – for all fault patterns $F$, is the overall total number of equivalence classes $[k]_F$ for all $F$. Each equivalence class can be identified with a string $w \in \{0, 1, *\}^{\log m}$ such that $w[i] = *$ denotes a bit in $F$ and the other $w[i]$'s are fixed as in $k$. Thus, the number of equivalence classes is: $|\{w \in \{0, 1, *\}^{\log m}\}| = 3^{\log m} = m^{\log 3}$. We thus obtain:

**Theorem 14.** *The faulty-bits problem can be solved in $O(|\Sigma| m^{\log 3})$ time.*

**Other Results on the Faulty Bits Problem.** In order to achieve faster algorithms for the faulty bits problem two alternative methods were used in [7]. The first alternative is to settle for an inexact computation of the buckets counters enabling to compute buckets together. In this direction there was a use of formal polynomials and the Schwartz-Zippel Lemma [35,37]. The result is a much faster randomized algorithm. The other alternative is to compute buckets counters exactly but to avoid the computation of all the buckets. This direction is based on the observation that the containment structure of the buckets enables to *approximate* the bucket of the minimum size from a special designed scheme of sparse subset of buckets to be computed. [7] prove that a deterministic such scheme exists and can be explicitly constructed. Another variant of the problem that was considered is the pattern matching version, i.e., where the text is longer than the pattern. The way to improve over the naive use of the algorithm for strings with the same length on each text location is to use special new variations of the KMR algorithm [32]. However, this method only works for pattern of length a power of 2. Standard techniques that employ the KMR algorithm for patterns of unlimited size do not apply here. This is another example of the need for different and special techniques adequate for handling rearrangements.

### 4.4 Internal Process Rearrangements: Summary

Table 4 summarizes the results on internal process rearrangement systems as studied in [7,8,10]. Note that in internal process rearrangement systems, as we

**Table 4.** Internal Process Rearrangement Systems: A Summary of Results

| The Problem | Equal Length strings | String Matching |
|---|---|---|
| **Flipped Bits** | $O(m \log m)$ [7] | $O(nm)$ $\star$ |
| **Faulty Bits** | $O(m^{\log_2 3}\lvert\Sigma\rvert)$ [7] $\star\star$ | $O(nm^{\log_2 3}\lvert\Sigma\rvert)$ $\star\star\star$ |
| **Stuck Bits** | $O(m \log m)$ [10] | $O(nm \log m)$ $\star\star\star$ |
| **Transient Stuck Bits** | $O(m^{2.2156} \log^2 m)$ [10] | $O(nm^{2.2156} \log^2 m)$ $\star\star\star$ |

$\star$ Implicit from [7] via $m$ activations of Knuth-Morris-Prat algorithm [33].
$\star\star$ The randomized algorithm has $O(m \log m)$ time and returns the correct answer with high probability.
$\star\star\star$ Via a naive use of the algorithms for text and pattern of equal size on each text position.

saw for external process rearrangement systems, the current state of the art in almost all operators does not offer good solutions for the string matching problem. This is a called for challenge for future research.

## References


1. Amir, A.: Asynchronous pattern matching. In: Lewenstein, M., Valiente, G. (eds.) CPM 2006. LNCS, vol. 4009, pp. 1–10. Springer, Heidelberg (2006) (invited Talk)
2. Amir, A., Aumann, Y., Benson, G., Levy, A., Lipsky, O., Porat, E., Skiena, S., Vishne, U.: Pattern matching with address errors: Rearrangement distances. In: Proc. 17th SODA, pp. 1221–1229 (2006)
3. Amir, A., Aumann, Y., Benson, G., Levy, A., Lipsky, O., Porat, E., Skiena, S., Vishne, U.: Pattern matching with address errors: Rearrangement distances. Journal of Computer and System Sciences 75(6) (2009)
4. Amir, A., Aumann, Y., Cole, R., Lewenstein, M., Porat, E.: Function matching: Algorithms, applications, and a lower bound. In: Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.) ICALP 2003. LNCS, vol. 2719, pp. 929–942. Springer, Heidelberg (2003)
5. Amir, A., Aumann, Y., Indyk, P., Levy, A., Porat, E.: Efficient computations of $\ell_1$ and $\ell_\infty$ rearrangement distances. In: Ziviani, N., Baeza-Yates, R. (eds.) SPIRE 2007. LNCS, vol. 4726, pp. 39–49. Springer, Heidelberg (2007)
6. Amir, A., Aumann, Y., Indyk, P., Levy, A., Porat, E.: Efficient computations of $\ell_1$ and $\ell_\infty$ rearrangement distances. Theoretical Computer Science 410(43), 4382–4390 (2009)
7. Amir, A., Aumann, Y., Kapah, O., Levy, A., Porat, E.: Approximate string matching with address bit errors. In: Ferragina, P., Landau, G.M. (eds.) CPM 2008. LNCS, vol. 5029, pp. 118–129. Springer, Heidelberg (2008)
8. Amir, A., Aumann, Y., Kapah, O., Levy, A., Porat, E.: Approximate string matching with address bit errors. Theoretical Computer Science 410(51) (2009); Special Issue of CPM 2008 Best Papers
9. Amir, A., Cole, R., Hariharan, R., Lewenstein, M., Porat, E.: Overlap matching. Information and Computation 181(1), 57–74 (2003)
10. Amir, A., Eisenberg, E., Keller, O., Levy, A., Porat, E.: Approximate string matching with stuck address bits (manuscript)

11. Amir, A., Hartman, T., Kapah, O., Levy, A., Porat, E.: On the cost of interchange rearrangement in strings. In: Arge, L., Hoffmann, M., Welzl, E. (eds.) ESA 2007. LNCS, vol. 4698, pp. 99–110. Springer, Heidelberg (2007)
12. Amir, A., Hartman, T., Kapah, O., Levy, A., Porat, E.: On the cost of interchange rearrangement in strings. SIAM Journal on Computing 39(4), 1444–1461 (2009)
13. Amir, A., Lewenstein, M., Porat, E.: Approximate swapped matching. Information Processing Letters 83(1), 33–39 (2002)
14. Angelov, S., Kunal, K., McGregor, A.: Sorting and selection with random costs. In: Laber, E.S., Bornstein, C., Nogueira, L.T., Faria, L. (eds.) LATIN 2008. LNCS, vol. 4957, pp. 48–59. Springer, Heidelberg (2008)
15. Bafna, V., Pevzner, P.A.: Sorting by transpositions. SIAM Journal on Discrete Mathematics 11, 221–240 (1998)
16. Bender, M.A., Ge, D., He, S., Hu, H., Pinter, R.Y., Skiena, S., Swidan, F.: Improved bounds on sorting with length-weighted reversals. In: Proc. 15th SODA, pp. 912–921 (2004)
17. Bender, M.A., Ge, D., He, S., Hu, H., Pinter, R.Y., Swidan, F.: Sorting by length-weighted reversals: Dealing with signs and circularity. In: Sahinalp, S.C., Muthukrishnan, S.M., Dogrusoz, U. (eds.) CPM 2004. LNCS, vol. 3109, pp. 32–46. Springer, Heidelberg (2004)
18. Berman, P., Hannenhalli, S.: Fast sorting by reversal. In: Hirschberg, D.S., Meyers, G. (eds.) CPM 1996. LNCS, vol. 1075, pp. 168–185. Springer, Heidelberg (1996)
19. Carpara, A.: Sorting by reversals is difficult. In: Proc. 1st Annual Intl. Conf. on Research in Computational Biology (RECOMB), pp. 75–83. ACM Press, New York (1997)
20. Cayley, A.: Note on the theory of permutations. Philosophical Magazine (34), 527–529 (1849)
21. Christie, D.A.: Sorting by block-interchanges. Information Processing Letters 60, 165–169 (1996)
22. Christie, D.A., Irving, R.W.: Sorting strings by reversals and by transpositions. SIAM Journal Discrete Math 14, 193–206 (2001)
23. Cole, R., Hariharan, R.: Verifying candidate matches in sparse and wildcard matching. In: Proc. 34st Annual Symposium on the Theory of Computing (STOC), pp. 592–601 (2002)
24. Fischer, M.J., Paterson, M.S.: String matching and other products. In: Karp, R.M. (ed.) SIAM-AMS Proceedings, Complexity of Computation, vol. 7, pp. 113–125 (1974)
25. Gupta, A., Kumar, A.: Sorting and selection with structured costs. In: Proc. 42nd Annual IEEE Symposium on Foundations of Computer Science (FOCS), pp. 416–425 (2001)
26. Heath, L.S., Vergara, J.P.C.: Sorting by bounded block-moves. Discrete Applied Mathematics 88(1-3), 181–206 (1998)
27. Heath, L.S., Vergara, P.C.: Sorting by short swaps. Journal of Computational Biology 10(5), 775–789 (2003)
28. Hennessy, J.L., Patterson, D.A.: Computer architecture: A quantitative approach, 3rd edn. Morgan Kauffmann, San Francisco (2002)
29. Jerrum, M.R.: The complexity of finding minimum-length generator sequences. Theoretical Computer Science 36, 265–289 (1985)
30. Kapah, O., Landau, G.M., Levy, A., Oz, N.: Interchange rearrangement: The element-cost model. In: Amir, A., Turpin, A., Moffat, A. (eds.) SPIRE 2008. LNCS, vol. 5280, pp. 224–235. Springer, Heidelberg (2008)

31. Kapah, O., Landau, G.M., Levy, A., Oz, N.: Interchange rearrangement: The element-cost model. Theoretical Computer Science 410(43), 4315–4326 (2009)
32. Karp, R., Miller, R., Rosenberg, A.: Rapid identification of repeated patterns in strings, arrays and trees. In: Symposium on the Theory of Computing, vol. 4, pp. 125–136 (1972)
33. Knuth, D.E., Morris, J.H., Pratt, V.R.: Fast pattern matching in strings. SIAM J. Comp. 6, 323–350 (1977)
34. Lowrance, R., Wagner, R.A.: An extension of the string-to-string correction problem. J. of the ACM, 177–183 (1975)
35. Schwartz, J.T.: Fast probabilistic algorithms for verification of polynomial identities. J. of the ACM 27, 701–717 (1980)
36. Yao, A.C.C.: Some complexity questions related to distributed computing. In: Proc. 11th Annual Symposium on the Theory of Computing (STOC), pp. 209–213 (1979)
37. Zippel, R.: Probabilistic algorithms for sparse polynomials. In: Ng, K.W. (ed.) EUROSAM 1979 and ISSAC 1979. LNCS, vol. 72, pp. 216–226. Springer, Heidelberg (1979)

# Maximal Words in Sequence Comparisons Based on Subword Composition

Alberto Apostolico*

Georgia Institute of Technology & Università di Padova

**Abstract.** Measures of sequence similarity and distance based more or less explicitly on subword composition are attracting an increasing interest driven by intensive applications such as massive document classification and genome-wide molecular taxonomy. A uniform character of such measures is in some underlying notion of relative compressibility, whereby two similar sequences are expected to share a larger number of common substrings than two distant ones. This paper reviews some of the approaches to sequence comparison based on subword composition and suggests that their common denominator may ultimately reside in special classes of subwords, the nature of which resonates in interesting ways with the structure of popular subword trees and graphs.

## 1 Structure, Similarity and Distance

The problem of comparing, classifying and indexing long textual files from large collections is becoming increasingly severe as web applications, digital libraries and genomic studies expand to an unprecedented scale. Established techniques of the past rarely work in these contexts. In computational molecular biology, for instance, edit distances become both computationally unbearable and scarcely significant when they are applied to entire genomes, and are being supplanted by global similarity measures that refer, implicitly or explicitly, to the subword composition of sequences (see, e.g., [4, 9, 10, 13, 14, 15, 16, 23, 24, 25, 30, 31, 33]).

Measures of density and dispersion over datasets are sometime classified as *distributive, algebraic* and *holistic* (refer, e.g., to [11]). Distributivity pertains to situations in which the measure can be applied to sub-aggregates and then combined as in, e.g., *sum, max, min*. A typical algebraic measure is *average*, which can be obtained by algebraic combination of distributive functions. The opposite of algebraic is holistic, which applies to measures that are impossible to compute by divide and conquer. E.g., knowing the rank of an element in a subset does not tell anything about the rank in the whole set, whence *rank, median, mode*, and the likes are holistic measures.

* College of Computing, Georgia Institute of Technology, 801 Atlantic Drive, Atlanta, GA 30318, USA, *and* Dipartimento di Ingegneria dell' Informazione, Università di Padova, Padova, Italy axa@cc.gatech.edu Work Supported in part by the Italian Ministry of University and Research under the Bi-National Project FIRB RBIN04BYZ7, by United States - Israel Binational Science Foundation (BSF) Grant No. 2008217, and by the Research Program of Georgia Tech.

T. Elomaa et al. (Eds.): Ukkonen Festschrift, LNCS 6060, pp. 34–44, 2010.

Here we will be concerned with algebraic measures of similarity among sequences, to be further adapted to the derivation of corresponding distances. In loose terms, one would like to base such measures on some notion of information content of sequences, but this has proved an elusive goal since von Mises' pioneering pursuit of the essence of randomness [32] and probably reaches well into the future [18]. Classical formalizations of such a notion include Shannon's information theory [8], Brillouin's adoption of Shannon's redundancy [6], and Kolmogorov's approach to information [20] which Lempel and Ziv specialized [21] to effective and elegant data compression methods.

In Shannon's formulation, a *finite scheme* is given comprising $n$ events, where event $E_i$ occurs with probability $p_i, i = 1, 2, , n$. If $p_i$ is very small and yet $E_i$ happens we will experience a very big surprise. At the other extreme, if $p_i = 1$ and $E_i$ occurs then there is no surprise. Therefore, we can use a monotonically decreasing function as a measure of the surprise caused by $E_i$. Specifically, we take $\log p_i$, which satisfies also the important additivity condition, that the surprise of joint events is the sum of the individual surprises. Now, $E_i$ is expected to occur with probability $p_i$ , whence the notion of expected surprise (sic), which is called the *entropy*. This is minimum (0) if one event has probability 1, maximum ($\log n$) if the $p_i$ s are equal. In this setting, information corresponds to the decrease of a-priori ignorance. By virtue of the important

**Theorem 1.** *(Gibbs Theorem) For any two distributions $p_i$ and $q_i$ on the events $E_i$'s*

$$-\sum p_i \log p_i \leq -\sum p_i \log q_i$$

one has that

$$\sum p_i \log(\frac{p_i}{p_j})$$

is always positive.

Considered as a measure of the information *content* of a sequence, Shannon's formalization was not immune from controversy. Brillouin [6] preferred to equate entropy with chaos and to assign the role of information to *redundancy* or *negentropy*. A possible explanation for such a radical divergence may reside with the fact that Shannon was concerned with information in transmission, whereas the one Brillouin wanted to capture was *stored* information, a notion more akin to organization and structure.

An important alternative was proposed by Kolmogorov [20]. According to it, the information content (alternatively, conditional information) or *Kolmogorov complexity* $K(x)$ of a string $x$ is the length of a shortest program in binary by which a universal Turing machine produces one string from scratch (alternatively, from another string). It is not hard to see that such a minimal description of a string cannot be too much larger than the string itself.

**Theorem 2.** *There is a constant $c$ such that $K(x) \leq |x| + c$.*

Unfortunately, there is no program that produces the integer $K(x)$ from an input string $x$.

**Theorem 3.** *$K$ is not a computable function.*

This is established by exhibiting an absurd program that would generate a string that could only be generated by a longer program. Nevertheless, it is easy to find an upper bound for $K(x)$: simply compress the string $x$ by one of the available methods, concatenate the compressed string to its suitably encoded decompressor, and measure the resulting length. A string $x$ is *compressible* if $K(x) \leq |x| - c$ for some constant $c$, otherwise $x$ is *incompressible* by $c$. One sees by a pigeonhole argument that incompressible strings are unavoidable. In fact, the programs of length less than $k$ are at most:

$$\lambda, 0, 1, 00, 01, 10, 11, ..., ..., 11...1 \text{ (or } k \text{ “1”)}$$

but the number of strings with a program of length less than k is

$$1 + 2 + ... + 4 + 2^{k-1} = 2^k - 1 < 2^k.$$

This suggests that, in the limit, a great many sequences of sufficiently large length are seen to be incompressible and hence will appear as random.

Any attempt at classifying sequences presupposes some measure of *similarity* or *distance*, i.e., a function $D(x, y)$ that assigns a real number to every pair of strings $x$ and $y$ such that the more $x$ is similar to $y$ the smaller is the value of $D(x, y)$. It seems especially desirable for a measure of similarity to enjoy the property of symmetry. As is well known, a *metric* distance must satisfy the three properties of:

1. non-negativity ($D(x, y) \geq 0$ and $D(x, y) = 0$ iff $x = y$) ;
2. symmetry ($D(x, y) = D(y, x)$);
3. triangle inequality ($D(x, y) + D(y, z) \geq D(x, z)$).

With sequences, it is generally hard to secure all or even only some of these properties. I will briefly review the notion of similarity under various models.

Shannon's notion of information invokes germane ones of *conditional* and *mutual* information, upon which it seems natural to articulate measures of similarity in terms of some kind of relative compressibility [8]. Considering the transition from one source to another, one immediate way to express gain in information is the difference between their two distributions, i.e.,

$$-\sum_{x \in X} p(x) \log p(x) + \sum_{x \in X} q(x) \log q(x).$$

This measure is global and can be either positive or negative. A better measure is the *Kullback-Leibler divergence* between the two distributions [8]

$$KL(p|q) = -\sum_{x \in X} p(x) \log \frac{p(x)}{q(x)},$$

further generalized by Alfred Renyi [26], which relies on Gibbs theorem above and is thus always positive. The $KL$ divergence tells the expected cost, in terms

of extra bits, needed to identify a value $x$ drawn from $X$, if a code is used that is tailored to the probability distribution $Q$, rather than the "true" distribution $P$. The "second term" of $KL$

$$D(p|q) = -\sum_{x \in X} p(x) \log q(x) = -E_p(\log q(X)),$$

may be interpreted as the asymptotic cost of compressing one string produced by an i.i.d. probability distribution $p(x)$ using the optimum dictionary developed for an i.i.d. distribution $q(x)$. This extends to markovian probability distributions as

$$D(p|q) = lim_{n \to \infty} -\frac{1}{n} \sum_{x^n \in X^n} p(x^n) \log q(x^n) = -E_p(\log q(X^n)).$$

One can see that $D(p|q) \neq D(q|p)$ whence $D$ is not a metric. One way to introduce symmetry (though not the triangle inequality) is through the Jensen-Shannon divergence as adopted in [27], which consists of

$$JS(p,q) = \frac{1}{2} KL(p|r) + \frac{1}{2} KL(q|r)$$

where the distribution $r(x) = p(x) + q(x)$. However, the fact that $D(p|q)$ is a natural distance between markovian distributions gives a theoretical basis for using the average longest substring as a distance [30]. If $x$ and $y$ are generated by markovian distributions of respective densities $p$ and $q$, then as the length of the string goes to infinity $-E_p \log q(X)$ is approximated by the length of the average common substring ($ACS$ for short) [30], a measure somewhat reminiscent of one used in [28] for approximate string matching. $ACS(x,y)$ is computed by taking, for every position $i$ of $x$, the length $\ell(i)$ of the longest substring of $y$ that can be copied starting at that position and averaging over all $n$ positions of $x$ to get $ACS(x,y) = \sum_i \ell(i)/n$. One then takes $ACS(x,y)/\log m$ to normalize with respect to the length $m$ of $y$.

Early compositional similarities based on relative abundance of $k$-mers are credited to Blaisdell [4] who used them in a euclidean distance between transition matrices. Karlin and Burge [19] found that some genomic signatures could be derived from the distribution of dinucleotides. Extensions were derived in [25], where each organism is represented by a *composition vector* the components of which correspond to the numbers of various (overlapping) $k$-peptides, for a fixed $k$, in all the translated amino acid sequences from an organism's genome. The numbers are modified by subtracting a statistical background to highlight the role of selective evolution. The subtraction procedure is based on a ($k$-2)-th order Markov prediction and therefore the minimum $k$ is 3.

Let $x$ be a sequence of length $n$ and consider, for each word $w[1..k]$ of a given length $k$ in $x$, the expression [25]:

$$a(w) = \begin{cases} \frac{p(w[1..k]) - p^o(w[1..k])}{p^o(w[1..k])} & \text{for } p^o(w[1..k]) \neq 0 \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

where $p(w[1..k])$ is the observed ratio $f(w)/(L - |w| + 1)$ between the count (possibly, zero) and the number of possible occurrences of the word $w$ in $x$, and $p^o(w[1..k])$ is the *markovian estimate* of the probability $p$ defined as

$$p^o(w[1..k]) = \frac{p(w[1..k-1])p(w[2..k])}{p(w[2..k-1])}.$$

This expression may be arrived at in multiple ways. For instance [12], begin by counting the occurrences $f(w)/(n - |w| + 1)$ of $w[1..k]$. Now, express the corresponding probability as $p(w[1..k]) = p(w_k|(w[1..k-1])p(w[1..k-1])$, where the conditional probability is unknown. The weak assumption that the farthest character can be neglected yields the estimate $p^o(w[1..k]) = p(w_k|(w[2..k-1])p(w[1..k-1])$ . Writing now one more exact expression for $p(w[2..k-1]) = p(w_k|w[2..k-1])p(w[2...k-1])$, and eliminating the unknown probability leads to the above expression for $p^o$.

With easy passages $a(w)$ can be rewritten as

$$a(w) = \begin{cases} \Lambda_k \times \frac{f(w[1..k])f(w[2..k-1])}{f(w[1..k-1])f(w[2..k])} - 1 \text{ for} \\ \quad f(w[1..k-1]) \geq 1 \quad \text{and} \quad f(w[2..k]) \geq 1 \\ 0 \qquad\qquad\qquad\qquad \text{otherwise} \end{cases} \tag{2}$$

where

$$\Lambda_k = \frac{(n-k+2)^2}{(n-k+1)(n-k+3)} \quad ,$$

so that the difference between the empirical probability of $w$ and its Markov-based prediction, divided by the latter is represented by Expression 2 as well. For a given collection of words (e.g., the set of all $k$-mers for a fixed $k$), all $a$-values are stored , in some suitable order in a vector, called the *composition vector*.

For two composition vectors $A$ and $B$, the similarity between the corresponding strings is measured by the cosine of the angle between the corresponding vectors in multidimensional space.

$$c(A,B) = \frac{\sum a_i b_i}{\left(\sum a_i^2 \times \sum b_i^2\right)^{1/2}} \tag{3}$$

where the $a_i$'s and $b_i$'s are computed by applying Expression 2 respectively to $A$ and $B$.

The *conditional Kolmogorov complexity* $K(x|y)$ of $x$ given $y$ is the length of the shortest program to compute $x$ from input string $y$. This is different from the *joint complexity* $K(x,y)$, which is the length of the shortest program which outputs the concatenation of $x$ and $y$. The relation between the conditional and joint complexities is $K(x|y) = K(x,y) - K(y)$. When $y$ contains no information about $x$, then $K(x|y) = K(x)$ and $K(x,y) = K(x) + K(y)$. Since the function $K$ is not computable, it is approximated in practice by some standard compressor such as, e.g., those falling in the family of Lempel-Ziv [21,34]. As is well known, the classical paradigm proceeds as follows:

1. initialize a dictionary to contain all characters of the alphabet $\Sigma$;
2. assume to have encoded $x[1, ..., i]$; let $s$ be the longest prefix of $x[i+1, ..., n]$ that has an occurrence starting at some position $j \leq i$, and let $x[i + |s| = a$; then append to the encoding the next *phrase* as the triplet $(j, |s|, a)$;
3. repeat the process starting at $x[i + |s| + 1]$.

As mentioned, $KL$ is not a metric distance, since it does not obey symmetry, and neither is $JS$. A distance may be based on $ACS$ [30] by first taking the inverse of the *similarity* measure $ACS(x, y)/\log m$ and then subtracting a term to guarantee the condition $d(x, x) = 0$. Specifically, this yields

$$\tilde{d} = \log m/ACS(x, y) - \log n/ACS(x, x),$$

where the correction term

$$\log n/ACS(x, x) = 2 \log n/n$$

vanishes as $n \to \infty$. Following this, one compensates for symmetry by taking

$$\frac{d(x, y) = \tilde{d}(x, y) + \tilde{d}(y, x)}{2}$$

as the final distance.

A distance based on Kolmogorov theory can be defined as [22]:

$$d(x, y) = 1 - \frac{K(x) - K(x|y)}{K(xy)}.$$

The numerator may be interpreted as the amount of information that $y$ knows about $x$ and, by a deep property of Kolmogorov complexity $K(x) - K(x|y) \approx K(y) - K(y|x)$, whereas the denominator serves as a normalizing factor, whence $d(x, y)$ ranges between 0 and 1. It s possible to prove that this $d$ is a metric distance if inequalities hold up to a $\log n$ factor. This can be further refined in [23]:

$$d(x, y) = \frac{max\{K(y|x), K(x|y)\}}{max\{K(x), K(y)\}} = \frac{K(x, y) - min\{K(y), K(x)\}}{max\{K(x), K(y)\}} \tag{4}$$

If a standard compressor such as the Lempel-Ziv one described above is used, it is possible to adapt it by approximating $K(x)$ by $C(x)$, the compressed version of $x$ and to compute $C(x|y)$ or $C(y|x)$ by concatenating the two strings and then letting the recopying be confined to the first one. Expression 4 translates then into:

$$\frac{C(x, y) - min\{C(x), C(y)\}}{max\{C(x), C(y)\}}$$

It should be noted that under this approximation $d(x, x) = 0$ is not obeyed in general by a real compressor, which defies the desirable condition $C(x, x) = C(x)$.

Finally, it is easy to build a distance measure on the cosine correlation of two composition vectors $A$ and $B$, by setting:

$$d(A,B) = 1/2\left(1 - \frac{\sum a_i b_i}{(\sum a_i^2 \times \sum b_i^2)^{1/2}}\right) \tag{5}$$

where the $a_i$'s and $b_i$'s are computed by applying Expression 2 respectively to $A$ and $B$.

## 2 The Ubiquitous Maximal Subwords and Related Computations

The apparent success of approaches to molecular taxonomy based on relatively short $k$-mer composition has been explained in more than one way. In [25], for instance, it is argued that the primordial soup must have contained only a small fraction of the about 64,000,000 possible 6-mers, which would then limit the populace of 6-mers produced by growth, fusion and mutation. This seems confirmed by the circumstance that the known existing proteins contain only a fraction of the possible 6-mers. For instance, the circa 100,000 proteins in SWISS-PROT feature less than 26% of the 6-mers [25]. At the same time, for very short $k$ one might expect to find all of the possible $k$-mers, a potential bias that calls for preprocessing filters of the kind described in [27]. With a random sequence, one expects in principle that for $k$ up to $\approx \log n$ all $|\Sigma|^k$ $k$-mers occur, whence $k$-mers in this range grow exponentially with $k$. At the same time, the number of $k$-mers in a sequence of $n$ characters is only $O(n)$ for any fixed $k$, while the total number of distinct words of any length found in that sequence can be at most $\Theta(n^2)$. Many of the sequence distances based on $k$-mer composition seem to indicate the existence of an optimum value of $k$. To test this fact, an extension [2] of the measure of [25] was developed which consists of a linear-time algorithm to compute composition vectors that include all (possibly $\Theta(n^2)$) words in the input sequences up to any arbitrary maximum length $K$. The experiments exhibited an increasing distortion in the resulting classifications with growing value of $K$. This seems in contrast with the good performance achieved on a variety of inputs by methods hinged on substrings of unbounded length such as $ACS$ or Kolmogorov-Ziv-Lempel compression which use, implicitly or explicitly, subwords of unbounded lengths. In the remainder of this section, I will focus on some special subsets of all substrings of a sequence that seem to be implicated in this second class of methods. As it turns out, when the unbounded-length composition vector distances in [2] are applied to these subsets rather than to the entire set of subwords, this seems to lead to convergence, instead of distortion, with increasing $K$.

It is well known (cf., e.g., [1,5]), that a notable family of less than $2n$ words exist that are *maximal* in the host sequence in the sense that it is impossible to extend a word in this class by appending one or more characters to it without losing some of its occurrences. More formally, one can actually define three

equivalence relations on the subwords in $x$, as follows. The first one puts in the same equivalence class strings that have precisely the same set of *starting* positions. The second one symmetrically assigns to the same class strings with the same *ending* positions. The third one is the transitive closure of the first two. The natural representatives in each equivalence relation will be taken as the words of maximum length in each class, which will be called here *right-maximal*, *left-maximal* or just *maximal*, respectively. Close relatives of these strings were called *special factors* in [7] and at any rate they all find a nice resonance in data structures such as *directed acyclic word graphs* and *subword trees* [1,5]. One very remarkable property of these partitions is the following [5]

**Theorem 4.** *The index of each equivalence relation is linear in the length of the host string.*

It is easily checked that in the recopying process inherent the Lempel-Zv compression paradigm in [21], every phrase is intrinsically a right-maximal word. Therefore, the related practical implementations of Kolmogorov complexity implicitly rely on such words. A subset of right-maximal words is involved in the $ACS$ distance measures [30]. The substrings involved in these approaches correspond to branching nodes of some suitable *suffix tree* [1, 29], built on a single string or on the concatenation of two strings. For example, imagine that for two input sequences their respective tries are drawn each with a different color, and then superimposed. Then the words considered in $ACS$ correspond to the longest substrings on each path from the root bearing both colors. If the terminal characters of the two input strings are unique to either string then any such path must end at a branching node, hence the corresponding words are some, thought not all of the right-maximal words for the combined input.

It is interesting to revisit the extension to all values of $k$ of the $k$-mer approach of [25] from the perspective of the equivalence relations above. Clearly, the $\Lambda$ term in the expression of $a(w)$ tends to 1 for $n$ much larger than $k$, but it must be accounted for in the exact computation of $a(w)$ for all, virtually $\Theta(n^2)$, $k$-mers. If the computation of $a(w)$ per Expression 2 is hinged on the term $w[2..k-1]$, then one sees that when such a word is not right-maximal this makes

$$\frac{f(w[1..k])f(w[2..k-1])}{f(w[1..k-1])f(w[2..k])} = 1,$$

whence $a(w) = 0$. In fact, it must be the case that $f(w[2..k-1]) = f(w[2..k])$ whence also $f(w[1..k-1]) = f(w[1..k])$ (since any suffix of a right-maximal word is right-maximal). Thus, the burden of computing $a(w)$ for non-maximal words is imposed solely by the $\Lambda$ factor. In [2], it is shown that this computation can still be carried out in overall linear time, but this entails some care in tallying the contribution of the virtually $\Theta(n^2)$ words that are not right-maximal. The un-aesthetic presence of $\Lambda$ in the expression of $a(w)$ is dissolved by an elegant setup devised by Andreas Dress and described in [3], which amounts to substitute right-maximal words with maximal words *tout court*, and carefully defining the probabilities at play.

As is customary, it simplifies the discussion to introduce the extended alphabet $\hat{\Sigma} = \Sigma \cup \{\$\}$, where this time \$ will be both prefixed and appended to the input string, that will be still referred to simply by $x$. For any substring $w$ of $x$, $(w \in \Sigma^*)$, consider the relative frequencies of one-letter extensions of the form $wa$, $bw$ and $bwa$ $(a, b \in \hat{\Sigma})$. Since, for any $w \in \Sigma^+$,

$$\sum_{a,b \in \hat{\Sigma}} f(bwa) = \sum_{a \in \hat{\Sigma}} f(wa) = \sum_{b \in \hat{\Sigma}} f(bw) = f(w),$$

then, $\forall a \in \hat{\Sigma}$

$$p(wa|w) = \frac{f(wa)}{f(w)} \quad \text{and} \quad p(aw|w) = \frac{f(aw)}{f(w)}$$

are probability distributions on $\hat{\Sigma}$, and $\forall w \in \Sigma^+$

$$p(bwa|w) = \frac{f(bwa)}{f(w)} \quad \text{and} \quad p^o(bwa) = p(bw|w)\ p(wa|w)$$

are probability distributions on $\hat{\Sigma} \times \hat{\Sigma}$. The first two distributions can be empirically derived, whereas the last two may be used to estimate.

We can now write for $y = bwa$ an expression similar to Expression 1 and yet thoroughly homogeneous in terms of *relative* probabilities, as:

$$\hat{a}(y) = \frac{p(y|w) - p^o(y|w)}{p^o(y|w)}$$

and, with the convention $\ln \frac{p(bwa|w)}{p^o(bwa|w)} = 0$ for $p^o(bwa|w) = 0$, the Kullback-Leibler distance

$$KL(p|p^o) = \sum_{(a,b) \in \hat{\Sigma} \times \hat{\Sigma}} p(bwa|w) \log \frac{p(bwa|w)}{p^o(bwa|w)}.$$

For any $w \in \Sigma^+$ and characters $a, b \in \hat{\Sigma}$, we have $p(bwa|w) = p^o(bwa|w)$ if and only if $f(bwa)\, f(w) = f(bw)\, f(wa)$. This is true when both $f(bwa) = f(bw)$ and $f(w) = f(wa)$ and in fact, as already observed, as soon as $f(w) = f(wa)$. Of course the converse is not true, i.e., one might have $f(bwa) = f(bw)$ and yet $f(w) \neq f(wa)$ . The longest word $y$ in a chain obeying $f(bwa) = f(bw)$ is a maximal word for the entire group of its substrings that occur only within the context of $y$.

As is well known, we can find right-maximal words at the branching nodes of a suffix tree, left-maximal words at the nodes of a Directed Acyclic Word Graph (DAWG) [5]. Where do we look for maximal words? These words are found at the intersection of the suffix trees and the DAWG for a string. In fact, it is possible to produce a DAWG [5] as the result of a two-steps transformation of a suffix tree. The first step consists of identifying and juxtaposing of all roots of

isomorphic subtrees. This produces a directed acyclic graph with one source and one sink taking linear space for nodes and edges except for the edge labels, that can charge quadratic space in the worst case. The second step further modifies the structure thereby reducing the overall space to linear. It is enough for our purposes to perform the first step and, interestingly enough, there is even no need to resort to the general linear-time tree isomorphism test [17], by virtue of the following, easy to prove

**Theorem 5.** *Any two subtrees of a suffix tree are isomorphic if and only if they have the same number of leaves and their roots are connected by a chain of suffix links.*

## 3 Conclusions

The variety of approaches to sequence distances based on subword similarity invoke, implicitly or explicitly, some notion of maximal substrings. The intriguing property underpinning this notion is the fact, that while a string of $n$ characters can host $\Theta(n^2)$ distinct substrings, the number of substrings that are maximal in any of the ways discussed here is only $O(n)$. The natural *habitat* for this phenomenon are data structures such as subword automata and trees for which efficient and beautiful constructions have been set up over a period of now more than three decades. In this author's experience, this is also one remarkable case where a subtle combinatorial property on strings chooses to incarnate into a data structure.

## References

1. Apostolico, A.: The myriad virtues of suffix trees. In: Apostolico, A., Galil, Z. (eds.) Combinatorial Algorithms on Words, pp. 85–96. Springer, Berlin (1985)
2. Apostolico, A., Denas, O.: Fast algorithms for computing sequence distances by exhaustive substring composition. Algorithms for Molecular Biology 3 (2008)
3. Apostolico, A., Denas, O., Dress, A.: Efficient tools for comparative substring analysis (submitted, 2009)
4. Blaidsell, B.: A measure of the similarity of sets of sequences not requiring sequence alignment. Proceedings of the National Academy of Sciences, 5155–5159 (1986)
5. Blumer, A., Blumer, J., Haussler, D., Ehrenfeucht, A., Chen, M.T., Seiferas, J.I.: The smallest automaton recognizing the subwords of a text. Theor. Comput. Sci. 40, 31–55 (1985)
6. Brillouin, L.: Science and Information Theory. Academic Press, London (1971)
7. Colosimo, A., de Luca, A.: Special factors in biological strings. J. Theor. Biol. 204, 29–47 (2000)
8. Cover, T.M., Thomas, J.A.: Elements of Information Theory. Wiley-Interscience, Hoboken (1991)
9. Edgar, R.: Local homology recognition and distance measures in linear time using compressed amino-acid alphabets. Bioinformatics 32, 380–385 (2004)
10. Ferragina, P., Giancarlo, R., Greco, V., Manzini, G., Valiente, G.: Compression-based classification of biological sequences and structures via the universal similarity metric: experimental assessment. BMC Bioinformatics 8, 252–272 (2007)
11. Han, J., Kamber, M.: Data mining: concepts and techniques. Morgan Kaufmann Publishers Inc., San Francisco (2000)

12. Hao, B.: Personal communication (2008)
13. Hao, B., Qi, J.: Procaryote phylogeny without sequence alignment: from avoidance singature to composition distance. Journal of Bioinformatics and Computational Biology 2, 1–19 (2004)
14. Van Helden, J.: Metrics for comparing regulatory sequences on the basis of pattern counts. Bioinformatics 20, 399–406 (2004)
15. Höhl, M., Ragan, M.A.: Is multiple-sequence alignment required for accurate inference of phylogeny? Syst. Biol. 56(2), 206–221 (2007)
16. Höhl, M., Rigutsos, I., Ragan, M.A.: Pattern-based phylogenetic distance estimation and tree recosntruction. Evolutionary Bioinformatics Online 2, 357–373 (2006)
17. Hopcroft, J.E., Wong, J.K.: Linear time algorithm for isomorphism of planar graphs (preliminary report). In: STOC, pp. 172–184 (1974)
18. Brooks Jr., F.P.: Three great challenges for half-century-old computer science. J. ACM 50(1), 25–26 (2003)
19. Karlin, S., Burge, C.: Dinucleotide relative abundance extremes: a genomic signature. Trends in genetics: TIG 11(7), 283–290 (1995)
20. Kolmogorov, A.N.: Three approaches to the quantitative definition of information. Problemi Pederachi Inf. 1 (1965)
21. Lempel, A., Ziv, J.: On the complexity of finite sequences. IEEE Transactions on Information Theory 22, 75–81 (1976)
22. Li, M., Badger, J.H., Chen, X., Kwong, S., Kearney, P.E., Zhang, H.: An information-based sequence distance and its application to whole mitochondrial genome phylogeny. Bioinformatics 17(1), 149–154 (2001)
23. Li, M., Chen, X., Li, X., Ma, B., Vitányi, P.M.B.: The similarity metric. IEEE Transactions on Information Theory 50(12), 3250–3264 (2004)
24. Otu, H., Sayood, K.: A new sequence distance measure for phylogenetic tree reconstruction. Bioinformatics 19, 2122–2130 (2003)
25. Qi, J., Wang, B., Hao, B.: Whole proteome prokaryote phylogeny without sequence alignment: A k-string composition approach. Molecular Evolution 58(1), 1–11 (2004)
26. Rényi, A.: On measures of information and entropy. In: Proceedings of the 4th Berkeley Symposium on Mathematics, Statistics and Probability, pp. 547–561 (1960)
27. Sims, G.E., Jun, S.R., Wu, G.A., Kim, S.H.: Alignment-free genome comparison with feature frequency profiles (ffp) and optimal resolutions. Proceedings of the National Academy of Sciences 106(8), 2677–2682 (2009)
28. Ukkonen, E.: Approximate string matching with q-grams and maximal matches. Theor. Comput. Sci. 92(1), 191–211 (1992)
29. Ukkonen, E.: On-line construction of suffix trees. Algorithmica 14(3), 249–260 (1995)
30. Ulitsky, I., Burstein, D., Tuller, T., Chor, B.: The average common substring approach to phylogenetic reconstruction. Journal of Computational Biology 13(2), 336–350 (2006)
31. Vinga, S., Almeida, J.: Alignment-free sequence comparison – a review. Bioinformatics 20, 206–215 (2004)
32. von Mises, R.: Probability, Statistics and Truth. MacMillan, Basingstoke (1939)
33. Wu, T.J., Bruke, J., Davison, D.: A measure of DNA dissimilarity based on the mahalanobis distance between frequencies of words. Biometrics 53, 1431–1439 (1997)
34. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Transactions on Information Theory 23(3), 337–343 (1977)

# Fast Intersection Algorithms for Sorted Sequences

Ricardo Baeza-Yates[1] and Alejandro Salinger[2]

[1] Yahoo! Research, Barcelona, Spain & Santiago, Chile
[2] Dept. of Computer Science, Univ. of Waterloo, Canada

**Abstract.** This paper presents and analyzes a simple intersection algorithm for sorted sequences that is fast on average. It is related to the multiple searching problem and to merging. We present the worst and average case analysis, showing that in the former, the complexity nicely adapts to the smallest list size. In the latter case, it performs less comparisons than the total number of elements on both inputs, $n$ and $m$, when $n = \alpha m$ ($\alpha > 1$), achieving $O(m \log(n/m))$ complexity. The algorithm is motivated by its application to fast query processing in Web search engines, where large intersections, or differences, must be performed fast. In this case we experimentally show that the algorithm is faster than previous solutions.

## 1 Introduction

Our problem is a particular case of a generic problem called multiple searching [2] (see also [20], research problem 5, page 156). Given an $n$-element data multiset, $D$, drawn from an ordered universe, search $D$ for each element of an $m$-element query multiset, $Q$, drawn from the same universe. An algorithm solving the problem must report any elements in both multisets. The metric is the number of three-way comparisons ($<, =, >$) between any pair of elements, worst case or average case. Throughout this paper $n \geq m$ and logarithms are base two unless explicitly stated otherwise.

Multiply search is directly related to computing the intersection of two sets. In fact, the elements found is the intersection of both sets. Although in the general case, $D$ and $Q$ are arbitrary, an important case is when $D$ and $Q$ are sets (and not multisets) already ordered. In this case, multiply search can be solved by merging both sets. However, this is not optimal for all possible cases. In fact, if $m$ is small (say if $m = o(n/\lg n)$), it is better to do $m$ binary searches obtaining an $O(m \lg n)$ algorithm. Can we have an adaptive algorithm that matches both complexities depending on the value of $m$? We present an algorithm which on average performs less than $m+n$ comparisons when both sets are ordered under some pessimistic assumptions. Fast average case algorithms are important for large $n$ and/or $m$.

This problem is motivated by Web search engines. Most search engines use inverted indexes, where for each different word, we have a list of positions or documents where it appears. In some settings those lists are ordered by position

T. Elomaa et al. (Eds.): Ukkonen Festschrift, LNCS 6060, pp. 45–61, 2010.

or by a global precomputed ranking, to facilitate set operations between lists (derived from Boolean query operations), which is equivalent to the ordered case. In other settings, the lists of positions are sorted by frequency of occurrence in a document, to facilitate ranking based on the vector model [1,3]. The same happens with word positions in each file (full inversion to allow sentence searching). Therefore, the complexity of this problem is interesting also for practical reasons, as in search engines, partial lists can have hundreds of millions elements for very frequent words.

In Section 2 we present related work. Section 3 presents our intersection algorithm for two sequences as well as its analytical and experimental analysis. We also extend the algorithm to multiple sequences. Section 4 presents several hybrid algorithms tuned through experimental analysis. Section 5 presents the motivation for our problem, Web search engines, and experimental results for this case. We end with some concluding remarks and on-going work. This paper is an extended and revised version of [6,7].

## 2 Related Work

If an algorithm determines whether any elements of a set of $n + m$ elements are equal, then, by the element uniqueness lower bound in algebraic-decision trees (see [16]), the algorithm requires $\Omega((n+m)\lg(n+m))$ comparisons in the worst case. However, this lower bound does not apply to the search problem because a search algorithm does not need to determine the uniqueness of either $D$ or $Q$; it need only to determine whether $D \cap Q$ is empty. For example, an algorithm for $m = 1$ must find whether some element of $D$ equals the element in $Q$, not whether any two elements of $D$ are equal. Conversely, however, lower bounds on the search problem (or, equivalently, the set intersection problem) apply to the element uniqueness problem [15]. In fact, this idea was exploited by Demaine *et al.* to define an *adaptive multiple set* intersection algorithm [13,14]. They also defined the difficulty of a problem instance, which was refined later by Barbay and Kenyon [8].

This adaptive algorithm [13,14] works as follows: we take one of the sets, and we choose its first element, which we call $x$. We search $x$ in the other set, making exponential jumps, this is, looking at positions $1, 2, 4, \ldots, 2^i$. If we overshoot, that is, the element in the position $2^i$ is larger than $x$, we binary search $x$ between positions $2^{i-1}$ and $2^i$. This is an application of what is called *doubling search* or *galloping search*, which mimics binary search for unbounded sequences obtaining the same $O(\log n)$ complexity, a classical result of Bentley and Yao [10]. If we find $x$, we add it to the result. Then, we remember the position where $x$ was (or the position where it should have been) so we know that from that position backwards we already processed the set. Now we set $x$ as the smallest element of the set that is greater than the former $x$ and we exchange roles, making jumps from the position that signals the processed part of the set. We finish when there is no element greater than the one we are searching.

For the ordered case, lower bounds on set intersection are also lower bounds for merging both sets. However, the converse is not true, as in set intersection

we do not need to find the actual position of each element in the union of both sets, just if it is in $D$ or not. Although there has been a lot of work on minimum comparison merging in the worst case, almost no research has been done on the average case because it does not make much of a difference. However, this is not true for multiple search, and hence for set intersection [2].

In the case of merging, Fernandez de la Vega *et al.* [18] analyzed the average case of a simplified version of Hwang-Lin's binary merge [19] finding that if $\alpha = n/m$ with $\alpha > 1$ and not a power of 2, then the expected number of comparisons is

$$\left( r + \frac{1}{1 - \left(\frac{\alpha}{\alpha+1}\right)^{2^r}} \right) \frac{n}{\alpha} ,$$

with $r = \lfloor \lg_2 \alpha \rfloor$. When $\alpha$ is a power of 2, the result is more complicated, but similar. Simplifying, the average complexity is $O(m \log(n/m))$. Fernandez de la Vega *et al.* [17] also designed a probabilistic algorithm that improved upon Hwang-Lin's algorithm on the worst case for $1.618m \leq n \leq 3m$.

In the case of upper bounds, good algorithms for multiple search can be used to compute the intersection of two sets, obtaining the same time complexity. They can be also used to compute the union of two sets, by subtracting the intersection of both sets to the set obtained by merging both sets. Similarly to compute the difference of two sets.

As the most time-demanding operation on inverted indexes is the merging or intersection of the lists of occurrences, it is important to optimize it. Consider one pair of lists of sizes $m$ and $n$ respectively, that needs to be intersected. If $m$ is much smaller than $n$, it is better to do $m$ binary searches in the larger list to do the intersection, obtaining an $O(m \lg n)$ algorithm. Hence, if $m$ is $o(n/\lg n))$, this algorithm is better than the linear merging algorithm that has complexity $O(n + m)$. Notice that each binary search can be performed in what was left to the right of the larger list in the previous binary search.

Later, Baeza-Yates' [6] devised a double binary search algorithm that is very fast if the intersection is trivially empty ($O(\log n)$) and requires less than $m + n$ comparisons on average. The exact average complexity is $O(m \log(n/m)))$ and although it is not shown explicitly in the original paper [13], the Adaptive algorithm also has the same average complexity for two sequences. A recent paper does a thorough performance comparison of these and other algorithms showing that the best algorithm depends also in the data distribution [9]. However, two other papers show that for compressed lists the best algorithms can be quite different [21,12].

## 3 A Simple But Good Average Case Algorithm

Suppose that $D$ is sorted. In this case, obviously, if $Q$ is small, will be faster to search every element of $Q$ in $D$ by using binary search. Can we do better if both sets are sorted? In this case set intersection can be solved by merging. In the

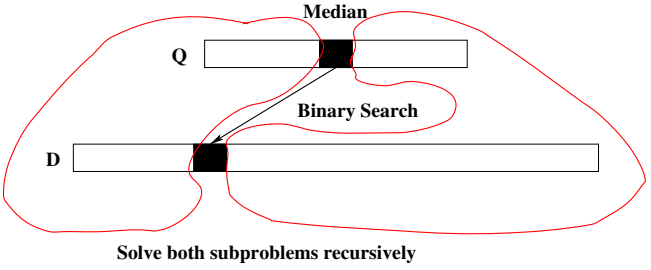



**Fig. 1.** Divide and conquer step of double binary search

worst or average case, straight merging requires $m+n-1$ comparisons. Can we do better for set intersection? The following simple algorithm improves on average under some pessimistic assumptions. We call it double binary search and can be seen as a balanced version of Hwang and Lin's [19] algorithm adapted to our problem, although in the literature is also called the Baeza-Yates' intersection algorithm (see for example [21]).

### 3.1 Double Binary Search

We first binary search the median (middle element) of $Q$ in $D$. If found, we add that element to the result (a technical caveat is described later). Found or not, we have divided the problem in searching the elements smaller than the median of $Q$ to the left of the position found on $D$, and the elements bigger than the median to the right of that position. We then solve recursively both parts using the same algorithm. If in any case, the size of the subset of $Q$ to be considered is larger than the subset of $D$, we exchange the roles of $Q$ and $D$. Note that set intersection is symmetric in this sense. If any of the subsets is empty, we do nothing. Figure 1 shows this divide and conquer approach.

An important detail is that if we want to use this algorithm in a sequential fashion, the output sequence should be sorted. For this, the comparison of the median should be done in between the two recursive calls as in Quicksort. Figure 3.1 shows the algorithm in pseudo-code.

A simple way to improve this algorithm is to start comparing the smallest elements of both sets with the largest elements in both sets. If both sets do not overlap, we use just $O(1)$ time. Otherwise, we search the smallest and largest element of $D$ in $Q$, to find the overlap, using just $O(\lg m)$ time. Then we apply the previous algorithm just to the subsets that actually overlap. This improves both, the worst and the average case. The dual case is also valid, but then finding the overlap is $O(\lg n)$, which is not good for small $m$.

In the case of variable size lists, these algorithms can be applied in sequence by following the lists in order, like in the merging algorithm.

```
Intersect(D, Q, minD, maxD, minQ, maxQ)
1.  //if Q or D are empty, we finish the recursion
2.  if minD > maxD bfor minQ > maxQ
3.      return ∅
4.  miqQ ← round((minQ + maxQ)/2)
5.  midQval ← Q[midQ]
6.  midD ← binsearch(midQval, D, minD, maxD)
7.  if |D[minD..midD − 1]| > |Q[minQ..midQ − 1]| // subset(D) > subset(Q)
8.      Result ← Result ∪ Intersect(D, Q, minD, midD − 1, minQ, midQ − 1)
9.  else //we exchange the roles of D and Q
10.     Result ← Result ∪ Intersect(Q, D, minQ, midQ − 1, minD, midD − 1)
11. if D[midD] == midQval
12.     Result ← Result ∪ {midQval}
13.     midD ← posModD − 1
14. if |D[midD + 1..maxD]| > |Q[midQ + 1..maxQ]|// subset(D) > subset(Q)
15.     Result ← Result ∪ Iintersect(D, Q, midD, maxD, midQ + 1, maxQ)
16. else //we exchange the roles of D and Q
17.     Result ← Result ∪ Intersect(Q, D, midQ + 1, maxQ, midD, maxD)
18. return Result
```

**Fig. 2.** Double binary search algorithm for intersecting two sorted sequences

### 3.2 Best and Worst Case Analysis

In the best case, the median element in each iteration always falls outside $D$ (that is, all the elements in $Q$ are smaller or larger than all the elements in $D$). Hence, the total number of comparisons is $\lceil \lg(m+1) \rceil \lceil \lg(n+1) \rceil$, which for $m = O(n)$ is $O(\lg^2 n)$. This shows that there is room for doing less work. The worst case happens when the median is not found and divides $D$ into two sets of the same size (intuitively seems that the best and worst case are reversed). Hence, if $W(m,n)$ is the cost of the set intersection in the worst case, for $m$ of the form $2^k - 1$, we have

$$W(m,n) = \lceil \lg(n+1) \rceil + W((m-1)/2, \lceil n/2 \rceil) + W((m-1)/2, \lfloor n/2 \rfloor) \ .$$

It is not difficult to show that

$$W(m,n) = 2(m+1)\lg((n+1)/(m+1)) + 2m + O(\lg n) \ .$$

That is, for small $m$ the algorithm has $O(m \lg n)$ worst case, while for $n = \alpha m$ it is $O(n)$. In this case, the ratio between this algorithm and merging is $2(1+\lg(\alpha))/(1+\alpha)$ asymptotically, being 1 when $\alpha = 1$. The worst case is worse than merging for $1 < \alpha < 6.3197$ having its maximum at $\alpha = 2.1596$ where it is 1.336 times slower than merging (this is shown in Figure 5). Hence the worst case of the algorithm matches the complexity of both, the merging and the multiple binary search, approaches, adapting nicely to the size of $m$. Figure 3 shows these two cases (top).

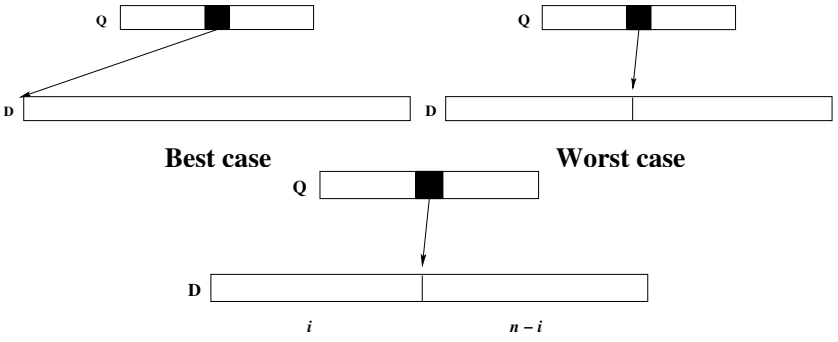



**Fig. 3.** Best and worst (top) as well as average (bottom) case analysis

### 3.3 Average Case Analysis

Let us consider now the average case. We use two assumptions: first, that we never find the median of $Q$ and hence we assume that some elements never appear in $D$; and second, that the median will divide $D$ in sets of size $i$ and $n-i$ with the same probability for all $i$ (this is equivalent to consider every element on $D$ as random, like in the average case analysis of Quicksort). The first assumption is pessimistic, while the second considers that overlaps are uniformly distributed, which is also pessimistic regarding our practical motivation as we do not take in account that word occurrences may and will have locality of reference. The recurrence that we have to solve is

$$A(m,n) = \lceil \lg(n+1) \rceil + A(\lceil (m-1)/2 \rceil, \lceil n/2 \rceil) + A(\lceil (m-1)/2 \rceil, \lfloor n/2 \rfloor) \ ,$$

with $A(m,n) = A(n,m)$ if $m > n$ and $A(m,0) = A(0,n) = 0$, where $A(m,n)$ is the average number of comparisons to intersect two lists of size $m$ and $n$. The rationale for this formula is shown in Figure 3 (bottom). Figure 4 shows the actual number of comparisons for $n = 128$ and all powers of 2 for $m \leq n$, for all the cases already mentioned.

To analyze the recurrence above, let us consider, without loss of generality, the case for $m$ of the form $2^k - 1$. Then we have

$$A(m,n) = \lceil \lg(n+1) \rceil + \frac{1}{n+1} \sum_{i=0}^{n} (A((m-1)/2, i) + A((m-1)/2, n-i)) \ .$$

We now show that

$$A(m,n) = (m+1)(\ln((n+1)/(m+1)) + 3 - 1/\ln(2)) + O(\lg n)$$

The recurrence equation can be simplified to

$$A(m,n) = \lceil \lg(n+1) \rceil + \frac{2}{n+1} \sum_{i=0}^{n} A((m-1)/2, i) \ .$$

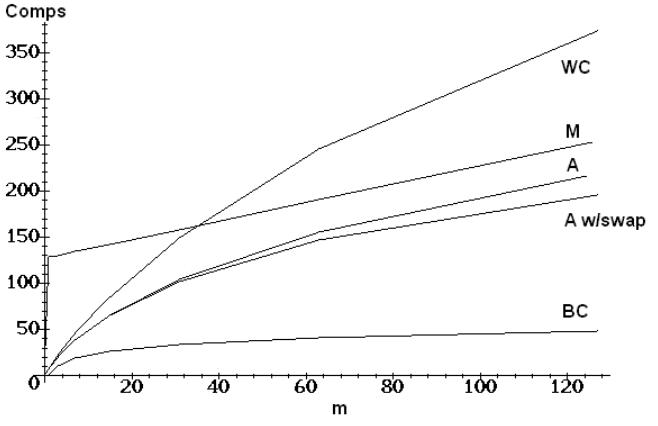



**Fig. 4.** Number of comparisons in the best, worst and average case (with and without swaps) for $n = 128$, as well as for merging (M)

As the algorithm is adaptive on the size of the lists, we have

$$A(m,n) = \lceil \lg(n+1) \rceil + \frac{2}{n+1} \left[ \sum_{i=0}^{(m-1)/2} A\left(i, \frac{m-1}{2}\right) + \sum_{(m+1)/2}^{n} A\left(\frac{m-1}{2}, i\right) \right]$$

by noticing that we switch the sets when $m > n$. However, solving this version of the recurrence is too hard, so we do not include this improvement in the analysis. Nevertheless, this does not affect the main order term. Notice that our analysis allows any value for $n$.

Making the change of variable $m = 2^k - 1$ and using $k$ as sub-index we get

$$A_k(n) = \lceil \lg(n+1) \rceil + \frac{2}{n+1} \sum_{i=0}^{n} A_{k-1}(i) \ .$$

Eliminating the sum, we obtain

$$(n+1)A_k(n) = nA_k(n-1) + 2A_{k-1}(n) + \lceil \lg(n+1) \rceil + n\delta(n = 2^j) \ ,$$

where $\delta(n = 2^j)$ is 1 if $n$ is a power of 2, or 0 otherwise. Let $T_n(z) = \sum_k A_k(n) z^k$ be the generating function of $A$ in the variable $k$. Hence

$$T_n(z) = \frac{n}{n+1-2z} T_{n-1}(z) + \frac{\lceil \lg(n+1) \rceil + n\delta(n = 2^j)}{(n+1-2z)(1-z)} \ .$$

Unwinding the recurrence in the sub-index of the generating function, as $T_0(z) = 0$, we get

$$T_n(z) = \frac{n!}{(1-z)\Gamma(n+2-2z)} \sum_{i=1}^{n} \frac{\Gamma(i+1-2z)}{i!} (\lceil \lg(i+1) \rceil + i\delta(i = 2^j)) \ ,$$

where $\Gamma(x)$ is the Gamma function (if $x$ is a positive integer, then $\Gamma(x) = (x-1)!$). Let $\alpha_{i,j}$ be $\lceil \lg(i+1) \rceil + i\delta(i = 2^j)$. Simplifying, we have

$$T_n(z) = \frac{1}{(1-z)(n+1)} \sum_{i=1}^{n} \prod_{j=i+1}^{n+1} \frac{\alpha_{i,j}}{\left(1 - \frac{2z}{j}\right)} .$$

Expanding we have

$$T_n(z) = \frac{\sum_{r \geq 0} z^r}{n+1} \sum_{i=1}^{n} \prod_{j=i+1}^{n+1} \alpha_{i,j} \sum_{\ell \geq 0} \left(\frac{2z}{j}\right)^{\ell} .$$

Now, we have $A(2^k - 1, n) = [z^k]T_n(z)$ where $[z^k]f(z)$ is the coefficient of $z^k$ in $f(z)$. As the coefficient of $z^r$ is 1, for $r \leq k$ we need to compute in the right side the coefficient of $z^{k-r}$. That is

$$A(2^k - 1, n) = \frac{1}{n+1} \sum_{r=0}^{k} \sum_{i=1}^{n} [z^{k-r}] \prod_{j=i+1}^{n+1} \alpha_{i,j} \sum_{\ell \geq 0} \left(\frac{2z}{j}\right)^{\ell} .$$

Then

$$A(2^k - 1, n) = \frac{1}{n+1} \sum_{r=0}^{k} 2^{k-r} \sum_{i=1}^{n} \prod_{\sum_{j=i+1}^{n+1} \ell_j = k-r} \alpha_{i,j} \left(\frac{1}{j^{\ell_j}}\right)^{\ell} .$$

With the help of the Maple symbolic algebra system, we obtain the main order terms sought.

For $n = \alpha m$, the ratio between this algorithm and merging is $(\ln(\alpha) + 3 - 1/\ln(2))/(1+\alpha)$ which is at most 0.7913 when $\alpha = 1.2637$ and 0.7787 when $\alpha = 1$. This is also shown in figure 5, where we also include the average case analysis of Hwang and Lin's algorithm [18]. Recall that this analysis uses different assumptions, however shows the same behavior, improving over merging when $\alpha \geq 2$.

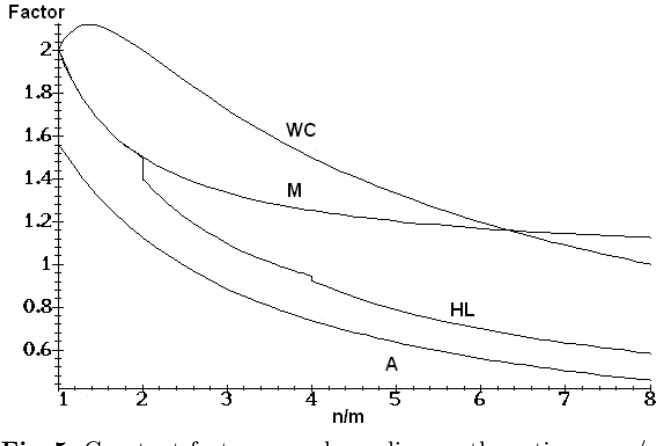



**Fig. 5.** Constant factor on $n$ depending on the ratio $\alpha = n/m$

### 3.4 Experimental Analysis

Now we compare the efficiency of the algorithm, which we call *Intersect* in this section, with an intersection algorithm based on merging, and with an adaptation of the *Adaptive* algorithm [13,14] for the intersection of two sequences. In addition, we show the results obtained with the optimizations of the algorithm.

We used sequences of integer random numbers, uniformly distributed in the range $[1, 10^9]$. We varied the length of one of the lists ($n$) from 1,000 to 22,000 with a step of 3,000. For each of these lengths we intersected those sequences with sequences of four different lengths ($m$), from 100 to 400. We use twenty random instances per case and ten thousand runs (to eliminate the variations due to the operating system given the small resulting times).

The programs were implemented in $C$ using the Gcc 3.3.3 compiler in a Linux platform running an Intel(R) Xeon(TM) CPU 3.06GHz with 512 Kb cache and 2Gb RAM.

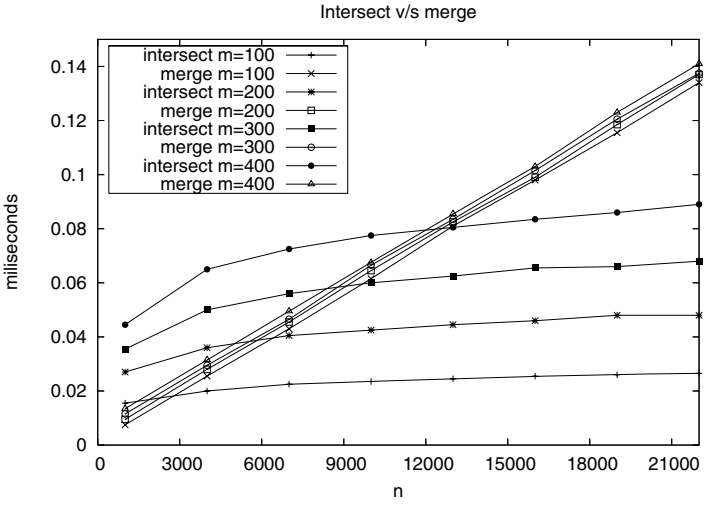



**Fig. 6.** Experimental results for Intersect and Merge for different values of $n$ and $m$

Figure 6 shows a comparison between Intersect and Merge. We can see that Intersect is better than Merge when $n$ increases and that the time increases for larger values of $m$.

Figure 7 shows a comparison between the times of Intersect and Adaptive. We can see that the times of both algorithms follow the same tendency and that Intersect is marginally better than Adaptive.

Figure 8 shows the results obtained with the Intersect algorithm and the optimization described at the end of the last section. For this comparison, we also added the computation of the overlap of both sequences to Merge.

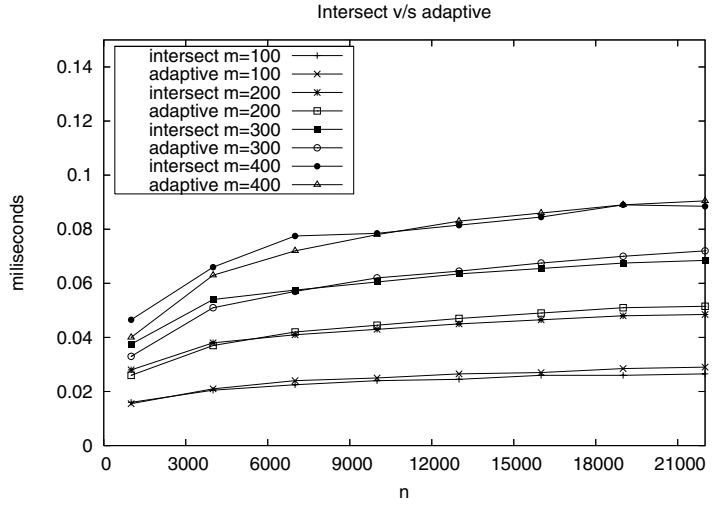



**Fig. 7.** Experimental results for Intersect and Adaptive, for different values of $n$ and $m$

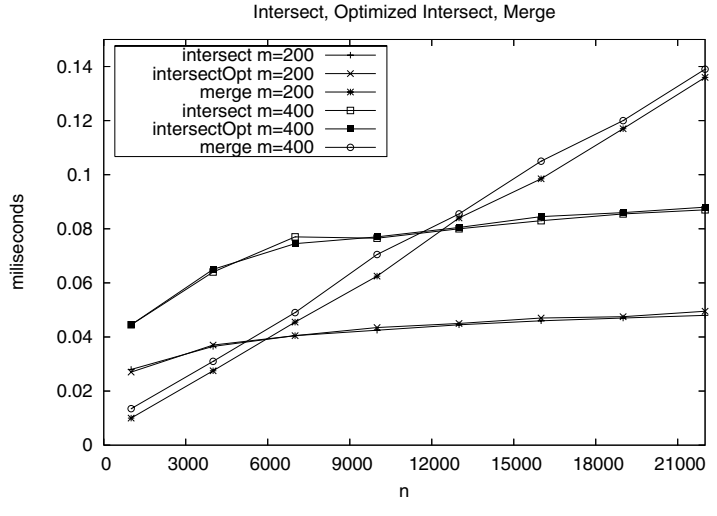



**Fig. 8.** Experimental results for Intersect, optimized Intersect and Merge, for different values of $n$ and $m = 200$ and $m = 400$

We can see that there is no big difference between the original and the optimized algorithm, and moreover, the original algorithm was a bit faster than the optimized one. The reason why the optimization did not result in an improvement can be the uniform distribution of the test data. As the random numbers are uniformly distributed, in most cases the overlap of both sets covers a big part of $Q$. Then, the optimization does not produce any improvement and it only results in a time overhead due to the overlap search.

### 3.5 Multiple Sequences

When there are more than two lists, there are several possible heuristics depending on the list sizes. One possible algorithm is two process just pairs of sequences. For three lists the best solution will be to intersect the two shortest lists and then intersect the result with the longer list. For four lists or more the heuristic will depend on the partial answers, and hence has to be adaptive. In general, doing a balanced merging tree that avoids the long lists until the end will perform well. On the other hand, in practice we will not have more than 6 to 8 lists. Hence, if intersecting the two shortest lists gives a very small answer, might be better to intersect that to the next shortest list, and so on. In general the optimal algorithm will depend in the partial answers and hence we would need a dynamic programming algorithm to obtain it.

Other possibility is to extend our algorithm to the case of $L$ lists. That is, we search the median of the shortest list on the other $L-1$ lists. If we find the median in the $L-1$ lists, we add that to the result. Next, we solve recursively for all the elements that are less than the median and for all the elements that are larger than the median. The complexity in this case is similar to the two sequence case using $m$ as the length of the shortest list and $n$ as the length of the rest of the lists.

## 4 Hybrid Algorithms

We can see from the experimental results obtained for the basic algorithm that there is a section of values of $n$ where Merge is better than Intersect. Hence, a natural idea is to combine both algorithms in one hybrid algorithm that runs each of them when convenient. However this will depend on the data, the implementation, and the actual hardware and software platform used. So the following discussion is based on our context but can be replicated, possibly with different results, for other cases.

In order to know where is the cutting point to use one algorithm instead of the other, we measured for each value of $n$ the time of both algorithms with different values of $m$ until we identified the value of $m$ where Merge was faster than Intersect. These values of $m$ form a straight line as a function of $n$, which we can observe in Fig. 9. This straight line is approximated by $m = 0.033n + 8.884$, with a correlation of $r^2 = 0.999$.

The hybrid algorithm works by running Merge whenever $m > 0.033n + 8.884$, and running Intersect otherwise. The condition is evaluated on each step of the recursion.

When we modify the algorithm, the cutting point changes. We would like to find the optimal hybrid algorithm. Using the same idea again, we found the straight line that defines the values where Merge is better than the hybrid algorithm. This straight line can be approximated by $m = 0.028n + 32.5$, with $r^2 = 0.992$. Hence, we define the algorithm Hybrid2, which runs Merge whenever $m > 0.028n + 32.5$ and runs Intersect otherwise. Finally, we combined both

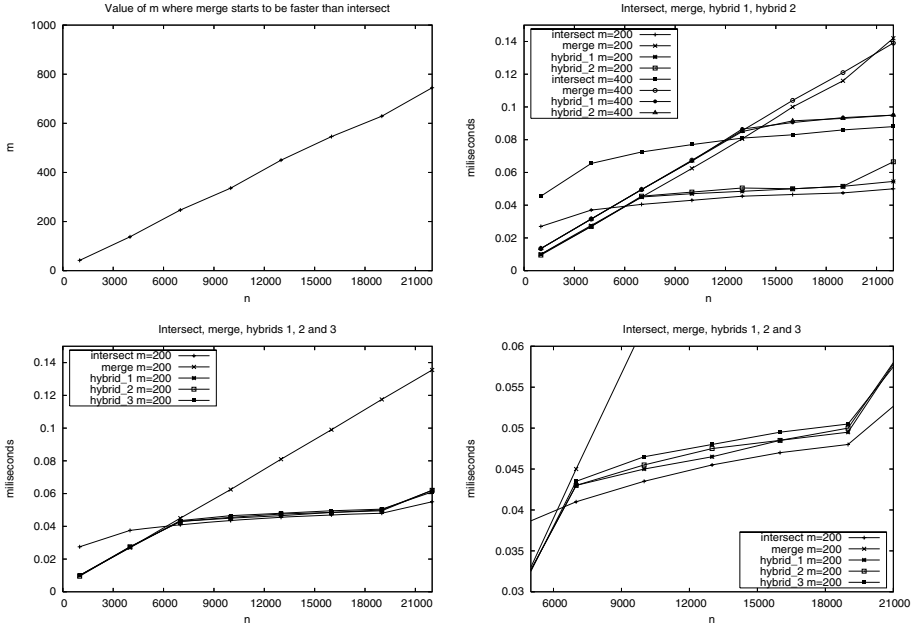



**Fig. 9.** Up: on the left, value of $m$ from which Merge is faster than Intersect. On the right, a comparison between the original algorithm, Merge and the hybrids 1 and 2 for $m = 200$ and $m = 400$. Down: comparison between Intersect, Merge and the three hybrids for $m = 200$. The plot on the right is a zoom of the one on the left.

hybrids, creating a third version where the cutting line between Merge and Intersect is the average between the lines of the hybrids 1 and 2. The resulting straight line is $m = 0.031n + 20.696$. Figure 9 shows the cutting line between the original algorithm and Merge, and the results obtained with the hybrid algorithms. The optimal algorithm would be on theory the Hybrid.$i$ when $i$ tends to infinity, as we are looking for a fixed point algorithm.

We can observe that the hybrid algorithms registered lower times than the original algorithm in the section where the latter is slower than Merge. However, in the other section the original algorithm is faster than the hybrids, due to the fact that in practice we have to evaluate the cutting point in each step of the recursion. Among the hybrid algorithms, we can see that the first one is slightly faster than the second one, and that this one is faster than the third one. An idea to reduce the time in the section that the original algorithm is faster than the hybrids is to create a new hybrid algorithm that runs Merge when it is convenient and that then runs the original algorithm, without evaluating the relation between $m$ and $n$ in order to run Merge. This algorithm shows the same times than Intersect in the section where the latter is better than Merge, combining the advantages of both algorithms in the best way. Figure 10 show the results obtained with this new hybrid algorithm.

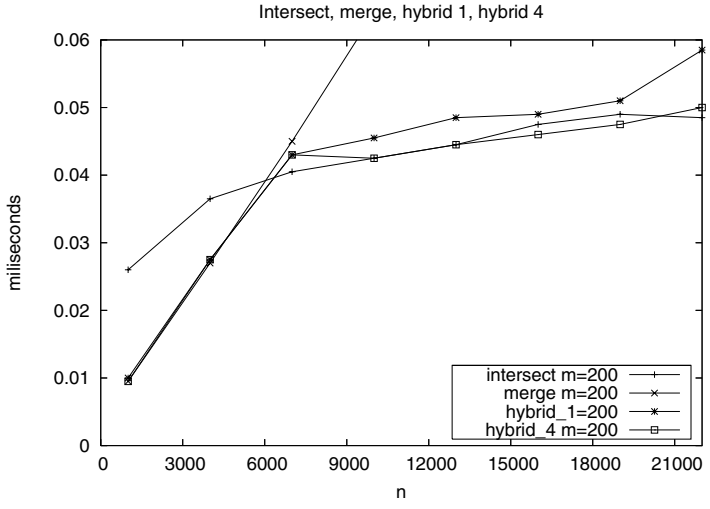



**Fig. 10.** Experimental results for Intersect, Merge and the hybrids 1 and 4 for different values of $n$ and for $m = 200$

## 5 Application to Query Processing in Inverted Indexes

### 5.1 Context

Inverted indexes are used in most text retrieval systems [3]. Logically, they are a vocabulary (set of unique words found in the text) and a list of references per word to its occurrences (typically a document identifier and a list of word positions in each document). In simple systems (Boolean model), the lists are sorted by document identifier, and there is no ranking (that is, there is no notion of relevance of a document). In that setting, our basic algorithm applies directly to compute Boolean operations on document identifiers: union is equivalent to merging, intersection is the complement operation (we only keep the repeated elements), and subtraction implies deleting the repeated elements. In practice, long lists are not stored sequentially, but in blocks. Nevertheless, these blocks are large, and the set operations can be performed in a block-by-block basis.

In complex systems ranking is used. Ranking is typically based in word statistics (number of word occurrences per document and the inverse of the number of documents having it). Both values can be precomputed and the reference lists are then stored by decreasing intra-document word frequency order to have first the most relevant documents. Lists are then processed by decreasing inverse extra-document word frequency order (that is, we process the shorter lists first), to obtain first the most relevant documents. However, in this case we cannot always have a document identifier mapping such that lists are sorted by that order.

The previous scheme was used initially on the Web, but as the Web grew, the ranking deteriorated because word statistics do not always represent the content and quality of a Web page and also can be "spammed" by repeating and adding

(almost) invisible words. In 1998, Brin and Page [11] described a search engine (which was the starting point of Google) that used links to rate the quality of a page. This is called a global ranking based in popularity, and is independent of the query posed. It is out of the scope of this paper to explain Pagerank, but it models a random Web surfer and the ranking of a page is the probability of the Web surfer visiting it. This probability induces a total order that can be used as document identifier. Hence, in a pure link based search engine we can use our intersection algorithm as before. However, nowadays hybrid ranking schemes that combine link and word evidence are used. In spite of this, a link based mapping still gives good results as approximates well the true ranking (which can be corrected while is being computed).

Another important type of query is sentence search. In this case we use the word position to know if a word follows or precedes a word. Hence, as usually sentences are small, after we find the Web pages that have all of them, we can process the first two words to find adjacent pairs and then those with the third word and so on. This is like to compute a particular intersection where instead of finding repeated elements we try to find correlative elements ($i$ and $i + 1$), and therefore we can use again our algorithm as word positions are sorted. The same is true for proximity search. In this case, we can have a range $k$ of possible valid positions (that is $i \pm k$) or to use a different ranking weight depending on the proximity.

Finally, in the context of the Web, our algorithm is in practice much faster because the uniform distribution assumption is pessimistic. In the Web, the distribution of word occurrences is quite biased. The same is true with query frequencies. Both distributions follow a power law (a generalized Zipf distribution) [3,5]. However, the correlation of both distributions is very small [4]. That implies that the average length of the lists involved in the query are not that biased. That means that the average lengths of the lists, $n$ and $m$, when sampled, will satisfy $n = \Theta(m)$ (uniform), rather than $n = m + O(1)$ (power law). Nevertheless, in both cases our algorithm makes an improvement. Now we study this case experimentally.

### 5.2 Sequence Lengths with Zipf's Distribution

Now we study the behavior of the Intersect algorithm depending of the ratio between the lengths of the two sequences when these lengths follow a Zipf distribution and the correlation between both sets is zero (ideal case). For this experiment, we took two random numbers, $a$ and $b$, uniformly distributed between 0 and 1,000. With these numbers we computed the lengths of the sequences $D$ and $Q$ as $n = K/a^\alpha$ and $m = K/b^\alpha$, respectively, with $K = 10^9$ and $\alpha = 1.8$ (a typical value for word occurrence distribution in English), making sure that $n > m$. We did 1,000 measurements, using 80 different sequences for each of them, and repeating 1,000 times each run.

Figure 11 shows the times obtained with both algorithms as a function of $n/m$, in normal scale and logarithmic scale.

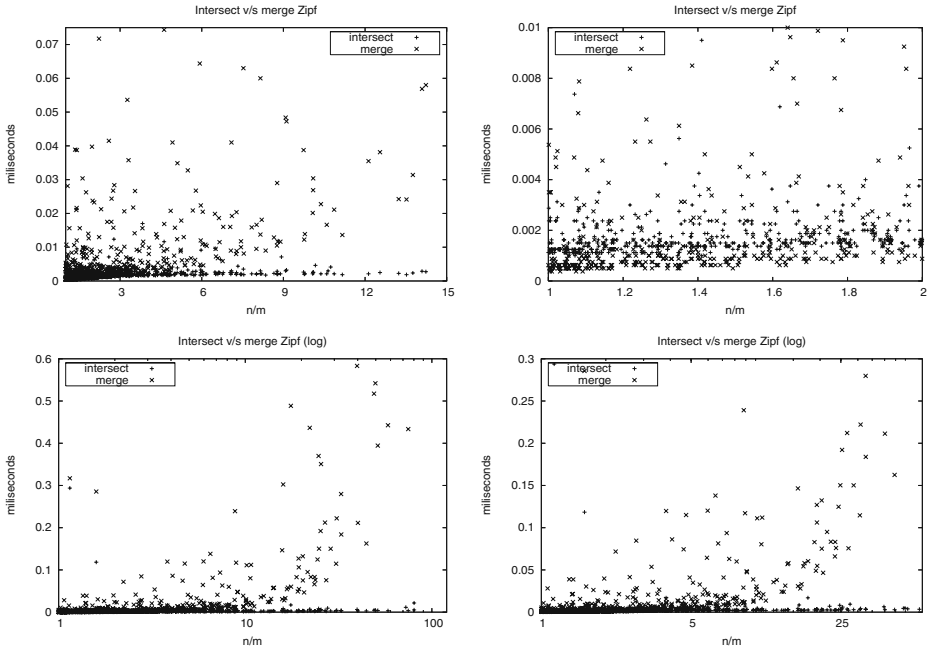



**Fig. 11.** Up: times for Intersect and Merge as a function of the ratio between the lengths of the sequences when they follow a Zipf distribution. The plot on the right is a zoom of the one on the left. Down: times for Intersect and Merge in logarithmic scale. The plot on the right is a zoom of the one on the left.

We can see that the times of Intersect are lower than the times of Merge when $n$ is much greater than $m$. When we decrease the ratio between $n$ and $m$, it is not so clear anymore which of the algorithms is faster. When $n/m < 2$, in most cases the times of Merge are better.

## 6 Concluding Remarks

We have presented a simple set intersection algorithm that performs quite well in average and does not inspect all the elements involved. It can be seen as a natural hybrid of binary search and merging. Our experiments show that the algorithm is also faster than Merge in practice when one of the sequences is much larger than the other one. This improvement is more evident when $n$ increases. In addition, our algorithm surpasses Adaptive [13,14] for every ratio between the sizes of the sequences. The hybrid algorithm that combines our algorithm with Merge according to the empiric information obtained, takes advantage of both algorithms and became the most efficient one.

In practice, queries are short (on average 2 to 3 words [5]) so there is almost no need to do multiset intersection and if so, they can be easily handled by pairing the smaller sets firsts, which seems to be the most used algorithm [14].

In addition, we do not need to compute the complete result, as most people only look at less than two result pages [5]. Moreover, computing the complete result is too costly if one or more words occur several millions of times as happens in the Web and that is why most search engines use an intersection query as default. Hence, lazy evaluation strategies are used. If we use the straight classical merging algorithm, this naturally obtains first the most relevant Web pages. For our algorithm, it is not so simple, because although we have to process first the left side of the recursive problem, the Web pages obtained do not necessarily appear in the correct order. A simple solution is to process the smaller set from left to right doing binary search in the larger set. However this variant is efficient only for small $m$, achieving a complexity of $O(m \lg n)$ comparisons. An optimistic variant can use a prediction on the number of pages in the result and use an intermediate adaptive scheme that divides the smaller sets in non-symmetric parts with a bias to the left side. Hence, it is interesting to study the best way to compute partial results efficiently.

As the correlation between both sets in practice is between 0.2 and 0.6, depending on the Web text used (Zipf distribution with $\alpha$ between 1.6 and 2.0) and the queries (Zipf distribution with a lower value of $\alpha$, for example 1.4), we would like to extend our experimental results to this case. However, we already saw that in both extremes (correlation 0 or 1), the algorithm studied is competitive.

## References

1. Baeza-Yates, R.A.: Efficient Text Searching. PhD thesis, Dept. of Computer Science. University of Waterloo (May 1989); Also as Research Report CS-89-17
2. Baeza-Yates, R.A., Bradford, P.G., Culberson, J.C., Rawlins, G.J.E.: The Complexity of Multiple Searching (1993) (unpublished manuscript)
3. Baeza-Yates, R.A., Ribeiro-Neto, B.: Modern Information Retrieval, 513 pages. ACM Press/Addison-Wesley, England (1999)
4. Baeza-Yates, R.A., Saint-Jean, F.: A three level search engine index based in query log distribution. In: Nascimento, M.A., de Moura, E.S., Oliveira, A.L. (eds.) SPIRE 2003. LNCS, vol. 2857, pp. 56–65. Springer, Heidelberg (2003)
5. Baeza-Yates, R.A.: Query usage mining in search engines. In: Scime, A. (ed.) Web Mining: Applications and Techniques. Idea Group, USA (2004)
6. Baeza-Yates, R.A.: A fast set intersection algorithm for sorted sequences. In: Sahinalp, S.C., Muthukrishnan, S., Dogrusöz, U. (eds.) CPM 2004. LNCS, vol. 3109, pp. 400–408. Springer, Heidelberg (2004)
7. Baeza-Yates, R.A., Salinge, A.: Experimental analysis of a fast intersection algorithm for sorted sequences. In: Consens, M.P., Navarro, G. (eds.) SPIRE 2005. LNCS, vol. 3772, pp. 13–24. Springer, Heidelberg (2005)
8. Barbay, J., Kenyon, C.: Adaptive Intersection and $t$-Threshold Problems. In: Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms, San Francisco, CA, January 2002, pp. 390–399 (2002)
9. Barbay, J., López-Ortiz, A., Lu, T., Salinger, A.: An experimental investigation of set intersection algorithms for text searching. Journal of Experimental Algorithms (JEA) 14(3), 7–24 (2009)
10. Bentley, J.L., Yao, A.C.-C.: An Almost Optimal Algorithm for Unbounded Searching. Information Processing Letters 5, 82–87 (1976)

11. Brin, S., Page, L.: The anatomy of a large-scale hypertextual Web search engine. In: 7th WWW Conference, Brisbane, Australia (April 1998)
12. Culpepper, J., Moffat, A.: Compact set representation for information retrieval. In: Ziviani, N., Baeza-Yates, R. (eds.) SPIRE 2007. LNCS, vol. 4726, pp. 137–148. Springer, Heidelberg (2007)
13. Demaine, E.D., López-Ortiz, A., Munro, J.I.: Adaptive set intersections, unions, and differences. In: Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms, San Francisco, CA, January 2000, pp. 743–752 (2000)
14. Demaine, E.D., López-Ortiz, A., Munro, J.I.: Experiments on adaptive set intersections for text retrieval systems. In: Buchsbaum, A.L., Snoeyink, J. (eds.) ALENEX 2001. LNCS, vol. 2153, pp. 91–104. Springer, Heidelberg (2001)
15. Dietz, P., Mehlhorn, K., Raman, R., Uhrig, C.: Lower Bounds for Set Intersection Queries. In: Proceedings of the $4^{th}$ Annual Symposium on Discrete Algorithms, pp. 194–201 (1993)
16. Dobkin, D., Lipton, R.: On the Complexity of Computations Under Varying Sets of Primitives. Journal of Computer and Systems Sciences 18, 86–91 (1979)
17. Fernandez de la Vega, W., Kannan, S., Santha, M.: Two probabilistic results on merging. SIAM J. on Computing 22(2), 261–271 (1993)
18. Fernandez de la Vega, W., Frieze, A.M., Santha, M.: Average case analysis of the merging algorithm of Hwang and Lin. Algorithmica 22(4), 483–489 (1998)
19. Hwang, F.K., Lin, S.: A Simple algorithm for merging two disjoint linearly ordered lists. SIAM J. on Computing 1, 31–39 (1972)
20. Rawlins, G.J.E.: Compared to What?: An Introduction the the Analysis of Algorithms. Computer Science Press/W.H. Freeman (1992)
21. Sanders, P., Transier, F.: Intersection in integer inverted indices. In: ALENEX 2007, pp. 71–83 (2007)

# Indexing and Searching a Mass Spectrometry Database

Søren Besenbacher[1], Benno Schwikowski[2], and Jens Stoye[3]

[1] deCODE Genetics, Reykjavik, Iceland
sorenb@decode.is
[2] Systems Biology Group, Institut Pasteur, Paris, France
benno@pasteur.fr
[3] Technische Fakultät, Universität Bielefeld, Germany
stoye@techfak.uni-bielefeld.de

**Abstract.** Database preprocessing in order to create an index often permits considerable speedup in search compared to the iterated query of an unprocessed database. In this paper we apply index-based database lookup to a range search problem that arises in mass spectrometry-based proteomics: given a large collection of sparse integer sets and a sparse query set, find all the sets from the collection that have at least $k$ integers in common with the query set. This problem arises when searching for a mass spectrum in a database of theoretical mass spectra using the *shared peaks count* as similarity measure. The algorithms can easily be modified to use the more advanced *shared peaks intensity* measure instead of the shared peaks count. We introduce three different algorithms solving these problems. We conclude by presenting some experiments using the algorithms on realistic data showing the advantages and disadvantages of the algorithms.

## 1 Background

Large-scale protein identification methods play a critical role for systems biology approaches [1]. In peptide mass fingerprinting and tandem mass spectrometry, an experimental spectrum is compared to large databases of theoretical spectra in time-consuming linear sweeps [13]. While sequence databases have already been growing exponentially [17], there are a number of recent developments that indicate even stronger growth in the databases of theoretical spectra that need to be searched in unbiased proteomics approaches. These developments include significant increases in the capacity of high-throughput sequencing [11], and the realization that cells abundantly employ post-transcriptional modifications, such as alternative splicing [7], and single-residue modifications [9]. Besides the increase in the size of the search databases itself, ongoing efforts attempt to improve the quality of the scoring functions used to compare an experimental spectrum to a single theoretical spectrum. Typically this comes at the cost of increasing the time of a comparison. Examples of improvements are the prediction of peak intensities in theoretical spectra [3,5] or the explicit consideration

T. Elomaa et al. (Eds.): Ukkonen Festschrift, LNCS 6060, pp. 62–76, 2010.

of peptide modifications in tandem mass spectrometry. As a consequence of the above, many large-scale proteomics efforts currently face the problem that the database searching takes much longer time than the experimental generation of data, making unbiased database search a bottleneck or impossibility in current proteomics pipelines, and interfering with the application of new, sophisticated scoring schemes.

One popular approach to speeding up database searching is to first employ a simple scoring function to filter away spectra that do not score high enough to be considered as true matches. As this step typically allows to quickly exclude most candidate spectra, more sophisticated scoring functions can be applied on the remaining, small, set of spectra. Other approaches attempt to avoid the linear sweep through the database altogether. One such approach is based on a standard method (MVP-tree) for accelerating $k$-nearest neighbor search in metric spaces [14]. However, the high dimensionality of the spectra significantly limits how much speedup can be achieved by this type of approach. Another approach is based on local sensitivity hashing to obtain fast search times despite the high dimensionality [2]. A drawback of local sensitivity hashing is a non-zero probability that some spectra might be overlooked even though they are within the chosen threshold range of the query spectrum. A third, heuristic approach is based on sequence tags, short sequences of consecutive peaks, and filtering away all spectra that do not fit these tags [4,10].

The approach presented here is based on the identification of high-scoring spectra according to the simple similarity measures *shared peaks count* (SPC) and its extension *shared peaks intensity* (SPI). Since the same database of theoretical spectra is typically used for many searches, the database can be preprocessed and stored in a data structure that enables faster searching. While in bioinformatics, index-based search has extensively been studied in the context of string pattern matching [12,15,16], we are not aware of any such approaches in the context of searching a mass spectrometry database.

In Section 2 we give a formal definition of the search problem, called SPC Range Search Problem, which we consider throughout most of this paper. In Sections 3, 4 and 5 we introduce three algorithms. Section 6 discusses extensions of the basic problem and how our algorithms can be adapted. In Section 7 we present empirical tests of the algorithms on realistic peptide mass fingerprinting data. Section 8 concludes.

## 2 Problem Definition

In the following, a mass spectrum is represented as a set of integer $m/z$ values in the range $\{1, \ldots, N\}$. A simple similarity measure between two spectra is the *shared peaks count* (SPC), the number of $m/z$ values that two spectra have in common.

*Problem 1 (SPC Range Search Problem).* Given a set $D = \{T_1, \ldots, T_n\}$ of theoretical spectra $T_i \subseteq \{1, \ldots, N\}$ and a query spectrum $Q \subseteq \{1, \ldots, N\}$, find all the spectra in $D$ that have at least $k$ peaks in common with $Q$, i.e. identify the

set $\{i : SPC(T_i, Q) \geq k\}$ where $SPC(S, Q) := |S \cap Q|$ is the *shared peaks count* of sets $S$ and $Q$.

Let $m$ denote the total number of peaks in all the spectra in the database, i.e. $m = \sum_{i=1}^{n} |T_i|$. If we assume that $D$ and $Q$ are given as sorted lists, then a straightforward algorithm for solving this problem would take $O(\sum_{i=1}^{n}(|T_i| + |Q|)) = O(m + n \cdot |Q|)$ time. However, if we are allowed to build more complex data structures storing $D$, faster query times are possible. In the following we disregard the preprocessing time needed to build the data structure, as long as it is polynomial, and mainly consider the query times that can be achieved once the data structure is built.

The above formal problem can be applied to the approaches of peptide mass fingerprinting and tandem mass spectrometry. In practice, $N$ is determined by the limited $m/z$ range and the resolution of the instrument used.

## 3 Lookup Algorithm

A simple data structure for speeding up the query time is an array, $A$, that maps each integer in the range $\{1, \ldots, N\}$ to a list of the spectra in $D$ that contains the integer in question. While considering the elements of $Q$ one after the other, another array, $B$, of length $n$ can be used to accumulate the shared peaks count for each of the $n$ spectra. Algorithm 1 shows pseudocode for this algorithm.

### 3.1 Analysis of Lookup Algorithm

The lookup algorithm assumes its worst case running time if all the peaks of $Q$ occur in all the spectra in the database. In this case the running time is $O(n \cdot |Q|)$. In order to give a better time analysis than this, we have to include some knowledge about the distribution of masses in the spectra in the database so that we know that not all the spectra in the database are expected to be in the result set. Counting the number of times each of the spectra occurs when we look up all

**Algorithm 1.** (Lookup SPC)

**Input:** array $A$ where $A[x] = \{i \mid x \in T_i\}$ for all $x \in \{1, \ldots, N\}$,
array $B$ where $B[i] = 0$ for all $i \in \{1, \ldots, n\}$
**Output:** set of indices $I = \{i : |T_i \cap Q| \geq k\}$

```
1: I ← ∅
2: for all x ∈ Q do
3:    for all y ∈ A[x] do
4:       B[y] ← B[y] + 1
5:       if B[y] = k then
6:          I.add(y)
7:       end if
8:    end for
9: end for
```

the peaks in the query spectrum means that we will never look at the spectra that have no peaks in common with the query spectrum. This should give a substantial speed-up since we expect $|Q|$ to be much smaller than $N$. If we assume that none of the $N$ possible peaks is contained in more than $P \cdot n$ spectra, the running time becomes $O(P \cdot n \cdot |Q|)$ which is sub-linear in $n$ since $P$ is a fraction between zero and one. The best case would occur if the peaks in the spectra are uniformly distributed over the range $\{1, \ldots, N\}$. Then the expected value of $P$ would be $\frac{m}{N \cdot n}$, but that is not realistic since we expect to see more small masses than large masses in a spectrum. In experiments with a realistic peptide mass fingerprinting database constructed from a list of all human proteins (see also Section 7) we have measured the value of $P$ to be 0.06 if we only look at masses over 500 Da and use a mass accuracy of 1 Da, see Fig. 1.

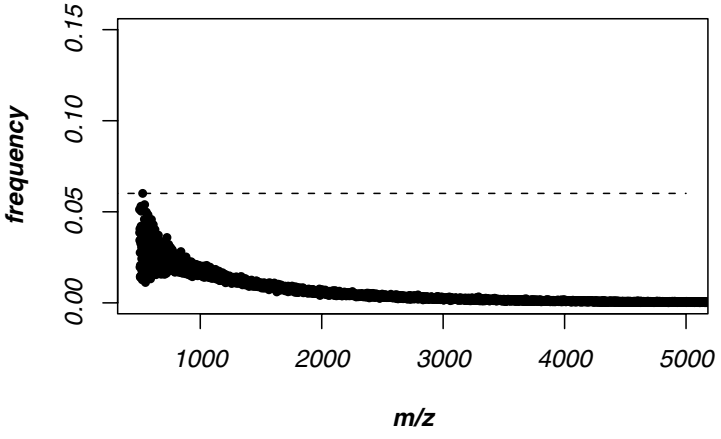



**Fig. 1.** Distribution of masses in theoretical tryptic digested spectra. Only masses above 500 Da are shown, bin-width is 1 Da. The horizontal line shows the value of $P$, the maximal frequency of any of the bins.

## 4 Folding Algorithm

Our second algorithm uses a mapping of the range of masses $\{1, \ldots, N\}$ into $N' \ll N$ bins. The mapping should be defined so that the probability that a spectrum contains a peak belonging to a certain bin should be approximately the same for all bins. One simple possibility that probably satisfies this property is by mapping mass $i$ to bin $H(i) := i \mod N'$.

In the following, let $V_S$ be a vector of length $N'$ so that $V_S[i]$ is the number of peaks in S that fall in the $i$th bin, $V_S[i] := |\{s \in S \mid H(s) = i\}|$. Given two spectra $S$ and $Q$, an element $i \in S \setminus Q$ contributes to $V_S[i]$, but not to $V_Q[i]$. Similarly, an element $i \in Q \setminus S$ contributes to $V_Q[i]$, but not to $V_S[i]$. Together, these form the symmetric difference of $S$ and $Q$, $S \triangle Q := (S \setminus Q) \cup (Q \setminus S)$. Summing over all these elements, we observe that the overall number of element-wise differences

**Algorithm 2.** (Folding SPC)

**Input:** $D = \{T_1, \ldots, T_n\}$, and $V_T$ for all $T \in D$
**Output:** set of indices $I = \{i : |T_i \cap Q| \geq k\}$

```
1: calculate V_Q
2: for i = 1 to n do
3:    if U(T_i, Q) ≥ k then
4:       if SPC(T_i, Q) ≥ k then
5:          I.add(i)
6:       end if
7:    end if
8: end for
```

in all bins, $\sum_{i=0}^{N'-1} |V_S[i] - V_Q[i]|$, is upper-bounded by the cardinality of $S \triangle Q$, $|S \triangle Q| = |S| + |Q| - 2SPC(S, Q)$. Thus, from the two vectors $V_S$ and $V_Q$ we can calculate an upper bound $U(S, Q)$ on the shared peaks count of these two spectra:

$$U(S,Q) \;:=\; \frac{|S| + |Q| - \sum_{i=0}^{N'-1} |V_S[i] - V_Q[i]|}{2} \;\geq\; SPC(S,Q).$$

The idea of the folding algorithm is to preprocess the database by storing for each spectrum $T_i \in D$ the corresponding vector $V_{T_i}$. Then, for a given query spectrum $Q$, the upper bounds $U(T_i, Q)$ are computed and subsequently the exact shared peaks count is computed only for those database entries $T_i$ whose bound was larger than $k$. Algorithm 2 shows pseudocode for this algorithm. A speed-up is achieved if $U$ can be computed faster than $SPC$ and only few computations of the actual shared peaks count are necessary.

### 4.1 Analysis of Folding Algorithm

Calculating all the upper bounds takes time $O(nN')$, but on top of that we need to calculate the actual shared peaks count of those spectra where the upper bounds were larger than $k$. To calculate the actual shared peaks count we just use the trivial algorithm for finding the intersection of two sorted lists which takes time proportional to the lengths of the two lists. The number of spectra for which we need to calculate the actual shared peaks count depends on $k$, $N'$ and $Q$. In the worst case we would need to calculate the shared peaks count of all the spectra, in which case the running time would be equal to the running time of the straightforward algorithm.

## 5 Clustering Algorithm

The third algorithm solves the SPC Range Search Problem by dividing the spectra (sets) in $D$ into disjoint subsets $C_1, \ldots, C_\ell$. The main idea is to perform a type of group testing: if the query set $Q$ has less than $k$ peaks in common with the union of the sets in a cluster $C_i$, then none of the spectra in the cluster is in the

**Algorithm 3.** (Clustering SPC)

**Input:** clustering $C = \{C_1, \ldots, C_\ell\}$
**Output:** set of indices $I = \{i : |T_i \cap Q| \geq k\}$

```
 1: I ← ∅
 2: for i = 1 to ℓ do
 3:    peaks ← ∅
 4:    for all x ∈ Q do
 5:       if x ∈ ⋃_{j∈C_i} T_j then
 6:          peaks.add(x)
 7:       end if
 8:    end for
 9:    if peaks.size() ≥ k then
10:       for all j ∈ C_i do
11:          count ← 0
12:          for all x ∈ peaks do
13:             if x ∈ T_j then
14:                count ← count + 1
15:             end if
16:          end for
17:          if count ≥ k then
18:             I.add(j)
19:          end if
20:       end for
21:    end if
22: end for
```

solution. If however the intersection between $Q$ and the union of the spectra in the cluster is larger than or equal to $k$, then we need to compare all the spectra in the cluster with the intersection $I_i := Q \cap (\bigcup_{T \in C_i} T)$ in order to obtain $SPC(T, Q) = |T \cap Q| = |T \cap I_i|$. Algorithm 3 shows pseudocode for this algorithm.

The performance of the clustering strategy will depend on how good the clustering is. The number of spectra that end up in clusters that share $k$ peaks with the query spectrum $Q$ should be as small as possible, but at the same time we want there to be as few clusters as possible. The difficulty in making the clustering algorithm effective is in finding a good trade-off between the number of clusters and the probability of the query spectrum having many peaks in common with a cluster.

The probability of a peak from $Q$ belonging to the union of the sets in a cluster should be about the same for all clusters. If we assume that the peaks are uniformly distributed, then this means that the size of a cluster should be some constant factor of the number of possible peaks, i.e. $|\bigcup_{T \in C_i} T| \approx \delta N$ for all $i$. Finding a clustering of $D$ that satisfies this constraint using as few clusters as possible is an NP-hard problem known as the *Set-Bin-Packing Problem* [6]. For this reason we do not try to find such an optimal clustering but use a heuristic to create the clustering. Algorithm 4 shows pseudocode for an $O(nm)$ greedy heuristic for finding a clustering. The first spectrum in a cluster is picked

**Algorithm 4.** (Clustering Heuristic)

**Input:** $D = \{T_1, \ldots, T_n\}$, $\delta$, $N$
**Output:** clustering $C = \{C_1, \ldots, C_\ell\}$ of $D$

```
 1: i ← 1
 2: while D is not empty do
 3:    C_i ← ∅
 4:    union_i ← ∅
 5:    T ← D.pop()
 6:    C_i.add(T)
 7:    for all x ∈ T do
 8:       union_i.add(x)
 9:    end for
10:    newsize ← |union_i|
11:    while newsize ≤ δN do
12:       best ← D.first()
13:       newsize ← |union_i ∪ best|
14:       for all T ∈ D do
15:          if |union_i ∪ T| < newsize or (|union_i ∪ T| = newsize and |T| > |best|)
          then
16:             best ← T
17:             newsize ← |union_i ∪ best|
18:          end if
19:       end for
20:       if newsize ≤ δN then
21:          C_i.add(best)
22:          for all x ∈ best do
23:             union_i.add(x)
24:          end for
25:          D.remove(best)
26:       end if
27:    end while
28:    i ← i + 1
29: end while
```

at random, and afterward we repeatedly add to the cluster the spectrum that increases the union of the number of different peaks in the cluster the least, until the union reaches the limit. If several spectra increase the number of peaks in the cluster by the same amount we add the largest of these to the cluster. Spectra whose size exceeds the limit become singleton clusters.

### 5.1 Analysis of Clustering Algorithm

For each cluster $C_i$ it takes $O(|Q|)$ time to calculate the intersection $I_i$ if we store the peaks in the clusters in a bitvector so that looking up whether a peak is in the union of a cluster takes constant time. If the size of $I_i$ is $k$ or larger, then we need to use additional time $O(|I_i| \cdot |C_i|)$. This gives us an expected running time of $O(\ell \cdot |Q| + \sum_{i=1}^{\ell} Pr(|I_i| \geq k) \cdot |I_i| \cdot |C_i|)$. Since we are looking at the

expected running time, we can use the expected value $\frac{n}{\ell}$ instead of $|C_i|$ and in view of $|Q| \geq |I_i|$ we can write the expected running time as $O(\ell \cdot |Q| + Pr(|I_i| \geq k) \cdot |Q| \cdot n)$, because we expect $Pr(|I_i| \geq k)$ to be the same no matter what the value of $i$ is. This is always better than the straightforward algorithm since $\ell < n$.

The value of $Pr(|I_i| \geq k)$ depends very much on the value of $k$. If $k$ is large, the probability will be small and the running time will be dominated by the factor $\ell \cdot |Q|$. If on the other hand $k$ is small, then $Pr(|I_i| \geq k)$ will be large and $Pr(|I_i| \geq k) \cdot |Q| \cdot n$ will dominate the running time.

There is a trade-off between the value of $\ell$ and $Pr(|I_i| \geq k)$, since making $\ell$ larger would make the clusters and thus $Pr(|I_i| \geq k)$ smaller. The actual correspondence between $Pr(|I_i| \geq k)$ and $\ell$ is difficult to calculate since it requires knowledge about the distribution of peaks in the spectra in the database and the query spectrum.

### 5.2 Recursive Clustering

The clustering idea can be applied recursively to yield a hierarchical clustering. The probability of a peak belonging to a cluster should be the same for all of the clusters on the same level, but it should be smaller for lower levels than for higher levels. We can still use Algorithm 4 to make the clustering, now we just need to apply it recursively to the clusters with a smaller value of $\delta$. The recursive clustering stops when we reach a certain level or if the clusters contain only a few peaks. For a cluster that is a sub-cluster of another cluster we do not need to remember which of all $N$ possible peaks are in the cluster, but only which of the peaks in the super-cluster are in the cluster. This allows us to save some space. For a cluster $C$ let $C$.subclusters be a list of subclusters of $C$ and let $C$.rank$[i]$ be $x$ if the $i$th peak of the super-cluster is in the cluster and there are $x - 1$ peaks in the cluster that have got smaller masses. If the $i$th peak of the super-cluster is not in the cluster then $C$.rank$[i]$ should be $-1$. Algorithm 5 shows pseudocode for a recursive algorithm to search a hierarchical clustering.

## 6 Extensions

### 6.1 Shared Peaks Intensities

Apart from their $m/z$ value, peaks in a mass spectrum also have an intensity. Since there is a higher risk that a peak with small intensity does not come from an actual protein fragment, but is just due to random noise, the high intensity peaks should be trusted more than the low intensity peaks. One way of giving more value to high intensity peaks is by using the *shared peaks intensity* (SPI) as similarity measure. The shared peaks intensity of two spectra is the sum of the intensities of all the peaks they have in common. The shared peaks count problem of the previous sections can be seen as a special case of the shared peaks intensity problem where all peaks have intensity one.

**Algorithm 5.** (Recursive Clustering SPC)

**Input:** cluster $C$, list of peaks *peaks*
**Output:** $I = \{i : |T_i \cap Q| \geq k\}$

```
 1: I ← ∅
 2: newPeaks ← ∅
 3: for all x ∈ peaks do
 4:    if C.rank[x] ≠ −1 then
 5:       newPeaks.append(C.rank[x])
 6:    end if
 7: end for
 8: if newPeaks.size() ≥ k then
 9:    if C contains only a single spectrum T_j then
10:       I.add(j)
11:    else
12:       for all y ∈ C.subclusters do
13:          make recursive call with y and newPeaks
14:       end for
15:    end if
16: end if
```

*Problem 2 (SPI Range Search Problem).* Given a set $D = \{T_1, \ldots, T_n\}$ of theoretical spectra $T_i \subseteq \{1, \ldots, N\}$ and a query spectrum $Q = \{q_1, \ldots, q_w\} \subseteq \{1, \ldots, N\}$ and their corresponding intensities $I(q_1), \ldots, I(q_w)$, find all the spectra in $D$ where the sum of the intensities of the peaks they have in common with $Q$ is at least a fraction $p$ of the total intensity of $Q$, i.e. identify the set $\{i : SPI(T_i, Q) \geq p\}$ where $SPI(S, Q) := \sum_{q \in S \cap Q} I(q) / \sum_{q \in Q} I(q)$ is the *shared peaks intensity* of sets $S$ and $Q$.

The lookup and clustering algorithms of the previous sections can easily be extended to address this problem instead of the SPC problem. The only change is that instead of counting we now need to sum the intensities.

### 6.2 Mapping Peaks to Integers

Our algorithms assume that the $m/z$ values are integers even though the values actually produced by the mass spectrometer are not integers. For this reason we need to map the real values to integers. A mass spectrometer will have an associated accuracy stating how much deviation between the real mass and the measured mass can be expected. If we know that there is only a small risk that the measured value deviates more than $\epsilon$ from the real value we can map the value $x$ to an integer by first dividing by a value larger than $2\epsilon$ and then rounding down: $x \to \lfloor \frac{x}{r\epsilon} \rfloor$ where $r > 2$. This, however, means that two values less than $\epsilon$ apart can be mapped to different integers. One way of addressing this problem is by mapping the spectra in the database as described above, but mapping the query spectrum so that if there are two integer values that are within $\frac{1}{r}$ of $\frac{x}{r\epsilon}$, then both of these integers are in the query set. Making the integer conversion this way means that one might get a higher shared peaks count than one would

get using a more precise alignment between spectra. This is, however, not a big problem since we intend to use our method as a fast filtering where a reasonable number of false positives is acceptable, but false negatives are not. The mapping described above might double the size of the query spectrum which of course affects the running times, but it ensures that there will be no false negatives.

### 6.3 Tandem Mass Spectrometry

The algorithms could also be used on tandem mass spectrometry data. In tandem mass spectrometry, the query spectra come from fragmented "parent" peptides, whose mass is usually known. This additional information means that, in the case of tandem mass spectrometry, the database does not need to be preprocessed as a whole, but independent index structures can be made for all different parent masses (the masses are rounded to integers). Even though this means that the spectra in the databases are distributed on many separate data structures, it does not mean that the data structures are necessarily small.

In tandem mass spectrometry one often wants to consider not just one spectrum for each peptide but also take into account the possibility that a peptide could have been modified by, for example, phosphorylation. The so called *virtual database* solution of searching for peptide modifications means that, for each possible position of all interesting modifications, a new spectrum is added to the database, resulting in a large increase of the database size. Hopefully the algorithms presented here help making this virtual database approach more feasible.

## 7 Experiments

We have implemented the following four algorithms in C++ in order to evaluate their search times in a comparative setting:

**Simple:** The straightforward algorithm that scans the database linearly for each query spectrum.

**Lookup:** The algorithm that stores for each mass a list of the spectra that contain this mass (Algorithm 1).

**Folding:** The algorithm that maps the mass range $\{1, \ldots, N\}$ into $N' \ll N$ bins (Algorithm 2).

**Cluster:** The algorithm that divides the spectra into disjoint subsets and then searches these subsets recursively (Algorithms 3–5).

In order to test the running times on realistic peptide mass fingerprint (PMF) data, we created a PMF database from a list of all human proteins obtained from The International Protein Index [8]. From each protein we generated a theoretical spectrum by simulating a tryptic digest of the protein. Trypsin is an enzyme that cleaves the protein after each occurrence of the amino acids lysine or arginine, except if the next amino acid is proline. So for the generation of theoretical spectra we first split the proteins into substrings based on the just mentioned rule and then calculated the masses of these substrings by summing the masses of their amino acids and then converting them to integer values.

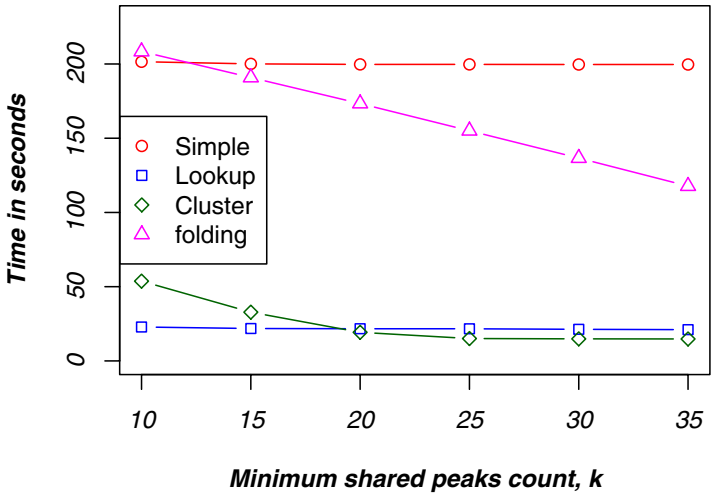



**Fig. 2.** Experimental results on simulated PMF data. The vertical axis shows the time in seconds it took to perform 1000 queries. The horizontal axis shows different values of the minimum shared peaks count $k$. The clustering algorithm used a recursive clustering with three levels where the $\delta$ parameters for the three levels were 0.05, 0.02 and 0.01.

We have simulated query spectra by randomly drawing a specific number of different peaks between 400 and 5000 Da using the distribution of peaks that we observed in the database. Figure 2 shows query times for searching 1000 different query spectra with 50 peaks each for different values of the threshold parameter $k$. It can be seen that while the time usage of the simple algorithm and the lookup algorithm is almost unaffected by changes in $k$, the time usage of the cluster algorithm does depend on this parameter. The lookup algorithm is always faster than the simple algorithm and for small values of $k$ it also beats the clustering algorithm, but for larger values of $k$ the clustering algorithm is the fastest.

Figure 3 compares the time usage for the clustering algorithm using clusterings built with different values of $\delta$ and a version with recursive clustering (parameters are given in the figure caption). The results show that for a given $k$ the recursive clustering algorithm is not much better than the best of the single level clusterings, but the main effect of the recursive clustering is that the curve is flatter so that the same data structure is good for more values of $k$.

The International Protein Index we used contains 69164 human proteins. To see how the size of the database affects the running times of the algorithms we also tried creating some smaller databases by only using a fraction of the 69164 proteins and some larger ones by including some extra spectra that were generated by adding a small value to all the peaks of an existing spectrum. Figure 4 shows running times for varying database sizes when searching for query spectra with 50 peaks and a threshold of $k = 15$. It is interesting to note that while the clustering

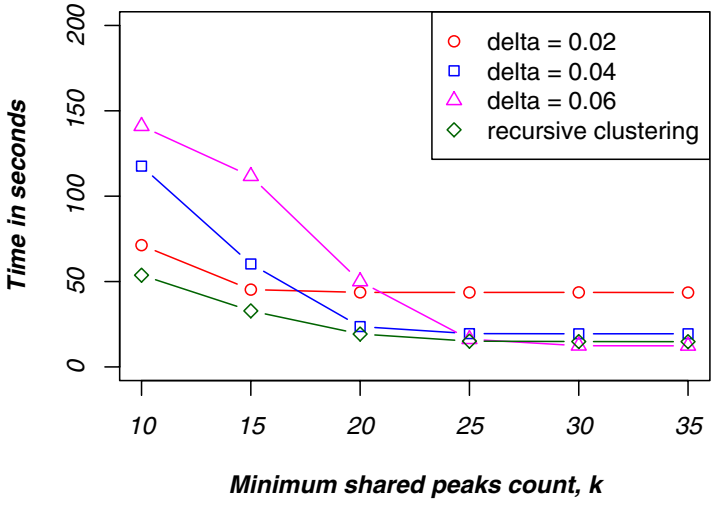



**Fig. 3.** Comparison of the clustering algorithm with different values of the $\delta$ parameter on simulated PMF data. The vertical axis shows the time in seconds it took to perform 1000 queries. The recursive clustering had three levels and the $\delta$ parameters for the three levels were 0.05, 0.02 and 0.01.

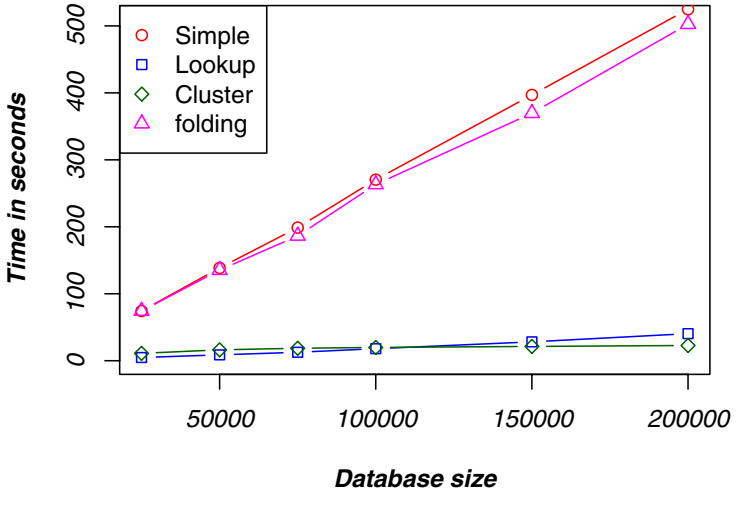



**Fig. 4.** Experimental results of SPC algorithms on simulated PMF data with varying database size. The vertical axis shows the time in seconds it took to perform 1000 queries.

algorithm takes twice the time of the lookup algorithm for $n = 25000$ it only takes half the time for $n = 200000$. This means that the clustering algorithm scales better with database size than the lookup algorithm does.

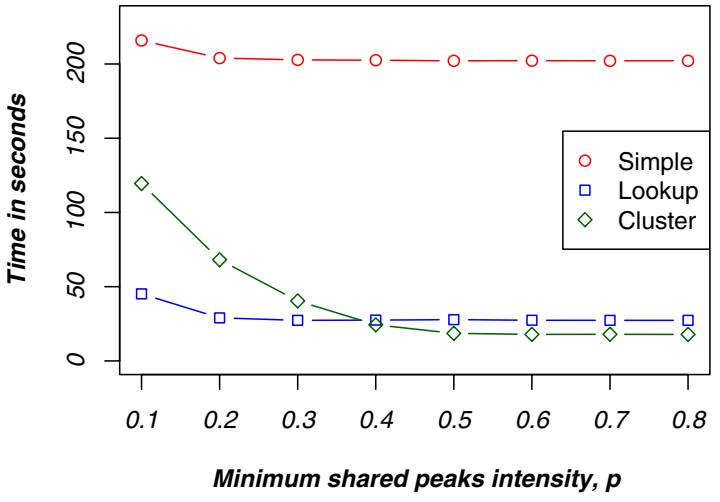



**Fig. 5.** Experimental results of SPI algorithms on simulated PMF data. The vertical axis shows the time in seconds it took to perform 1000 queries.

We have also tested the algorithms on the shared peaks intensity measure by giving the peaks in the query spectra intensities picked randomly between 0 and 1 from a uniform distribution. The results can be seen in Fig. 5. They are similar to the corresponding results for SPC.

## 8 Conclusion

We have developed three algorithms for searching in an indexed mass spectrometry database using the simple shared peaks count and shared peaks intensity similarity measures. The algorithms can be used to filter potential candidates in a database before ranking them using a more sophisticated scoring thus reducing the overall time of the search. Speeding up database searching is becoming increasingly important due to growing databases and faster data generation.

It is hard to make a good general analysis of the presented algorithms since the running times depend on the distribution of peaks in the spectra in the database and in the query spectrum, and these differ between different databases and different mass spectrometry equipment. So we can not give solid theoretical evidence that our algorithms will always be much faster than the trivial algorithm, but our experiments show that on realistic data our algorithms do give a significant speed-up.

A direction to be explored in the future might be the combination of different of our algorithms. In particular, the folding algorithm and the clustering algorithm are somewhat complementary, such that a hybrid might provide an additional speed-up.

## References


1. Aebersold, R., Mann, M.: Mass spectrometry-based proteomics. Nature 422(6928), 198–207 (2003)
2. Dutta, D., Chen, T.: Speeding up tandem mass spectrometry database search: metric embeddings and fast near neighbor search. Bioinformatics 23(5), 612–618 (2007)
3. Elias, J.E., Gibbons, F.D., King, O.D., Roth, F.P., Gygi, S.P.: Intensity-based protein identification by machine learning from a library of tandem mass spectra. Nat. Biotechnol. 22(2), 214–219 (2004)
4. Frank, A., Tanner, S., Pevzner, P.A.: Peptide sequence tags for fast database search in mass-spectrometry. In: Miyano, S., Mesirov, J., Kasif, S., Istrail, S., Pevzner, P.A., Waterman, M. (eds.) RECOMB 2005. LNCS (LNBI), vol. 3500, pp. 326–341. Springer, Heidelberg (2005)
5. Havilio, M., Haddad, Y., Smilansky, Z.: Intensity-based statistical scorer for tandem mass spectrometry. Anal. Chem. 75(3), 435–444 (2003)
6. Izumi, T., Yokomaru, T., Takahashi, A., Kajitani, Y.: Computational complexity analysis of set-bin-packing problem. IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences E81-A(5), 842–849 (1998)
7. Johnson, J.M., Castle, J., Garrett-Engele, P., Kan, Z., Loerch, P.M., Armour, C.D., Santos, R., Schadt, E.E., Stoughton, R., Shoemaker, D.D.: Genome-wide survey of human alternative pre-mRNA splicing with exon junction microarrays. Science 302(5653), 2141–2144 (2003)
8. Kersey, P.J., Duarte, J., Williams, A., Karavidopoulou, Y., Birney, E., Apweiler, R.: The international protein index: An integrated database for proteomics experiments. proteomics 4(7), 1985–1988 (2004)
9. Mann, M., Jensen, O.N.: Proteomic analysis of post-translational modifications. Nature Biotechnol. 21(3), 255–261 (2003)
10. Mann, M., Wilm, M.: Error-tolerant identification of peptides in sequence databases by peptide sequence tags. Anal. Chem. 66(24), 4390–4399 (1994)
11. Margulies, M., Egholm, M., Altman, W.E., Attiya, S., Bader, J.S., Bemben, L.A., Berka, J., Braverman, M.S., Chen, Y.J., Chen, Z., Dewell, S.B., Du, L., Fierro, J.M., Gomes, X.V., Godwin, B.C., He, W., Helgesen, S., Ho, C.H., Irzyk, G.P., Jando, S.C., Alenquer, M.L., Jarvie, T.P., Jirage, K.B., Kim, J.B., Knight, J.R., Lanza, J.R., Leamon, J.H., Lefkowitz, S.M., Lei, M., Li, J., Lohman, K.L., Lu, H., Makhijani, V.B., McDade, K.E., McKenna, M.P., Myers, E.W., Nickerson, E., Nobile, J.R., Plant, R., Puc, B.P., Ronan, M.T., Roth, G.T., Sarkis, G.J., Simons, J.F., Simpson, J.W., Srinivasan, M., Tartaro, K.R., Tomasz, A., Vogt, K.A., Volkmer, G.A., Wang, S.H., Wang, Y., Weiner, M.P., Yu, P., Begley, R.F., Rothberg, J.M.: Genome sequencing in microfabricated high-density picolitre reactors. Nature 437(7057), 376–380 (2005)
12. McCreight, E.M.: A space-economical suffix tree construction algorithm. J. ACM 23(2), 262–272 (1976)
13. Palagi, P.M., Hernandez, P., Walther, D., Appel, R.D.: Proteome informatics I: Bioinformatics tools for processing experimental data. Proteomics 6(20), 5435–5444 (2006)
14. Ramakrishnan, S.R., Mao, R., Nakorchevskiy, A.A., Prince, J.T., Willard, W.S., Xu, W., Marcotte, E.M., Miranker, D.P.: A fast coarse filtering method for peptide identification by mass spectrometry. Bioinformatics 22(12), 1524–1531 (2006)

15. Ukkonen, E.: On-line construction of suffix trees. Algorithmica 14(3), 249–260 (1995)
16. Weiner, P.: Linear pattern matching algorithms. In: Proceedings of the 14th Annual IEEE Symposium on Switching and Automata Theory, pp. 1–11. IEEE Press, Los Alamitos (1973)
17. Whitfield, E.J., Pruess, M., Apweiler, R.: Bioinformatics database infrastructure for biotechnology research. J. Biotechnol. 124(4), 629–639 (2006)

# Extended Compact Web Graph Representations

Francisco Claude[1,⋆] and Gonzalo Navarro[2,⋆⋆]

[1] David R. Cheriton School of Computer Science, University of Waterloo
fclaude@cs.uwaterloo.ca
[2] Department of Computer Science, University of Chile
gnavarro@dcc.uchile.cl

**Abstract.** Many relevant Web mining tasks translate into classical algorithms on the Web graph. Compact Web graph representations allow running these tasks on larger graphs within main memory. These representations at least provide fast navigation (to the neighbors of a node), yet more sophisticated operations are desirable for several Web analyses.

We present a compact Web graph representation that, in addition, supports reverse navigation (to the nodes pointing to the given one). The standard approach to achieve this is to represent the graph and its transpose, which basically doubles the space requirement. Our structure, instead, represents the adjacency list using a compact sequence representation that allows finding the positions where a given node $v$ is mentioned, and answers reverse navigation using that primitive. This is combined with a previous proposal based on grammar compression of the adjacency list. The combination yields interesting algorithmic problems. As a result, we achieve the smallest graph representation reported in the literature that supports direct and reverse navigation, and also obtain other variants that occupy relevant niches in the space/time tradeoff.

## 1 Introduction and Related Work

The Web can be modeled as a directed graph: every page corresponds to a node and every link between two pages is represented as a directed edge between the corresponding nodes. This so-called "Web graph" contains an enormous amount of useful information, which is used for a wealth of purposes, from technical (such as improving search engines) to economic (such as detecting potential customers) to scientific (such as carrying out sociological studies).

Methods to discover Web communities, Web spam, Web structure, hubs and authorities, and many others, rely on classical graph algorithms. Donato et al. [16] show how several common Web mining techniques used to discover the structure and evolution of the Web graph build on classical graph algorithms such as depth- and breadth-first-search, reachability, and weakly and strongly connected components. Saito et al. [28] presents a technique for Web spam detection that boils down to algorithms for finding strongly connected components,

⋆ Funded by NSERC of Canada and Go-Bell Scholarships Program.
⋆⋆ Funded in part by Fondecyt Grant 1-080019, and by Millennium Institute for Cell Dynamics and Biotechnology, Grant ICM P05-001-F, Mideplan, Chile.

T. Elomaa et al. (Eds.): Ukkonen Festschrift, LNCS 6060, pp. 77–91, 2010.

for clique enumeration, and for minimum cuts. A simple representation that allows *direct* navigation (to the nodes pointed from the current one) suffices for these purposes. Yet, there are other important applications where efficient *reverse* navigation (to the nodes pointing to the current one) is also necessary. The HITS algorithm [23] to find hubs and authorities on the Web starts by selecting random pages and finding the induced subgraphs, which are the pages that point to or are pointed from the selected pages [22]. Uniform sampling methods [27] also require direct and reverse navigation, and are usually replaced by suboptimal alternatives due to the difficulty of implementing the latter.

An important limitation when processing this kind of graphs is their size. Many of the graph algorithms we mentioned are not disk-friendly, and thus the sizes of the graphs that can be analyzed by the Web mining applications, and consequently the quality of their results, is limited by the main memory size. Much effort has been spent in representing Web graphs in compressed form, so that the direct neighbors of a node can be efficiently retrieved [10,1,30,8,13]. Boldi and Vigna [8] achieve currently the least space combined with efficient navigation. In later work [13,14] we introduced the use of grammar compression of the adjacency lists (more precisely, Re-Pair [24]). This required more space than the best achievable by Boldi and Vigna, but when both methods used the same space, ours was faster. Other techniques [4], instead, achieve even less space than Boldi and Vigna, yet with much higher access times.

By essentially doubling the space of these solutions, one can represent the graph and its transpose, thus providing reverse navigation as well. Needless to say, adding such an amount of redundancy is against the goal of providing a compact representation. The only approach we know of where direct and reverse navigation is supported [9], the $k^2$-tree, is based on representing the adjacency matrix in a way that takes advantage of its sparseness. They achieve similar times for direct and reverse queries.

In this work we introduce an alternative way of supporting direct and reverse navigation. Let $G = (V, E)$ be our graph, where $n = |V|$ and $m = |E|$. We resort to previous work by regarding $G$ as a *binary relation* on $V \times V$, and then use the techniques of Barbay et al. [5], where forward and reverse traversal operations can be solved in time $O(\log \log n)$ per node delivered. A more recent followup [6] retains those times and reduces the space to the *worst-case* entropy of the binary relation, that is, $\log \binom{n^2}{m}$ (our logarithms are in base 2).

This worst-case compression, however, is poor for Web graphs, as these are far from random in the Erdõs-Rényi sense [17]. To illustrate this, we downloaded four Web crawls from the WebGraph project[1], which will be used for the experiments along the article. Table 1 shows their main characteristics. The third column shows the size (in bits per edge, bpe) required by a plain adjacency list representation using 4-byte integers. The fourth column shows the space required for a plain representation of the graph plus its transpose. The fifth column shows the lower bound given by the worst-case entropy, and the last column

[1] `http://law.dsi.unimi.it`

**Table 1.** Some characteristics of the four crawls used in our experiments, as well as expected space usage (in bpe) with some known methods

| Crawl | Nodes | Edges | Plain | 2×Plain | Bin.Rel. | Re-Pair |
|---|---|---|---|---|---|---|
| EU | 862,664 | 19,235,140 | 20.81 | 41.62 | 15.25 | 7.65 |
| Indochina | 7,414,866 | 194,109,311 | 23.73 | 47.46 | 17.95 | 4.54 |
| UK (2002) | 18,520,486 | 298,113,762 | 25.89 | 51.78 | 20.13 | 7.50 |
| Arabic | 22,744,080 | 639,999,458 | 25.51 | 51.02 | 19.66 | 5.53 |

the space actually achieved by the best method based on Re-Pair compression [13] for the direct plus the transposed graph. This shows that the worst-case entropy measure is a poor estimation of the compression that can be achieved.

In this article we combine our previous technique based on Re-Pair [13], which has been successful to compress Web graphs while supporting direct navigation, with the binary relation idea [5]. The latter boils down to representing the adjacency lists using a sequence representation that allows finding the occurrences of a symbol (that is, the places where a given node $v$ is mentioned in some list), and then find the reverse neighbors using this primitive. The combination with grammar compression poses some interesting algorithmic problems, however, because the compressed text is a sequence of terminals and nonterminals and thus the technique cannot be directly applied. The result achieves forward and reverse navigation within competitive times, and significantly less space, than representing the direct plus the transposed graph using previous techniques. Depending on the compact data structure we use to represent the sequence, we obtain, on one hand, the smallest reported space for a structure that supports bidirectional navigation (indeed, within $O(\log n)$ time per delivered neighbor); and on the other, a faster ($O(\log \log n)$ time) and larger data structure that occupies a relevant niche in the space/time tradeoff of the current state of the art.

## 2 Basic Concepts

### 2.1 Compact Data Structures for Sequences

A compact data structure aims at representing the same data as its classical counterpart in little space, while still supporting interesting queries without expanding the whole data structure. Sometimes compact data structures require more time per query in theory, but since they use less space, they can fit in smaller and faster memories. This important advantage allows them to outperform their classical counterparts, especially if we consider the scenario where the classical version of the data structure has to resort to disk, while the compact data structure fits in main memory.

A basic tool used in many compact data structures is the bitmap with *rank* and *select* capabilities [25]. Consider a binary string $B[1, n]$ We support the following queries:

- $rank_B(b, i)$: counts how many times the bit $b$ appears in the prefix $B[1, i]$.
- $select_B(b, j)$: returns the position of the $j$-th occurrence of bit $b$ in $B$.
- $access_B(i)$: retrieves $B[i]$.

Clark [12] proposed a solution that achieves constant time for the queries and requires $n + o(n)$ bits. This was later improved by Raman, Raman and Rao (RRR) [26]. They achieved constant time for the queries while using $nH_0(B) + o(n)$ bits, where $H_0$ represents the zero-order entropy. The zero-order (empirical) entropy of a sequence $S$, drawn from an alphabet $\Sigma$ of size $\sigma$, is defined as $H_0(S) = -\sum_{c \in \Sigma} p_c \log p_c \leq \log \sigma$, where $p_c = n_c/n$ and $n_c$ is the number of occurrences of character $c$ in $S$.

*Rank/select/access* queries naturally extend to sequences, where $b \in \Sigma$. A sequence representation supporting these primitives is the wavelet tree [20,25], which achieves $O(\log \sigma)$ time per query and requires $n \log \sigma + o(n) \log \sigma$ bits. The wavelet tree stores a number of bitmaps, which can be compressed using RRR, in which case the space requirement drops to $nH_0(S) + o(n) \log \sigma$ bits.

Golynski, Munro, and Rao (GMR) [18] presented another representation that achieves time $O(\log \log \sigma)$ for *rank* and *access*, and $O(1)$ for *select*. Alternatively, they can achieve $O(1)$ time for *access*, $O(\log \log \sigma)$ for *select*, and $O(\log \log \sigma \log \log \log \sigma)$ for *rank*. The structure requires $n \log \sigma + n\, o(\log \sigma)$ bits.

Note both structures replace the sequence. Claude and Navarro [15] carried out a practical evaluation of *rank/select/access* capable bitmap and sequence representations. They included a simplified version of GMR for the case $n \approx \sigma$, which we call *chunk*. The chunk proved to be very fast for large alphabets, while requiring little extra space on top of $n \log \sigma$. Wavelet trees, on the other hand, not only supported the three queries, but also were shown to be an interesting alternative for compressing sequences over large alphabets, where classical methods like Huffman fail due to the alphabet representation overhead. The paper also showed how to omit the pointers of the wavelet tree while preserving its time performance, which saves much space overhead on large alphabets.

### 2.2 Re-Pair Compression of Web Graphs

Re-Pair [24] is a grammar-based compression algorithm consisting of repeatedly finding the most frequent pair of symbols in a sequence of integers and replacing it with a new symbol, until no more replacements are convenient. Re-Pair works as follows over a sequence $L$: (1) It identifies the most frequent pair $ab$ in $L$. (2) It adds the rule $s \rightarrow ab$ to a dictionary $R$, where $s$ is a new symbol not appearing in $L$. (3) It replaces every occurrence of $ab$ in $L$ by $s$. (4) It iterates until the replacements do not compensate for the increase of $R$.

We call $C$ the sequence resulting from $L$ after compression. Every symbol in $C$ represents a *phrase* (a substring of $L$), which is of length 1 if it is an original symbol (called a *terminal*) or longer if it is an introduced one (a *nonterminal*). Any phrase can be recursively expanded in optimal time (that is, proportional to its length), even if $C$ is stored on secondary memory (as long as the dictionary of rules $R$ is kept in RAM).

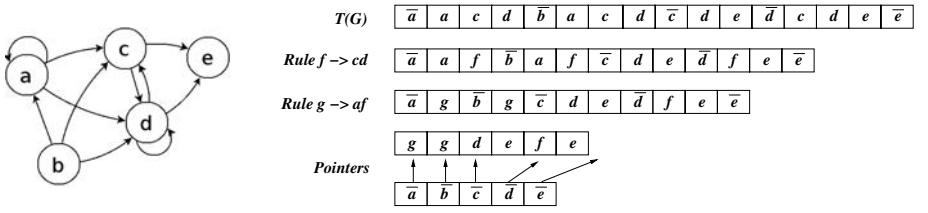



**Fig. 1.** A small example graph, its $T(G)$ representation, the Re-Pair compression of it, and the final replacement of the $\bar{v}_i$s by pointers

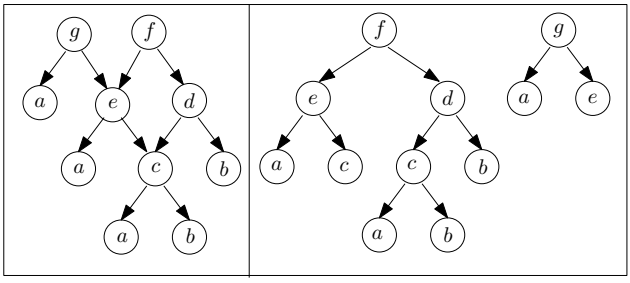
Re-Pair can be implemented in linear time [24]. However, this requires several data structures to track the pairs that must be replaced. This is problematic when applying it to large sequences. We developed an approximate version [13,14] that requires little space on top of the sequence.

In our proposal [13] to represent a Web graph $G$, each node $v$ has a special identifier $\bar{v}$ to mark the beginning of its adjacency list. The representation of the graph, $T(G)$, is the concatenation of the representations of all the adjacency lists, defined as $T(v_i) = \bar{v}_i v_{k_1} v_{k_2} \dots v_{k_r}$ where $v_{k_j}, 1 \leq j \leq r$, are the nodes pointed from $v_i$. Now $T(G)$ is compressed using Re-Pair. Since symbols $\bar{v}_i$ are unique, they stay as terminals in $C$. Therefore adjacency lists correspond to substrings in $C$, and thus can be decompressed in optimal time. The values $\bar{v}_i$ are afterwards removed from the sequence, and instead $n$ pointers to the beginning in $C$ of the list of each node is stored (in about the same space gained with the removal of the $\bar{v}_i$s). This allows direct navigation in optimal time, but not reverse navigation. Later [14], we proposed several variations achieving better space/time. Figure 1 illustrates the process for a small graph.

### 2.3 Representing the Re-Pair Rules

The dictionary $R$ can be represented as an array of pairs of integers, or in some compact form. We use a representation [19] that reduces it to about 50% while retaining efficient access to $R$. The set of rules can be seen as a directed acyclic graph of outdegree 2 where internal nodes are nonterminals and leaves are terminals. This is converted into a forest of binary trees, where nonterminal leaves signal shared subtrees. The forest is represented as a sequence $R_S$ of leaf values and a bitmap $R_B$ that defines the tree shape. Nonterminals are identified with the starting position of the (sub)tree that defines them in $R_B$. In $R_B$, the trees are described by a preorder traversal where a 1 represents an internal node (with 2 children) and a 0 represents a leaf. The (terminal or nonterminal) leaf value corresponding to $R_B[i] = 0$ can be found at $R_S[rank_{R_B}(0, i)]$.

For example, the set of rules $c \to ab$, $d \to cb$, $e \to ac$, $f \to ed$ and $g \to ae$, over terminals $\{a, b\}$, can be represented as shown in Figure 2. We have $R_B =$ `110011000100` and $R_S =$ `a6abba2`, where the '6' represents the nonterminal 'c', whose tree is at position 6 in $R_B$; similarly '2' represents 'e'.




| Position | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $R_B$ | = | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| $R_S$ | = | | | a | 6 | | | a | b | b | | a | 2 |

**Fig. 2.** Example of our representation of Re-Pair rules. Top left: the initial DAG. Top right: the forest representation. Bottom: Encoding with $R_B$ and $R_S$.

To expand a given nonterminal at position $i$ in $R_B$, we scan $R_B[i \ldots]$ until we have seen more 0s than 1s, and then collect all the consecutive leaf values. Leaf values corresponding to nonterminals must be recursively expanded.

## 3 A Simple Representation Based on Binary Relations

We note that sequence $T(G)$ (without Re-Pair compression), armed with symbol *rank* and *select* operations, is already able of handling an extended set of queries that includes reverse navigation, using an approach similar to Barbay et al.'s [5]. Assume we store a bitmap $B$ marking the positions of $T(G)$ where each adjacency list starts, more precisely, of the positions of the $\bar{v}_i$s: $start(v_i) = select_{T(G)}(\bar{v}_i, 1)$. We will also use operation $pred(i) = select_B(1, rank_B(1, i))$ that finds the last 1 up to position $i$ in $B$. We can support the following queries. Note $|T(G)| = n+m$.

- outdegree of $v_i$: it is $start(v_{i+1}) - start(v_i) - 1$.
- the $k$-th direct neighbor of $v_i$: it is $T[start(v_i) + k]$.
- indegree of $v_i$: it is $rank_{T(G)}(v_i, m+n)$, the number of times $v_i$ is mentioned in some adjacency list.
- the $k$-th reverse neighbor: $T[pred(select_{T(G)}(v_i, k))]$ gives the corresponding identifier $\bar{v}$ of the $k$-th reverse neighbor.
- edge $(v_i, v_j)$ exists: if $rank_{T(G)}(v_j, start(v_{i+1})) - rank_{T(G)}(v_j, start(v_i)) = 1$.

In practice we remove the $\bar{v}_i$s from $T(G)$ and set $n$ pointers $S[v_i]$ to the beginning of each list, so that $start(v_i) = S[v_i]$ and the $k$-th reverse neighbor becomes $rank_B(select_{T(G)}(v_i, k))$,[2] and the slow-in-practice [15] $select_B$ operation is totally avoided. Array $S$ requires $n \log m$ bits of space. Bitmap $B$ requires

[2] Nodes with zero outdegree must be handled somehow so that they do not interfere with $rank_B$, for example by marking them in another bitmap, or renumbering them after all the other nodes.

**Table 2.** Size required by our simple representation, using wavelet trees without pointers and RRR for the bitmaps [15], measured in bpe. The last column adds up the Re-Pair representations for the original and transposed graph.

| Crawl | Wavelet Tree | Plain | Bin.Rel. | Re-Pair |
|---|---|---|---|---|
| EU | 13.67 | 20.81 | 15.25 | 7.65 |
| Indochina | 14.16 | 23.73 | 17.95 | 4.54 |
| UK | 15.05 | 25.89 | 20.13 | 7.50 |
| Arabic | 15.30 | 25.51 | 19.66 | 5.53 |

at most $mH_0(B) + o(m) = n \log \frac{m}{n} + O(n) + o(m)$ bits using RRR (Section 2.1). The remaining $T(G)$ can be represented using GMR (Section 2.1), requiring $m \log n + m\, o(\log n)$ bits. The total space used is $m \log n + O(n \log m) + o(m \log n)$ bits. This is basically the space of a plain adjacency list representation, yet we have the extended functionality. On the other hand, the upper bound is higher than the $\log \binom{n^2}{m} = m \log \frac{n^2}{m} + O(m)$ worst-case entropy of the graph. Operations outdegree, indegree, and reverse neighbors are carried out in constant time per delivered datum[3], whereas checking existance of edges and retrieving direct neighbors cost $O(\log \log n)$.

With a wavelet tree representation, instead, every delivered direct or reverse neighbor (and checking edge existance) takes $O(\log n)$ time, but $T(G)$ can be compressed to $mH_0(T(G)) + o(m) \log n$ bits. Let $n_i$ be the indegree of node $v_i$, thus $v_i$ appears $n_i$ times in $T(G)$. Then $mH_0(T(G)) = \sum n_i \log \frac{m}{n_i}$. Web graphs are known to have varying indegrees: a Zipf-distribution with parameter $\theta = 2.1$ has been observed [2,10]. Under this distribution we obtain $mH_0(T(G)) = c \cdot m \log n + O(m)$, with $c = ((\theta - 1) \sum_{i \geq 1} i^{-\theta})^{-1} \approx 0.58$. On the other hand, $m/n$ is around 15–30 on Web graphs, so the worst-case entropy is $\log \binom{n^2}{m} = m \log n - O(m)$. Table 2 shows that this representation takes less space than a plain adjacency list (which does not answer reverse queries) on our four Web crawls, by a factor remarkably close to 0.58 (except on the smaller EU, where it is 0.65). It also takes less space than the worst-case graph entropy. Still, the last column reminds us that it is still far from the state of the art. In the next section we will combine this idea with Re-Pair compression.

## 4 Combining Re-Pair with Binary Relations

The result of the previous section makes it clear that we cannot go too far with zero-order compression of $T(G)$ or the graph binary relation. Re-Pair is much more successful. In fact, Re-Pair compression on graphs can be regarded as (and attribute its success to) the decomposition of the graph binary relation into two:

- Nodes are related to the Re-Pair symbols that conform their (compressed) adjacency list.
- Re-Pair symbols are related to the graph nodes they expand to.

[3] For indegree one needs to use other $n \log m$ bits, otherwise it costs $O(\log \log n)$.

The regularities exposed by this factorization go well beyond those captured by the worst-case entropy of the original binary relation or zero-order entropy of its sequence representation. In very broad terms, we attempt at representing the graph as the composition of these two binary relations. Using the technique of Barbay et al. [5], each direct neighbor would be retrieved in time $O(\log \log n)$, by finding all the Re-Pair symbols that conform its adjacency list (first relation) and then the graph nodes each such symbol expands to (second relation).

Finding the reverse neighbors of node $v$, on the other hand, is harder. We should first find all the Re-Pair symbols (nonterminals) that expand to $v$ (second relation), and then, for each such symbol, all the nodes in which adjacency list the symbol participates (first relation). The problem is that many nonterminals exist in the dictionary for the sake of structuring the grammar but do not appear in $C$, and thus we can carry out much work that does not lead to any result.

A further challenge is that representing the second binary relation as such, with the rules in fully expanded form, could require space $\omega(|R|)$. Thus we must use the representation of Section 2.3 for the second binary relation, and this complicates the operations we must carry out on it. We describe now our solution.

### 4.1 Representation

We apply our simple sequence representation of Section 3 on top of the Re-Pair compressed $T(G)$, instead of on the plain sequence. We compress $T(G)$ and represent sequences $C$ and $R_S$ not in plain form, but instead using a *rank*/*select*/*access* capable representation (see Section 2.1). This can be either:

- The GMR representation. It does not compress $C$ or $R_S$ any further, but it provides *access* and symbol *rank* in time $O(\log \log n)$ (yet typically constant), and symbol *select* in constant time.
- A wavelet tree. The operations are carried out in $O(\log n)$ time, but the representation compresses further $T(G)$ up to its zero-order entropy. Albeit significantly slower, this achieves unprecedented space results, as we see later.

Extraction of the direct neighbors is done exactly as in previous work [13], using *access* on the sequences $C$ and $R_S$ to expand the list of the desired node.

### 4.2 Extracting Reverse Neighbors

As explained, to find reverse neighbors of $v$ we must consider that $v$ may appear not only explicitly in $C$, but also implicitly, in the form of a nonterminal that expands to $v$. Given our representation of the rules $R$, we must look for $v$ in $R_S$ and, for each occurrence, collect all of its ancestors in $R_B$, and look for each of them in $C$. Because each such ancestor might appear in $R_S$ again (due to the conversion of the DAG into a forest) and have further ancestors, the process has to be repeated recursively for every ancestor found.

The occurrences of $v_i$ (or, recursively, any other nonterminal) in $R_S$ are obtained using $select_{R_S}(v_i, k)$. In order to extract the ancestors, we can use an alternative representation for $R_B$ called LOUDS [21]. LOUDS represents each leaf

```
rev-adj(v)
1.  For k ← 1 to rank_C(v, |C|) Do
2.      occ ← select_C(v, k)
3.      report rank_B(1, occ)
4.  For k ← 1 to rank_{R_S}(v, |R_S|) Do
5.      occ ← select_{R_S}(v, k)
6.      For each s ancestor of R_S[occ] in R_S Do
7.          rev-adj(s)
```

**Fig. 3.** Obtaining the reverse adjacency list

with a 0 and has been shown to be very effective when only parent/child traversals are required [3], which makes it ideal for our purpose. We adapt LOUDS to binary forests as follows. Let $f$ be the number of trees in the dictionary forest. The forest is traversed level-wise and left-to-right within each level, and for each node found we write a 1 if the node has (two) children and a 0 if not. In Figure 2, $f = 2$ and $R_B = 111100001000$. Each node is identified with its corresponding bit position $i \geq 1$. Now, LOUDS formulas become as follows:

- $child_{left/right}(i) = f - 1 + 2 \cdot rank_{R_B}(i, 1) + 0/1$,
- $parent(i) = select_{R_B}(\lfloor (i - f + 1)/2 \rfloor)$ ($i$ is a root if $i \leq f$).

We call this solution `GMR LOUDS*`. Although constant-time in theory, this solution resorts to *select* on bitmaps, which is not that fast in practice [15]. An alternative, less sophisticated, solution is to mark the beginning of the top-level trees of $R_B$ in another bitmap. Then we unroll the whole tree containing the occurrence in $R_S$ and spot the ancestors. We call this second solution `GMR`.

Figure 3 shows the algorithm for retrieving the reverse neighbors. We use the bitmap $B$ that marks the beginning of each adjacency list in $C$. This bitmap is included in the original structure [14], so it does not add any more space and allows us to determine to which list a position in $C$ belongs.

No reverse neighbor is reported twice: Even if we find several times the same position of $C$ along the process, it will be for different occurrences within $C$.

On the other hand, we are not providing any time guarantee for the process, because as explained we might do sterile work for ancestors in $R_B$ which do not appear in $C$. For the others we obtain at least one occurrence per access to the sequences. We address this problem next.

## 5 Guaranteeing Reverse Neighbor Retrieval Time

A way to alleviate the problem in practice is to include a bitmap, parallel to $R_B$, which indicates, for each internal node, whether or not it appears in $C$ or in $R_S$. This can be combined either with the LOUDS or the basic representation of $R_B$. Each time we find an ancestor, the parallel bitmap indicates immediately whether it is worth paying the effort of looking for it in $C$ and $R_S$. Indeed, given the negligible cost of such a bitmap, we opt for storing two of them: one referring

to $C$ and the other to $R_S$. This helps us limiting further unnecessary searches, although we still pay a constant-time cost to process useless nodes. We will test these bitmaps in combination with binary-tree LOUDS (`GMR LOUDS* M`), with general LOUDS (`GMR LOUDS M`, just to test it loses to the previous one), and with the basic representation (`GMR M`).

An alternative to achieve a logarithmic-time guarantee per retrieved neighbor is to use balanced Re-Pair [29], which enforces logarithmic rule heights. Since the roots of the DAG must appear in $C$, in the worst case we pay $O(1)$ time to discard (using the bitmaps introduced above) each element of an upward path (of length now limited to $O(\log m)$), except the root. As the root yields at least one neighbor, we can charge this $O(\log m)$ time to that result.

A solution that guarantees a constant number of operations on the sequences per reverse neighbor delivered is to effectively remove from the forest those nodes that do not appear in $C$ nor in $R_S$. The children of the removed node become children of their former grandparent. This can be done precisely because those nodes will not be accessed from elsewhere. The result is not anymore binary, but a general tree that is represented with the original LOUDS format [21,3]. Now we can prove our result.

**Lemma 1.** *Using the reduced tree, we pay $O(1)$ operations on the sequences per reverse neighbor delivered.*

*Proof.* Consider the original DAG with the useless nodes removed. Then each node either ($a$) is a root, ($b$) appears in $C$, or ($c$) has at least two parents (i.e., appears again in $R_S$). The algorithm in Figure 3 is equivalent to starting from some arbitrary node and traversing the DAG upwards, so that a constant number of operations on $R_S$, $R_B$ and $C$ are carried out ($i$) per DAG node considered and ($ii$) per result retrieved. We focus on ($i$). Nodes of type ($a$) and ($b$) yield at least one result, so their cost can be absorbed by ($ii$). For nodes of type ($c$), they have at least two parents, and thus each unit of work invested on them increases at least by 1 the number of results to report. □

Therefore, combined with GMR representation, we recover the constant time per reverse neighbor we had in Section 3. This representation is called `GMR LOUDS` in the experiments.

## 6 Experimental Results

The experiments were run on a 2GHz Intel Xeon (8 cores) with 16 GB RAM, running Ubuntu GNU/Linux with kernel 2.6.22-14 SMP (64 bits). The code was compiled with `g++` using the `-O9` directive.

From the several Re-Pair based versions studied in previous work [14], we chose "Reord CDict NoPtrs". For the wavelet trees we used the version without pointers [15]. For the GMR structure we used the simpler and faster *chunk* variant [15], as the grown alphabet after running Re-Pair on $T(G)$ (originally $n$) and the reduced length (originally $m$) become sufficiently similar.

**Table 3.** Space consumption (in bpe) of the Re-Pair based compressed representations of the adjacency lists, and previous work

| Crawl | Re-Pair WT | Re-Pair GMR | Re-Pair (dir+rev) | $k^2$-tree | WebGraph (dir+rev) | Asano ×2 |
|---|---|---|---|---|---|---|
| `EU` | 3.93 | 5.86 | 7.65 | 5.20 | 7.20 | 5.56 |
| `Indochina` | 2.30 | 3.65 | 4.54 | 2.82 | 2.94 | |
| `UK` | 3.98 | 6.22 | 7.50 | 4.20 | 4.34 | |
| `Arabic` | 2.72 | 4.15 | 5.53 | | 3.25 | |

We first focus on achieving minimum space usage, while still retrieving direct and reverse neighbors within reasonable time, that is, much faster than decompressing the whole graph. Later we focus on the faster alternatives. The space we report does not include special bitmaps for computing in/outdegrees.

Table 3 shows the space required for the four crawls. The first two columns are our contributions. In column `Re-Pair WT` we represent $C$ and $R_S$ using a compressed wavelet tree with sample value 64 [15] (space decreases by about 0.20 bpe more with larger sample values, but retrieval times degrade). In column `Re-Pair GMR` we show the representation that combines Re-Pair with a GMR *chunk*. Next columns are previous alternatives. Column `Re-Pair` shows the space needed by Re-Pair compression (variant "Diffs CDict NoPtrs") of the graph plus its transpose (so as to support direct and reverse queries) [13]. Column $k^2$`-tree` gives the smallest space achieved by that technique [9] (the space for the largest graph, `Arabic`, is not reported in there, and we could not build it either).

Column `WebGraph` gives the space achieved by the *WebGraph* technique [7], version 2.4.2, using variant `strictHostByHostGray`, which gave the best results. We add up the space for the direct and the transposed graph. We account only the space the structure requires on disk, even if the process requires much more memory to run. On the other hand, we account for their "offset" structure, which is the one providing direct access to the neighbors (without the offsets, the structure degenerates into a pure compression scheme). For this experiment we set the parameters so as to largely favor compression over speed (window size 10, maximum reference unlimited). With this compression they retrieve direct neighbors in about 100 microseconds.

Finally, column `Asano`×2 shows the space achieved by Asano et al. [4] on the `EU` graph (which is the largest graph they report). We double the space to account for the transposed graph. The time they report is over 1 millisecond per neighbor retrieved, whereas typical times (as shown next) are a few microseconds. Doubling the space is a bit pessimistic, as the transposed graph compresses slightly better, but still the difference with `Re-Pair WT` is significant, and this was the only reasonable way we found to try including them in the comparison.

We observe that our techniques require less space than adding up direct and reverse Re-Pair compressed graphs, while achieving good performance, as we see soon. By combining with a wavelet tree, on the other hand, we achieve *the smallest space reported in the literature* while supporting direct and reverse neighbors in reasonable time: around 35 microseconds/edge for direct and 55 for

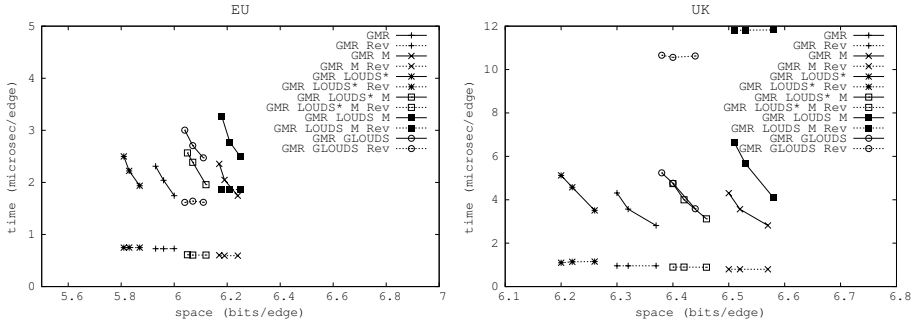



**Fig. 4.** Space/time tradeoffs of the different dictionary representations

reverse neighbors. The next experiments show that, using more space, one can reduce these times by an order of magnitude. However, no alternative scheme can operate within tens of microseconds and achieve the space of Re-Pair WT.

Figure 4 shows direct and reverse neighbor retrieval times on two crawls, for the different alternatives studied in Sections 4.2 and 5. Reverse retrieval times are marked `Rev`. As it can be seen, the general LOUDS versions (without modifier "`*`") lose to the simpler ones, and also the idea of marking nodes (suffix "`M`") does not pay off. The space/time map is dominated by `GMR` and `GMR LOUDS*`. We use `GMR` for the rest of the experiments.

Figure 5 shows retrieval times obtained for the four crawls, for both forward and reverse neighbors. We include only the techniques that are most competitive in time: `WebGraph` (storing both direct and reverse graphs), `Re-Pair` (storing both direct and reverse graphs), `Re-Pair GMR` (ours), and $k^2$`-trees` (variants called `Hybrid5` and `Hybrid37`, which give the best space/time tradeoffs [9]). We also include a variant of `Re-Pair GMR` labeled "`(2)`", where we use the variant of GMR that solves *access* in $O(1)$ time and *select* in time $O(\log \log n)$. Thus, while `Re-Pair GMR` is faster for reverse neighbors (using constant-time *select*), `Re-Pair GMR (2)` is faster on direct neighbors (using constant-time *access*)[4]. When times are not constant, an internal sampling used to compute an inverse permutation produces the observed space/time tradeoff.

`Re-Pair GMR` is not as fast as `Re-Pair` (at best, 3 times slower), but it requires significantly less space (about 25%). The $k^2$`-tree` (with variant `Hybrid5`) can achieve about 13% less space than the second point of `Re-Pair GMR` (recall that $k^2$`-tree` used much more space than `Re-Pair WT`, but it is much faster than it). Yet, when using that space, it is either 4–7 times slower for direct neighbors and 1.0–1.5 times faster for reverse neighbors (if using our variant `(2)`), or about 3–5 times slower for reverse neighbors and similar for direct neighbors (if using our

[4] Alternatively, we could have used the original structure and index the transposed graph, but this turned out not to be a good idea: compression of the reverse graph generates many more dictionary symbols and deeper dictionary trees, and thus both queries are slower than on `Re-Pair GMR (2)`.

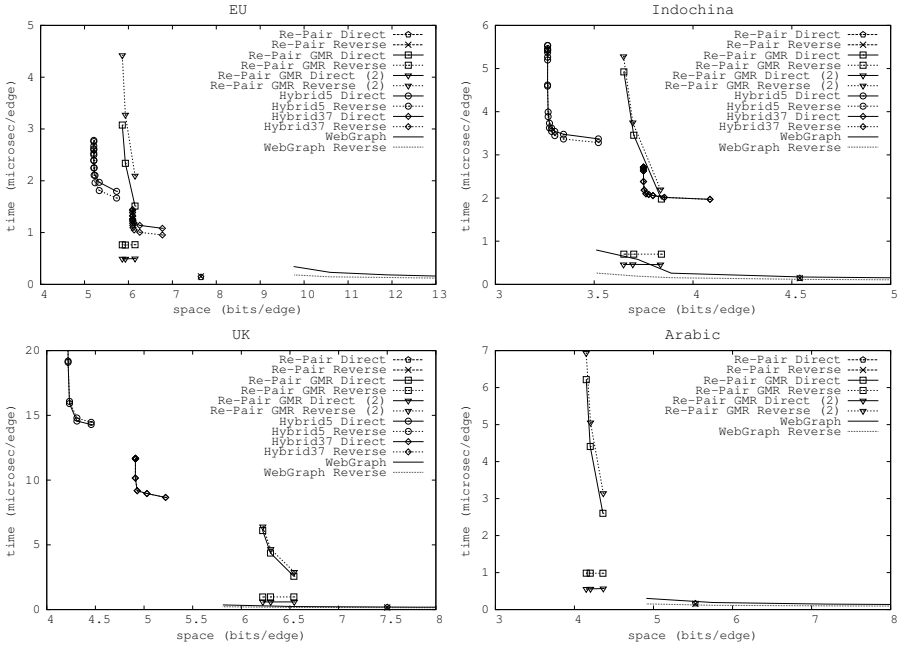



**Fig. 5.** Space/time tradeoffs of the most competitive variants

first variant). A nice point in $k^2$-tree is that it is symmetric (in technique and time) to obtain forward or reverse neighbors. A nice point in our structure is that, if one is interested mainly in direct or reverse neighbors, one can choose one of the two alternatives and be much faster on those queries, while still supporting the others in reasonable time. Adding up both times, we see that our alternative would be very close to Hybrid37, in space and time, if both direct and reverse neighbors had to be obtained. (An exception is graph UK, where $k^2$-tree is 35% smaller than Re-Pair GMR, but siginificantly slower in all aspects.)

For WebGraph we show the curves reaching as much as possible to the left; using less space yields a sudden increase in time. The comparison with our technique is mixed. On EU and Arabic, WebGraph cannot approach the space we use (while maintaining reasonable retrieval performance). On Indochina, both achieve comparable results. On UK, instead, WebGraph dominates.

## 7 Conclusions

We introduced a technique to represent Web graphs in compressed form so that not only fast access to the (direct) neighbors is supported, but also to the reverse neighbors (that is, nodes pointing to a given one). This has many applications to several Web analysis and mining tasks, where the memory limitations pose serious obstacles to analyzing massive graphs. Our representation combines grammar-based compression with compact data structures for sequences

that represent the compressed adjacency lists relations, and for trees that represent the grammar DAG. We provide several solutions, which support direct and reverse neighbor retrieval within a time that ranges from constant to logarithmic.

We achieve several relevant space/time tradeoffs. On one hand, we achieve the most compact functional Web graph representation reported up to date. On a sample of Web crawls, it required 2.3–4.0 bits per edge (bpe) while supporting direct and reverse navigation within a few tens of microseconds per neighbor. The best alternatives require 2.8–5.2 bpe for the same functionality. Compared to a $2 \times 32$-bit plain representation of the graph plus its transpose, we allow handling graphs 15–30 times larger within the same main memory.

If slightly more space is available, our faster representation requiring 3.6–6.5 bpe is of interest. It supports direct and reverse navigation within 1–3 microseconds per neighbor, occupying a relevant niche among alternative representations.

It would be of interest to extend this research to the compression of other types of networks with similar characteristics. For example, compression of social networks is starting to receive attention [11]. These share some characteristics with Web graphs, yet they have other unique ones such as reciprocity in links and presence of relatively large cliques or bicliques. In particular, many social networks are undirected. With current techniques, the representation of an undirected graph forces either to duplicate each edge $\{u, v\}$ as $(u, v)$ and $(v, u)$, or to choose arbitrarily from both, but then the (undirected) neighbors of $v$ will be the union of its direct and reverse neighbors under this representation. Data structures like ours ours are ideal for this scenario.

## References

1. Adler, M., Mitzenmacher, M.: Towards compressing Web graphs. In: Proc. 11th DCC, pp. 203–212 (2001)
2. Aiello, W., Chung, F., Lu, L.: A random graph model for massive graphs. In: Proc. 32th STOC, pp. 171–180 (2000)
3. Arroyuelo, D., Cánovas, R., Navarro, G., Sadakane, K.: Succinct trees in practice. In: Proc. 11th ALENEX, pp. 84–97 (2010)
4. Asano, Y., Miyawaki, Y., Nishizeki, T.: Efficient compression of Web graphs. In: Hu, X., Wang, J. (eds.) COCOON 2008. LNCS, vol. 5092, pp. 1–11. Springer, Heidelberg (2008)
5. Barbay, J., Golynski, A., Munro, I., Rao, S.S.: Adaptive searching in succinctly encoded binary relations and tree-structured documents. In: Proc. 17th CPM, pp. 24–35 (2006)
6. Barbay, J., He, M., Munro, I., Rao, S.S.: Succinct indexes for strings, binary relations and multi-labeled trees. In: Proc. 18th SODA, pp. 680–689 (2007)
7. Boldi, P., Santini, M., Vigna, S.: Permuting web graphs. In: Avrachenkov, K.E., Donato, D., Litvak, N. (eds.) WAW 2009. LNCS, vol. 5427, pp. 116–126. Springer, Heidelberg (2009)
8. Boldi, P., Vigna, S.: The WebGraph framework I: compression techniques. In: Proc. 13th WWW, pp. 595–602 (2004)
9. Brisaboa, N.R., Ladra, S., Navarro, G.: $k^2$-trees for compact web graph representation. In: Hyyro, H. (ed.) SPIRE 2009. LNCS, vol. 5721, pp. 18–30. Springer, Heidelberg (2009)

10. Broder, A., Kumar, R., Maghoul, F., Raghavan, P., Rajagopalan, S., Stata, R., Tomkins, A., Wiener, J.: Graph structure in the Web. Journal of Computer Networks 33(1-6), 309–320 (2000)
11. Chierichetti, F., Kumar, R., Lattanzi, S., Mitzenmacher, M., Panconesi, A., Raghavan, P.: On compressing social networks. In: Proc. 15th KDD, pp. 219–228 (2009)
12. Clark, D.: Compact Pat Trees. Ph.D. thesis, University of Waterloo (1996)
13. Claude, F., Navarro, G.: A fast and compact Web graph representation. In: Ziviani, N., Baeza-Yates, R. (eds.) SPIRE 2007. LNCS, vol. 4726, pp. 105–116. Springer, Heidelberg (2007)
14. Claude, F., Navarro, G.: Fast and compact Web graph representations. Tech. Rep. TR/DCC-2008-3, Dept. of Comp. Sci., Univ. of Chile (2008)
15. Claude, F., Navarro, G.: Practical rank/select queries over arbitrary sequences. In: Amir, A., Turpin, A., Moffat, A. (eds.) SPIRE 2008. LNCS, vol. 5280, pp. 176–187. Springer, Heidelberg (2008)
16. Donato, D., Laura, L., Leonardi, S., Meyer, U., Millozzi, S., Sibeyn, J.: Algorithms and experiments for the Web graph. Journal of Graph Algorithms and Applications 10(2), 219–236 (2006)
17. Erdős, P., Rényi, A.: On random graphs I. Publicationes Mathematicae 6, 290–297 (1959)
18. Golynski, A., Munro, I., Rao, S.: Rank/select operations on large alphabets: a tool for text indexing. In: Proc. 17th SODA, pp. 368–373 (2006)
19. González, R., Navarro, G.: Compressed text indexes with fast locate. In: Ma, B., Zhang, K. (eds.) CPM 2007. LNCS, vol. 4580, pp. 216–227. Springer, Heidelberg (2007)
20. Grossi, R., Gupta, A., Vitter, J.: High-order entropy-compressed text indexes. In: Proc. 14th SODA, pp. 841–850 (2003)
21. Jacobson, G.: Succinct Static Data Structures. Ph.D. thesis, Carnegie Mellon University (1989)
22. Kleinberg, J., Kumar, R., Raghavan, P., Rajagopalan, S., Tomkins, A.: The Web as a graph: Measurements, models, and methods. In: Asano, T., Imai, H., Lee, D.T., Nakano, S.-i., Tokuyama, T. (eds.) COCOON 1999. LNCS, vol. 1627, pp. 1–17. Springer, Heidelberg (1999)
23. Kleinberg, J.M.: Authoritative sources in a hyperlinked environment. Journal of the ACM 46(5), 604–632 (1999)
24. Larsson, J., Moffat, A.: Off-line dictionary-based compression. Proc. IEEE 88(11), 1722–1732 (2000)
25. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Computing Surveys 39(1), article 2 (2007)
26. Raman, R., Raman, V., Rao, S.: Succinct indexable dictionaries with applications to encoding $k$-ary trees and multisets. In: Proc. 13th SODA, pp. 233–242 (2002)
27. Rusmevichientong, P., Pennock, D., Lawrence, S., Giles, C.L.: Methods for sampling pages uniformly from the World Wide Web. In: Proc. AAAI Fall Symposium on Using Uncertainty Within Computation, pp. 121–128 (2001)
28. Saito, H., Toyoda, M., Kitsuregawa, M., Aihara, K.: A large-scale study of link spam detection by graph algorithms. In: Proc. 3rd AIRWeb (2007)
29. Sakamoto, H.: A fully linear-time approximation algorithm for grammar-based compression. Journal of Discrete Algorithms 3(2-4), 416–430 (2005)
30. Suel, T., Yuan, J.: Compressing the graph structure of the Web. In: Proc. 11th DCC, pp. 213–222 (2001)

# A Parallel Algorithm for Fixed-Length Approximate String-Matching with *k*-mismatches

Maxime Crochemore[1,2], Costas S. Iliopoulos[1,3], and Solon P. Pissis[1]

[1] Dept. of Computer Science, King's College London, London WC2R 2LS, UK
[2] Institut Gaspard-Monge, Université Paris-Est, 77454 Marne-la-Vallée, France
[3] Digital Ecosystems & Business Intelligence Institute, Curtin University
GPO Box U1987 Perth WA 6845, Australia
{mac,csi,pississo}@dcs.kcl.ac.uk

**Abstract.** This paper deals with the approximate string-matching problem with Hamming distance. The approximate string-matching with $k$-mismatches problem is to find all locations at which a query of length $m$ matches a factor of a text of length $n$ with $k$ or fewer mismatches. The approximate string-matching algorithms have both pleasing theoretical features, as well as direct applications, especially in computational biology. We consider a generalisation of this problem, the *fixed-length approximate string-matching with k-mismatches problem*: given a text $t$, a pattern $x$ and an integer $\ell$, search for all the occurrences in $t$ of all factors of $x$ of length $\ell$ with $k$ or fewer mismatches with a factor of $t$. We present a practical parallel algorithm of comparable simplicity that requires only $\mathcal{O}(\frac{nm\lceil \ell/w \rceil}{p})$ time, where $w$ is the word size of the machine (e.g. 32 or 64 in practice) and $p$ the number of processors. Thus the algorithm's performance is independent of $k$ and the alphabet size $|\Sigma|$. The proposed parallel algorithm makes use of message-passing parallelism model, and word-level parallelism for efficient approximate string-matching.

**Keywords:** string algorithms, parallel algorithms, approximate string-matching.

## 1 Introduction

The problem of finding factors of a text similar to a given pattern has been intensively studied over the last thirty years and it is a central problem in a wide range of applications, including file comparison, spelling correction, information retrieval, and searching for similarities among biosequences.

One of the most common variants of the approximate string-matching problem is that of finding factors that match the pattern with at most $k$-differences. The first algorithm addressing exactly this problem is attributable to Sellers [15]. Sellers algorithm requires $\mathcal{O}(mn)$ time, where $m$ is the length of the query and $n$ is the length of the text. One of the first intensive study on the question is by Ukkonen [16]. A thread of practice-oriented results exploited the hardware word-level

T. Elomaa et al. (Eds.): Ukkonen Festschrift, LNCS 6060, pp. 92–101, 2010.

parallelism of bit-vector operations. Wu and Manber in [18] showed an $\mathcal{O}(knm/w)$ algorithm for the $k$-differences problem, where $w$ is the number of bits in a machine word. Baeza-Yates and Navarro in [1] have shown a $\mathcal{O}(knm/w)$ variation on the Wu-Manber algorithm, implying $\mathcal{O}(n)$ performance when $km = \mathcal{O}(w)$. Another general solution based on existing algorithms can be found in [3].

In this paper, we consider the following versions of the sequence comparison problem: given a solution for the comparison of $A$ and $B = b\hat{B}$, can one incrementally compute a solution for $A$ versus $\hat{B}$? and given a solution for the comparison of $A$ and $\hat{B}$, can one incrementally compute a solution for $A$ versus $\hat{B}c$? Here $b$ and $c$ are additional symbols. By solution we mean some encoding of a relevant portion of the traditional dynamic programming matrix $D$ computed for comparing $A$ and $B$.

Landau, Myers and Schmidt in [8] demonstrated the power of efficient algorithms answering the above questions, with a variety of applications to computational problems such as "the longest common subsequence problem", "the longest prefix approximate match problem", "approximate overlaps in the fragment assembly problem", "cyclic string comparison" and "text screen updating".

The above ideas are the bases of the *fixed-length approximate string-matching problem*: given a text $t$ of length $n$, a pattern $x$ of length $m$ and an integer $\ell$, compute the optimal alignment of all factors of $x$ of length $\ell$ with factors of $t$. Iliopoulos, Mouchard and Pinzon in [7] presented the MAX-SHIFT algorithm, a bit-vector algorithm that requires $O(nm\lceil \ell/w \rceil)$ time and its performance is independent of $k$. As such, it can be used to compute blocks of dynamic programming matrix as the 4-Russians algorithm (see [19]).

In this paper, we consider the *fixed-length approximate string-matching with k-mismatches problem*: given a text $t$, a pattern $x$ and an integer $\ell$, search for all the occurrences in $t$ with $k$ or fewer mismatches of all factors of $x$ of length $\ell$. There has been ample work in the literature for devising parallel algorithms for different models and platforms, for the approximate string-matching problem [2], [4], [6], [10], [13]. We design and analyse a practical parallel algorithm for addressing the fixed-length approximate string-matching problem with $k$-mismatches in $\mathcal{O}(\frac{nm\lceil \ell/w \rceil}{p})$ time. Thus the algorithm's performance is independent of $k$ and the alphabet size $|\Sigma|$ (provided that a letter fits in a computer word). The proposed algorithm makes use of message-passing parallelism model, and word-level parallelism for efficient approximate string matching.

The rest of the paper is structured as follows. In Section 2, the basic definitions that are used throughout the paper are presented. In Section 3, we formally define the problem solved in this paper. In Sections 4 and 5, we present the sequential and the parallel algorithm, respectively. In Section 6, we present the experimental results of the proposed algorithm. Finally, we briefly conclude in Section 7.

## 2 Basic Definitions

A *string* or *sequence* is a succession of zero or more symbols from an alphabet $\Sigma$ of cardinality $s$; the string with zero symbols is denoted by $\epsilon$. The set of

all strings over the alphabet $\Sigma$ is denoted by $\Sigma^*$. A string $x$ of length $m$ is represented by $x[1\,..\,m]$, where $x[i] \in \Sigma$ for $1 \leq i \leq m$. The length of a string $x$ is denoted by $|x|$. We say that $\Sigma$ is *bounded* when $s$ is a constant, *unbounded* otherwise. A string $w$ is a factor of $x$ if $x = uwv$ for $u, v \in \Sigma^*$.

Consider the sequences $x$ and $y$ with $x[i], y[i] \in \Sigma \cup \{\epsilon\}$. If $x[i] \neq y[i]$, then we say that $x[i]$ *differs* from $y[i]$. We distinguish among the following three types of differences:

1. A symbol of the first sequence corresponds to a different symbol of the second one, then we say that we have a *mismatch* between the two characters, i.e., $x[i] \neq y[i]$.
2. A symbol of the first sequence corresponds to "no symbol" of the second sequence, that is $x[i] \neq \epsilon$ and $y[i] = \epsilon$. This type of difference is called a *deletion*.
3. A symbol of the second sequence corresponds to "no symbol" of the first sequence, that is $x[i] = \epsilon$ and $y[i] \neq \epsilon$. This type of difference is called an *insertion*.

As an example of the types of differences, see Figure 1.

```
          1 2 3 4 5 6 7 8
String x: B A D F E ϵ C A
          | | | | | | |
String y: B C D ϵ E B C A
```

**Fig. 1.** Types of differences: mismatch in position 2 (A, C), deletion in position 4 (F, $\epsilon$), insertion in position 6 ($\epsilon$, B)

Another way of seeing this difference is that one can transform the $x$ sequence to $y$ by performing operations. The edit distance, $\delta_E(x, y)$, between strings $x$ and $y$, is the minimum number of operations required to transform $x$ into $y$. These operations are *Replacement* of a mismatched symbol, a *Deletion* or an *Insertion* of a symbol. The edit distance is symmetrical, and it holds $0 \leq \delta_E(x, y) \leq max(|x|, |y|)$.

Let $t = t[1\,..\,n]$ and $x = x[1\,..\,m]$ with $m \leq n$. We say that $x$ occurs at position $q$ of $t$ with at most $k$-differences (or equivalently, a *local alignment of $x$ and $t$ at position $q$ with at most $k$-differences*), if $t[q] \ldots t[r]$, for some $r > q$, can be transformed into $x$ by performing at most $k$ of the following operations: inserting, deleting or replacing a symbol.

The Hamming distance $\delta_H$ is defined only for strings of the same length. For two strings $x$ and $y$, $\delta_H(x, y)$ is the number of places in which the two strings differ, i.e. have different characters. Formally

$$\delta_H(x, y) = \sum_{i=1}^{|x|} 1_{x[i] \neq y[i]} \text{ , where } 1_{x[i] \neq y[i]} = \begin{cases} 1, & \text{if } x[i] \neq y[i] \\ 0, & \text{otherwise} \end{cases} \quad (1)$$

The Hamming distance is symmetrical, and it holds $0 \leq \delta_H(x, y) \leq |x|$.

## 3 Problem Definition

The focus is on computing matrix $M$, which contains the number of mismatches of all factors of pattern $x$ of length $\ell$ and any contiguous factor of the text $t$ of length $\ell$.

*Example.* Let the text $t = x = GGGTCTA$ and $\ell = 3$. Table 1 shows the matrix $M$.

**Table 1.** Matrix $M$ for $t = x = GGGTCTA$ and $\ell = 3$

| | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|---|
| | | $\epsilon$ | $G$ | $G$ | $G$ | $T$ | $C$ | $T$ | $A$ |
| 0 | $\epsilon$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | $G$ | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 2 | $G$ | 2 | 1 | 0 | 0 | 1 | 2 | 2 | 2 |
| 3 | $G$ | 3 | 2 | 1 | 0 | 1 | 2 | 3 | 3 |
| 4 | $T$ | 3 | 3 | 2 | 1 | 0 | 2 | 2 | 3 |
| 5 | $C$ | 3 | 3 | 3 | 2 | 2 | 0 | 3 | 2 |
| 6 | $T$ | 3 | 3 | 3 | 3 | 2 | 3 | 0 | 3 |
| 7 | $A$ | 3 | 3 | 3 | 3 | 3 | 2 | 3 | 0 |

*Example.* Let the text $t = GTGAACT$, $x = GTCACGT$ and $\ell = 3$. Table 2 shows the matrix $M$.

**Table 2.** Matrix $M$ for $t = GTGAACT$, $x = GTCACGT$ and $\ell = 3$

| | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|---|
| | | $\epsilon$ | $G$ | $T$ | $G$ | $A$ | $A$ | $C$ | $T$ |
| 0 | $\epsilon$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | $G$ | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 |
| 2 | $T$ | 2 | 2 | 0 | 2 | 1 | 2 | 2 | 1 |
| 3 | $C$ | 3 | 3 | 3 | 1 | 3 | 2 | 2 | 3 |
| 4 | $A$ | 3 | 3 | 3 | 3 | 1 | 2 | 3 | 2 |
| 5 | $C$ | 3 | 3 | 3 | 3 | 3 | 2 | 1 | 3 |
| 6 | $G$ | 3 | 2 | 3 | 2 | 3 | 3 | 2 | 1 |
| 7 | $T$ | 3 | 3 | 1 | 3 | 2 | 3 | 3 | 2 |

The *fixed-length approximate string-matching with at most k-mismatches* problem can be formally defined as follows.

**Problem 1.** Given a text $t$ of length $n$, a pattern $x$ of length $m$ and an integer $\ell$, find all factors of $x$ of length $\ell$ that match any contiguous factor of $t$ of length $\ell$ with at most $k$-mismatches.

## 4 The Bit-Vector-Mismatches Algorithm

Iliopoulos, Mouchard and Pinzon in [7] presented the Max-Shift algorithm, a bit-vector algorithm that solves the *fixed-length approximate string-matching* problem: given a text $t$ of length $n$, a pattern $x$ of length $m$ and an integer $\ell$, compute the optimal alignment of all factors of $x$ of length $\ell$ and a factor of $t$. The focus of the Max-Shift algorithm is on computing matrix $D'$, which contains the best scores of the alignments of all factors of pattern $x$ of length $\ell$ and any contiguous factor of the text $t$.

The Max-Shift algorithm makes use of word-level parallelism in order to compute matrix $D'$ efficiently, similar to the manner used by Myers in [12]. The algorithm is based on the $\mathcal{O}(1)$ time computation of each $D'[i,j]$ by using bit-vector operations, under the assumption that $\ell \leq w$, where $w$ is the number of bits in a machine word or $\mathcal{O}(\ell/w)$-time for the general case. The algorithm maintains a bit-vector matrix $B[0\,..\,m, 0\,..\,n]$, where the bit integer $B[i,j]$, holds the binary encoding of the path in $D'$ to obtain the optimal alignment at $i,\ j$ with the differences occurring as leftmost as possible.

Here the key idea is to devise a bit-vector algorithm for the *fixed-length approximate string-matching with at most k-mismatches* problem. We maintain the bit-vector $B[i,j] = b_\ell\,..\,b_1$, where $b_\lambda = 1,\ 1 \leq \lambda \leq \ell$, if there is a mismatch of a contiguous factor of the text $t[i-\ell+1\,..\,i]$ and $x[j-\ell+1\,..\,j]$ in the $\lambda^{th}$ position. Otherwise we set $b_\lambda = 0$.

Given the restraint that the integer $\ell$ is less than the length of the computer word $w$, then the bit-vector operations allow to update each entry of the matrix $B$ in constant time (using "shift"-type of operation on the bit-vector). The maintenance of the bit-vector is done via operations defined as follows:

1. *shiftc*$(x)$: shifts and truncates the leftmost bit of $x$.
2. $\delta_H(x,y)$: returns the minimum number of replacements required to transform $x$ into $y$

The Bit-Vector-Mismatches algorithm for computing the bit-vector matrix $B$ and matrix $M$ is outlined in Figure 2.

*Example.* Let the text $t = x = GGGTCTA$ and $\ell = 3$. Table 3 shows the bit-vector matrix $B$. Consider the case when $i = 7$ and $j = 5$. Cell $B[7,5] = 101$ denotes that factors $t[3\,..\,5] = CTA$ and $t[5\,..\,7] = GTC$ have a mismatch in position 1, a match in position 2, and a mismatch in position 3, resulting in a total of two mismatches, as shown in cell $M[7,5]$ (see Table 1).

Assume that the bit-vector matrix $B[0\,..\,m, 0\,..\,n]$ is given. We can use the function *ones*$(v)$, which returns the number of 1's (bits set on) in the bit-vector $v$, to compute matrix $M$ (see Figure 2, line 11).

**Theorem 1.** Given the text $t = t[1\,..\,n]$, the pattern $x = x[1\,..\,m]$, the motif length $\ell$, and the size $w$ of the computer word, the Bit-Vector-Mismatches algorithm correctly computes the matrix $M$ in $\mathcal{O}(nm\lceil \ell/w \rceil)$ units of time.

```
Bit-Vector-Mismatches
▷Input: t, n, x, m, ℓ
▷Output: B, M
 1  begin
 2    ▷ Initialisation
 3    for i ← 0 until n do
 4      B[0, i] ← 0; M[0, i] ← 0
 5    for i ← 0 until m do
 6      B[i, 0] ← min(i, ℓ) 1's; M[i, 0] ← min(i, ℓ)
 7    ▷ Matrix B and Matrix M computation
 8    for i ← 1 until m do
 9      for j ← 1 until n do
10          B[i, j] ← shiftc(B[i − 1, j − 1]) OR δ_H(x[i], t[j])
11          M[i, j] ← ones(B[i, j])
12  end
```

**Fig. 2.** The BIT-VECTOR-MISMATCHES algorithm for computing matrix $B$ and matrix $M$

**Table 3.** The bit-vector matrix $B$ for $t = x = GGGTCTA$ and $\ell = 3$

| | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|---|
| | | $\epsilon$ | $G$ | $G$ | $G$ | $T$ | $C$ | $T$ | $A$ |
| 0 | $\epsilon$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | $G$ | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 2 | $G$ | 11 | 10 | 00 | 00 | 01 | 11 | 11 | 11 |
| 3 | $G$ | 111 | 110 | 100 | 000 | 001 | 011 | 111 | 111 |
| 4 | $T$ | 111 | 111 | 101 | 001 | 000 | 011 | 110 | 111 |
| 5 | $C$ | 111 | 111 | 111 | 011 | 011 | 000 | 111 | 101 |
| 6 | $T$ | 111 | 111 | 111 | 111 | 110 | 111 | 000 | 111 |
| 7 | $A$ | 111 | 111 | 111 | 111 | 111 | 101 | 111 | 000 |

*Proof.* Without loss of generality, assume that we want to compute cell $M[i, j]$, where

$$M[i, j] = \delta_H(x[i - \ell + 1 \,..\, i], t[j - \ell + 1 \,..\, j]) \tag{2}$$

It is not difficult to see that,

$$\delta_H(x[i-\ell+1 \,..\, i], t[j-\ell+1 \,..\, j]) = \delta_H(x[i-\ell+1 \,..\, i-1], t[j-\ell+1 \,..\, j-1]) + \delta_H(x[i], t[j]) \tag{3}$$

Let $last(b[\ell] \,..\, b[1])$ be an operation that returns the leftmost bit of the bit-vector $b$. It follows that,

$$\delta_H(x[i-\ell+1 \,..\, i-1], t[j-\ell+1 \,..\, j-1]) = M[i-1, j-1] - last(B[i-1, j-1]) \tag{4}$$

From Equations 2, 3 and 4,

$$M[i,j] = M[i-1,j-1] - last(B[i-1,j-1]) + \delta_H(x[i], t[j]) \tag{5}$$

Equation 5 is equivalent to line 10 of the BIT-VECTOR-MISMATCHES algorithm. □

Hence, this algorithm runs in $\mathcal{O}(nm)$ under the assumption that $\ell \leq w$ and its space complexity is reduced to $\mathcal{O}(n)$ by noting that each row of $B$ depends only on its immediately preceding row.

## 5 The PARALLEL-BIT-VECTOR-MISMATCHES Algorithm

The next proposed parallel algorithm makes use of the message-passing parallelism model by using $p$ processors. The following assumptions for the model of communications in the parallel computer are made. The parallel computer comprises a number of nodes. Each node comprises one or several identical processors interconnected by a switched communication network. The time taken to send a message of size $n$ between any two nodes is independent of the distance between nodes and can be modelled as $t_{comm} = t_s + nt_w$, where $t_s$ is the latency or start-up time of the message, and $t_w$ is the transfer time per data. The links between two nodes are full-duplex and single-ported: a message can be transferred in both directions by the link at the same time, and only one message can be sent and one message can be received at the same time.

We will use the *functional* decomposition, in which the initial focus is on the computation that is to be performed rather than on the data manipulated by the computation. We assume that both text $t$ and pattern $x$ are stored locally on each processor. This can be done by using a *one-to-all* broadcast operation in $(t_s + t_w(n+m)) \log p$ communication time, which is asymptotically $\mathcal{O}(n \log p)$.

The key idea behind parallelising the BIT-VECTOR-MISMATCHES algorithm, is that cell $B[i,j]$ can be computed only in terms of $B[i-1,j-1]$. Based on this, if we partition the problem of computing matrix $B$ (and $M$) into a set of diagonal vectors $\Delta_0, \Delta_1, .., \Delta_{n+m}$, as shown in Equation 6, the computation of each one of these would be independent, and hence parallelisable.

$$\Delta_\nu[x] = \begin{cases} B[\nu - x,\ x] \ :\ 0 \leq x \leq \nu, & \text{(a)} \\ B[m - x,\ \nu - m + x] \ :\ 0 \leq x < m + 1, & \text{(b)} \\ B[m - x,\ \nu - m + x] \ :\ 0 \leq x < n + m - \nu + 1, & \text{(c)} \end{cases} \tag{6}$$

where,
(a) if $0 \leq \nu < m$
(b) if $m \leq \nu < n$
(c) if $n \leq \nu < n + m + 1$

It is possible that in a certain diagonal $\Delta_\nu$, $\nu > 0$, a processor will need a cell or a pair of cells, which were not computed on its local memory in diagonal $\Delta_{\nu-1}$. We need a communication pattern in each diagonal $\Delta_\nu$, for all $0 \leq \nu < n + m$, which minimises the data exchange between the processors. It is obvious, that

in each diagonal, each processor needs only to communicate with its neighbours. In particular, in each diagonal, each processor needs to swap the boundary cells with its left and right neighbour processor.

An outline of the Parallel-Bit-Vector-Mismatches algorithm in each diagonal $\Delta_\nu$, for all $0 \leq \nu < n + m + 1$, is as follows:

**Step 1.** Each processor is assigned with $|\Delta_\nu|/p$ cells (without loss of generality).
**Step 2.** Each processor computes each allocated cell using the Bit-Vector-Mismatches algorithm.
**Step 3.** Processors communication involving point-to-point boundary cells swaps.

**Theorem 2.** Given the text $t = t[1\,..\,n]$, the pattern $x = x[1\,..\,m]$, the motif length $\ell$, the size $w$ of the computer word, and the number of processors $p$, the Parallel-Bit-Vector-Mismatches algorithm computes the matrix $M$ in $\mathcal{O}(\frac{nm\lceil \ell/w \rceil}{p})$ units of time.

*Proof.* We partition the problem of computing matrix $B$ into a set of $n + m + 1$ diagonal vectors, thus $\mathcal{O}(n)$ supersteps. In step 1, the allocation procedure runs in $\mathcal{O}(1)$ time. In step 2, the cells computation requires $\mathcal{O}(\frac{m\lceil \ell/w \rceil}{p})$ time. In step 3, the data exchange between the processors involves $\mathcal{O}(1)$ point-to-point message transfers. Hence, asymptotically, the overall time is $\mathcal{O}(\frac{nm\lceil \ell/w \rceil}{p})$. □

Hence, the parallel algorithm runs in $\mathcal{O}(\frac{nm}{p})$ under the assumption that $\ell \leq w$, and its space complexity is reduced to $\mathcal{O}(n)$ by noting that each diagonal vector $\Delta_\nu$ of matrix $B$, for all $2 \leq \nu \leq n + m$, depends only on $\Delta_{\nu-2}$.

## 6 Experimental Results

In order to evaluate the parallel efficiency of our algorithm, we implemented the Bit-Vector-Mismatches algorithm in ANSI $C$ language and parallelised it with the use of the *MPI* library. Both implementations, the sequential and the parallel algorithm, are available at a website[1], which has been set up for maintaining the source code and the documentation.

Experimental tests were run on 1 up to 16 processing nodes (2.6 GHz AMD Opteron) of a cluster architecture. As an input, DNA sequences of the mouse chromosome $X$ were used, retrieved from the *Ensembl* genome database. Experimental results regarding the execution time and measured speed-up are illustrated in Figures 3 and 4, respectively. The speed-up is calculated as the ratio of elapsed time with $p$ processors to elapsed time with one processor.

The presented experimental results demonstrate a good scaling of the code. The proposed algorithm scales well even for small problem sizes. As expected in some cases, when increasing the problem size, the algorithm achieves a linear speed-up, confirming our theoretical results. Further tests were conducted for different values of fixed-length $\ell$, with no difference observed, regarding the execution time.

[1] http://www.dcs.kcl.ac.uk/pg/pississo/

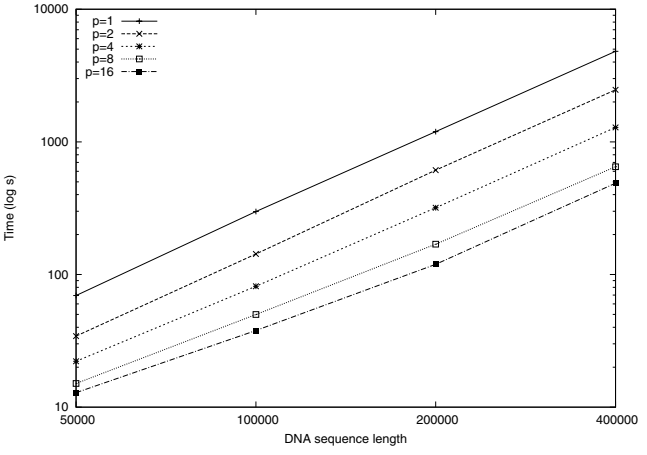



**Fig. 3.** Execution time for $t = x$ and $\ell = 20$

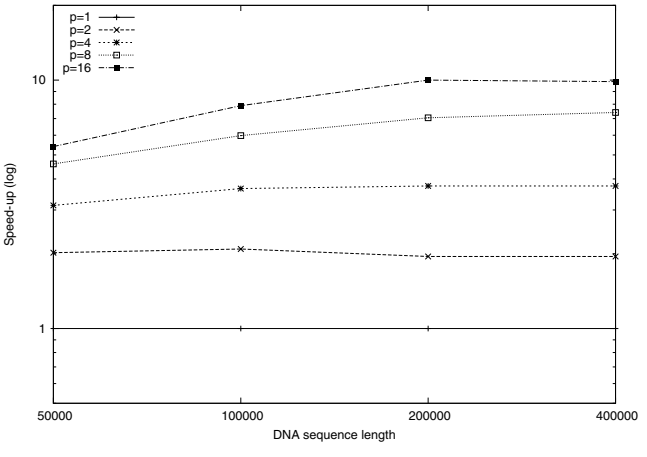



**Fig. 4.** Measured speed-up for $t = x$ and $\ell = 20$

## 7 Conclusion

We have presented a practical parallel algorithm that solves a generalisation of the approximate string-matching problem. In particular, the proposed parallel algorithm solves the fixed-length approximate string matching with $k$-mismatches problem in $\mathcal{O}(\frac{nm\lceil \ell/w \rceil}{p})$ time, which is $\mathcal{O}(\frac{nm}{p})$, in practical terms. It is considerably simple and elegant, it achieves a theoretical and practical linear speed-up, it does not require text preprocessing, it does not use/store look up tables and it does not depend on the number of differences $k$ and the alphabet size $|\Sigma|$.

## References


1. Baeza-Yates, R.A., Navarro, G.: A faster algorithm for approximate string matching. In: Hirschberg, D., Myers, G. (eds.) CPM 1996. LNCS, vol. 1075, pp. 1–23. Springer, Heidelberg (1996)
2. Bertossi, A.A., Luccio, F., Pagli, L., Lodi, E.: A parallel solution to the approximate string-matching problem. The Computer Journal 35(5), 524–526 (1992)
3. Crochemore, M., Iliopoulos, C.S., Pinzon, Y.J.: Speeding up Hirschberg and Hunt-Szymanski LCS algorithms. Fundamenta Informaticae 56(1,2), 89–103 (2002)
4. Galper, A.R., Brutlag, D.R.: Parallel similarity search and alignment with the dynamic programming method. Technical Report KSL 90–74. Stanford University, p. 14 (1990)
5. Hall, N.: Advanced sequencing technologies and their wider impact in microbiology. J. Exp. Biol. 210(pt 9), 1518–1525 (2007)
6. Huang, X.: A space-efficient parallel sequence comparison algorithm for a Message-Passing Multiprocessor. International Journal of Parallel Programming 18(3), 223–239 (1990)
7. Iliopoulos, C.S., Mouchard, L., Pinzon, Y.J.: The Max-Shift algorithm for approximate string matching. In: Brodal, G.S., Frigioni, D., Marchetti-Spaccamela, A. (eds.) WAE 2001. LNCS, vol. 2141, pp. 13–25. Springer, Heidelberg (2001)
8. Landau, G., Myers, G., Schmidt, J.P., Schmidt, P.: Incremental String Comparison. SIAM Journal on Computing 27, 557–582 (1995)
9. Landau, G.M., Vishkin, U.: Fast string matching with $k$ differences. Journal of Computer and Systems Sciences 37(1), 63–78 (1988)
10. Landau, G.M., Vishkin, U.: Fast parallel and serial approximate string matching. Journal of Algorithms 10(2), 157–169 (1989)
11. Margulies, E.H., Birney, E.: Approaches to comparative sequence analysis: towards a functional view of vertebrate genomes. Nat. Rev. Genet. 9(4), 303–313 (2008)
12. Myers, E.W.: A Fast Bit-Vector Algorithm for Approximate String Matching Based on Dynamic Programming. Journal of the ACM 46, 395–415 (1999)
13. dos Reis, C.C.T.: Approximate string-matching algorithm using parallel methods for molecular sequence comparisons. In: Portuguese conference on Artificial intelligence. EPIA 2005, pp. 140–143 (2005)
14. Schuster, S.C.: Next-generation sequencing transforms today's biology. Nature Methods 5(1), 16–18 (2007)
15. Seller, P.H.: The theory and computation of evolutionary distances: Pattern recognition. Journal of Algorithms 1(4), 359–373 (1980)
16. Ukkonen, E.: Finding approximate patterns in strings. J. of Algorithms 6(1), 132–137 (1985)
17. Wold, B., Myers, R.: Sequence consensus methods for functional genomics. Nature Methods 5(1), 19–21 (2007)
18. Wu, S., Manber, U.: Fast text searching allowing errors. CACM 35(10), 83–91 (1992)
19. Wu, S., Manber, U., Myers, G.: A subquadratic algorithm for approximate limited expression matching. Algorithmica 15(1), 50–67 (1996)

# Covering Analysis of the Greedy Algorithm for Partial Cover

Tapio Elomaa and Jussi Kujala

Department of Software Systems
Tampere University of Technology
P.O. Box 553, FI-33101 Tampere, Finland
elomaa@cs.tut.fi, jussi.kujala@iki.fi

**Abstract.** The greedy algorithm is known to have a guaranteed approximation performance in many variations of the well-known minimum set cover problem. We analyze the number of elements covered by the greedy algorithm for the minimum set cover problem, when executed for $k$ rounds. This analysis quite easily yields in the $p$-partial cover problem over a ground set of $m$ elements the harmonic approximation guarantee $H(\lceil pm \rceil)$ for the number of required covering sets. Thus, we tie together the coverage analysis of the greedy algorithm for minimum set cover and its dual problem partial cover.

## 1 Introduction

Minimum Set Cover is a fundamental combinatorial optimization problem with many practical applications. It is one of the oldest problems known to be NP-complete [1,2]. The goal in Minimum Set Cover is to cover all elements of the ground set by using as few subsets as possible from a given collection.

What also makes this problem very interesting is the fact that it can be approximated efficiently within guaranteed performance by the straightforward greedy algorithm [3,4,5]. Greedy approximation of Minimum Set Cover underlies approximation algorithms in many application fields; e.g., in machine learning [6,7], combinatorial pattern matching [8,9], and bioinformatics [10,11,12]. Thus, Minimum Set Cover has received a lot of analytical attention over the years. The endmost approximation possibilities of the problem and the performance of the greedy algorithm are well understood topics today.

Partial Cover [6] is a generalization of Minimum Set Cover in which one asks how many subsets are required to cover at least a fraction $p$, $0 < p \leq 1$, of the elements of the ground set. The greedy algorithm can be used also to approximate this problem, but it has to be changed in order to cope with Partial Cover. The required modifications, though, are small.

In this paper we draw a connection that has not been explicit before. We show that directly by analyzing the element covering performance of the greedy algorithm for Minimum Set Cover during its execution, one can obtain reasonably tight performance bounds for the $p$-Partial Cover problem. The bound that we obtain for a ground set of $m$ elements is the harmonic bound $H(\lceil pm \rceil)$, which

T. Elomaa et al. (Eds.): Ukkonen Festschrift, LNCS 6060, pp. 102–113, 2010.

is the best known performance guarantee for the weighted PARTIAL COVER problem [13]. A somewhat tighter bound is known to hold for the unweighted version of the problem [14].

Our analysis asks how large portion of the elements in the ground set can be covered by using at most $k$ subsets. We analyze the relation between the number of covered elements when the subsets are selected greedily and that when the subsets are chosen optimally. We then apply this relationship to the PARTIAL COVER problem to obtain the harmonic bound.

The remainder of this paper is organized as follows. In Section 2 we briefly review work on MINIMUM SET COVER together with its variants and recapitulate the greedy algorithm for PARTIAL COVER. The element covering analysis for the greedy algorithm is presented in Section 3 and its application to partial covers is the topic of Section 4. We consider possibilities to extend this approach further in Section 5. A brief survey of related work is given in Section 6 before concluding this paper in Section 7.

## 2 Minimum Set Cover and the Greedy Algorithm

A collection $S = \{ S_1, \ldots, S_n \}$ of subsets of some finite set $U$ is a *cover* of $U$ if $\bigcup_{i=1}^{n} S_i = U$. Moreover, $S' \subseteq S$ is a *subcover* of $U$ if $S'$ itself is a cover of $U$. In the classical MINIMUM SET COVER problem one is given as an instance a finite set $U$ and a cover $S = \{ S_1, \ldots, S_n \}$ of $U$ and is requested to find a subcover $S' \subseteq S$ of $U$ of minimum cardinality. To put this more exactly, in terms of the approximation setting MINIMUM SET COVER problem is as follows:

**Instance:** A cover $S = \{ S_1, \ldots, S_n \}$ of $U$.
**Solution:** A subcover $S' \subseteq S$ of $U$.
**Measure:** Cardinality of the subcover, $|S'|$.

In the decision version of this problem one asks whether there exists a subcover of cardinality at most $K$. This problem was shown to be NP-complete by Karp [1] through a polynomial-time reduction from the VERTEX COVER problem. Throughout this paper we denote the cardinality of the ground set $U$ by $m$.

The greedy algorithm for the set cover problem is one of the best-known polynomial-time approximation algorithms. It chooses at each step the unused set which covers the largest number of remaining elements. This algorithm was shown by Johnson [3] and Lovász [4] to have approximation ratio no worse than $H(m)$, where $H(m) = 1 + 1/2 + \cdots + 1/m$ is the $m$th harmonic number. Recall that $\ln m < H(m) \leq \ln m + 1$. Chvátal [5] extended the harmonic performance ratio also to the weighted version of MINIMUM SET COVER. This time the greedy selection picks at each step the covering set with the minimum cost per remaining element.

Feige [15] proved — using interactive proof techniques — that no polynomial time algorithm can approximate MINIMUM SET COVER within $(1 - \epsilon) \ln m$ for any $\epsilon > 0$, unless NP $\subseteq$ DTIME($n^{\log \log n}$). Hence, under this plausible structural complexity assumption, the performance ratio of any polynomial time algorithm

can improve on the harmonic bound of the greedy algorithm by at most $o(\ln m)$. The analysis of the greedy algorithm for Minimum Set Cover was essentially completed by Slavík [14,16] who proved a performance ratio of exactly $\ln m - \ln\ln m + \Theta(1)$ for the algorithm. More precisely

$$\ln m - \ln\ln m - 0.31 < |G|/|0| < \ln m - \ln\ln m + 0.78,$$

where $G$ is the cover selected by the greedy algorithm and $O$ is the optimal cover.

The proof technique of Slavík was to recursively define the "greedy numbers" $N(k, l)$, which correspond to the size of the smallest ground set $U$ for which it is possible to have a cover of $U$ with the optimal cardinality $l$ and greedy cover of size $k$. The same technique can be adapted to also apply for fractional [4,17] and partial covers [6].

Two natural variations of Minimum Set Cover are its weighted version and $d$-Set Cover, where all members of the cover $S$ have cardinality of at most $d$. In the weighted Minimum Set Cover all elements of $S$ have a positive cost associated with them and the goal is to find a subcover of minimum total cost. Both variations, naturally, are NP-complete, since they contain the original problem as a special case. The greedy algorithm also approximates these problems within the harmonic bound in polynomial time [3,4,5].

A somewhat more general NP-hard set covering problem is Partial Cover [6]. We say that $S' \subseteq S$ is a *$p$-partial cover* of $U$ if

$$\left| \bigcup_{S_j \in S'} S_j \right| \geq pm.$$

An instance of the Partial Cover problem consists of a finite set $U$, a finite cover $S = \{ S_1, \ldots, S_n \}$ of $U$, and a real $p$, $0 < p \leq 1$. The goal is to find a $p$-partial cover $S' \subseteq S$ of $U$ of minimum cardinality:

**Instance:** A cover $S = \{ S_1, \ldots, S_n \}$ of $U$ and a number $p$, $0 < p \leq 1$.
**Solution:** A $p$-partial cover $S' \subseteq S$ of $U$.
**Measure:** Cardinality of the $p$-partial cover, $|S'|$.

Table 1 shows the greedy algorithm for the weighted Partial Cover problem. In it one searches at each step for the unused subset that covers as many elements as possible — though, not excessive elements — with as low average cost per element as possible. When the required fraction of elements covered has been reached, the algorithm halts. Observe that in Step 5 of the algorithm the elements of the newly chosen covering subset are removed from the remaining subsets. Here it has to be done to keep the average cost per element of remaining subsets an informative measure. However, the same cleaning of remaining subsets can be carried out in the unweighted case as well without any harm. In the following we assume that such a cleaning operation is part of the greedy algorithm. The algorithm of Table 1 is very similar to the greedy algorithm for the weighted Minimum Set Cover problem [5,6].

**Table 1.** The greedy algorithm for the weighted Partial Cover

---

**Input:** A cover $S = \{ S_1, \ldots, S_n \}$ of a finite set $U$, positive costs $c = \{ c_1, \ldots, c_n \}$ of the covering sets, and a number $p$.
**Output:** A $p$-partial cover $S' \subset S$ of $U$.

1. $S' \leftarrow \emptyset$.
2. Find out the number $r$ of elements of $U$ that still need to be covered in order to obtain a $p$-partial cover:
$$r \leftarrow \lceil pm \rceil - \left| \bigcup_{S_j \in S'} S_j \right|.$$
3. If $r \leq 0$, then return $S'$.
4. Find $S_i \in S \setminus S'$, $S_i \neq \emptyset$, that minimizes the quotient
$$\frac{c_i}{\min(r, |S_i|)}.$$
5. $S' \leftarrow S' \cup S_i$.
For each $S_j \in S \setminus S'$: $S_j \leftarrow S_j \setminus S_i$.
Go to step 2.

---

The straightforward analysis of the greedy method for Partial Cover becomes quite complicated because the optimal solution may cover a different set of elements than those chosen by the greedy algorithm. Thus, the methods used by Johnson [3], Lovász [4], and Chvátal [5] to establish the harmonic bound in case of complete covers do not generalize directly to this problem.

Nevertheless, Kearns [6] managed to prove the weak harmonic performance guarantee of $2H(m) + 3$ for the greedy algorithm by bounding separately the weights of those elements that are covered by the greedy algorithm but do not belong to the optimal cover, and those that are members of both solutions. Using a completely different approach Slavík [13] proved that for the weighted $p$-partial cover problem a bound similar to the classical one holds: The performance ratio of the greedy algorithm for this problem is no worse than $H(\lceil pm \rceil)$. One can construct an example to show that this bound is also tight [13]. The bound contains, as special cases, the classical harmonic bounds for Minimum Set Cover.

This time the proof technique of Slavík was to contrast directly the weights of the optimal and greedy cover from iteration to iteration in the execution of the greedy algorithm. Slavík's [14] exact analysis of the Minimum Set Cover problem also holds for Partial Cover when the subsets are unweighted. Thus, unweighted Partial Cover can be approximated using the greedy algorithm with ratio $\ln\lceil pm \rceil - \ln\ln\lceil pm \rceil + \Theta(1)$.

Subsequent Minimum Set Cover approximation approaches — aiming to improve additive constants, which is the most one can hope for after Feige's [15] proof — include Srinivasan's [17] application of the randomized rounding

technique [18] to obtain improved performance ratio in special cases. Another line of research has been the work of Halldórsson [19,20] who applied a *local improvements* modification to the greedy algorithm to obtain an improved upper bound of $H(m) - 0.43$. In this approach one applies optimization techniques to the subsets that are small enough. This approach was taken further by Duh and Fürer [21] in their *semi-local optimization* approach. This leads to the polynomial-time approximation algorithm with the best worst-case performance guarantee of $H(m) - 1/2$.

Slavík's [13] proof of the harmonic bound is based on an analysis of the cost per remaining relevant element of a subset chosen to the greedy partial cover. Unfortunately, this does not lead to an intuitive proof. In the following we show that the harmonic performance guarantee can be obtained directly through an analysis of the greedy algorithm for Minimum Set Cover.

The greedy algorithm works in rounds choosing the subsets to the evolving cover one by one. Hence, it is natural to consider how many elements are covered by the greedy algorithm after $r$ rounds and compare it to the optimal covering in $k$ rounds. Viewing the greedy algorithm as gradually covering more and more elements gives a concrete connection between its performance in the two problems. The analysis easily yields the harmonic bound for Partial Cover. This simplified proof is of interest because of the importance and wide use of Minimum Set Cover and its generalization.

## 3 Covering Analysis of the Greedy Algorithm

Let us analyze the greedy set covering algorithm from the point of view of its covering performance. We show that the number of elements covered by $r$ greedily chosen subsets is not much less than the total number of elements in $k$, $k \leq r$, optimally chosen sets of $S$. Here the $r$ greedily or $k$ optimally chosen subsets do not have to constitute a cover for the whole of $U$. However, setting $r$ and $k$ large enough, will eventually yield a full cover of $U$.

Let $g_i$ denote the size of the subset chosen by the greedy algorithm on the $i$th round and let $G_r = \sum_{i=1}^{r} g_i$. The maximum number of elements covered by $k$ optimally chosen subsets is denoted by $O_k$.

**Lemma 1.** *For $r \geq k \geq 1$ the following holds*

$$G_r \geq \left(1 - \left(1 - \frac{1}{k}\right)^r\right) O_k.$$

*Proof.* By the pigeonhole principle the largest subset $g_1$, which is chosen by the greedy algorithm on the first round, must contain at least as many elements as there on average are in the $k$ maximally-covering sets, for any $k$; i.e., $g_1 \geq O_k/k$. The pigeonhole principle applies also on the second and subsequent rounds. However, $g_2$ can only be guaranteed to have size $(O_k - g_1)/k$. In general, on round $n+1$, $n \geq 1$, one must reduce the number of elements in the subsets already

chosen by the greedy algorithm on previous rounds, $G_n$, from the maximum number of elements covered by $k$ subsets and we have

$$g_{n+1} \geq \frac{O_k - G_n}{k}. \tag{1}$$

Let us, thus, consider the sequence

$$\begin{cases} x_1 = O_k/k; \\ x_{n+1} = x_n - (x_n/k) = (1 - 1/k)\, x_n, \end{cases}$$

for which it holds

$$x_n = \left(1 - \frac{1}{k}\right)^{n-1} \frac{O_k}{k}. \tag{2}$$

By induction we can show that $G_n \geq \sum_{i=1}^{n} x_i$. The base case was already stated above. Let us, then, assume that the claim holds for values less than $n$. Now, by inequality (1), the inductive hypothesis, and the definition of the sequence, we get

$$\begin{aligned} G_{n+1} &= G_n + g_{n+1} \\ &\geq G_n + \frac{O_k - G_n}{k} \\ &= \frac{O_k}{k} + \left(1 - \frac{1}{k}\right) G_n \\ &\geq \frac{O_k}{k} + \left(1 - \frac{1}{k}\right) \sum_{i=1}^{n} x_i \\ &= \frac{O_k}{k} + \sum_{i=1}^{n} \left(1 - \frac{1}{k}\right) x_i \\ &= \frac{O_k}{k} + \sum_{i=1}^{n} x_{i+1} \\ &= x_1 + \sum_{i=2}^{n+1} x_i \\ &= \sum_{i=1}^{n+1} x_i. \end{aligned}$$

Combining this with equality (2) gives the lower bound for $G_r$:

$$\begin{aligned} G_r &\geq \sum_{i=1}^{r} x_i \\ &= \sum_{i=1}^{r} \left(1 - \frac{1}{k}\right)^{i-1} \frac{O_k}{k} \\ &= \left(1 - \left(1 - \frac{1}{k}\right)^{r}\right) O_k, \end{aligned}$$

where the last equality is by the value of a geometric series.

Particularly interesting special case is $r = k$, which corresponds to asking what is the performance guarantee of the greedy algorithm with respect to the number of covered elements. This question has independently been studied by Hochbaum and Pahtria [22,23], who also obtained the following results.

By the above result, for example, two greedily chosen subsets are guaranteed to cover at least 3/4 of the largest number of elements that can, on the whole, be covered by two subsets. Asymptotically the lower bound behaves as follows when $r = k$. One can simplify the above result by recalling that for all $x$

$$\lim_{n\to\infty} \left(1 + \frac{x}{n}\right)^n = e^x$$

and observing that $(1-(1-1/k)^k)$ is decreasing, which gives the greedy algorithm the following approximation guarantee in the number of elements covered:

$$G_k \geq \left(1 - \frac{1}{e}\right) O_k.$$

## 4 Application of the Analysis to Partial Cover

Let APP now denote the number of subsets chosen by the greedy algorithm in order to cover at least a fraction $p$ of the $m$ elements in the ground set. Respectively, OPT is the minimum number of subsets required to cover at least proportion $\lceil pm \rceil$ of the elements.

**Theorem 1.** $\text{APP}/\text{OPT} \leq H(\lceil pm \rceil)$.

*Proof.* Without loss of generality, we can assume that $O_{\text{OPT}} = \lceil pm \rceil$. This can be accomplished by removing some of the elements from the sets belonging to the optimal solution. The remaining elements still constitute a $p$-partial cover of $U$ and, therefore, the value of the optimal solution does not change. On the other hand, this modification cannot improve the solution of the greedy algorithm.

Observe that the case $\text{OPT} = 1$ is not interesting, because the greedy algorithm will also output the one subset that covers a fraction $p$ of $U$. Hence, in this case $\text{APP} = \text{OPT}$. In the following we consider only partial covers for which $\text{OPT} \geq 2$.

Let us consider the least $r$ such that $G_r \geq O_{\text{OPT}} - c$ for some constant $c$. Thus, by Lemma 1, we want to solve

$$\left(1 - \left(1 - \frac{1}{\text{OPT}}\right)^r\right) O_{\text{OPT}} = O_{\text{OPT}} - c \Leftrightarrow$$
$$-\left(1 - \frac{1}{\text{OPT}}\right)^r = -\frac{c}{O_{\text{OPT}}}.$$

Taking natural logarithms of both sides gives

$$-r \ln\left(1 - \frac{1}{\text{OPT}}\right) = \ln O_{\text{OPT}} - \ln c. \tag{3}$$

Recalling that for $x > -1 : \ln(1+x) \leq x$, where equality holds only for $x = 0$, leads to

$$\frac{r}{\text{OPT}} < \ln O_{\text{OPT}} - \ln c. \tag{4}$$

The $r$ greedily selected subsets now cover $O_{\text{OPT}} - c$ elements. Thus, at most $c$ further subsets are needed to cover in total at least $O_{\text{OPT}}$ elements. Hence, $\text{APP} \leq \lceil r \rceil + c \leq r + 1 + c$, and by inequality (4) we have

$$\begin{aligned}\frac{\text{APP}}{\text{OPT}} &\leq \frac{r+c+1}{\text{OPT}} \\ &< \ln O_{\text{OPT}} - \ln c + \frac{c+1}{\text{OPT}}.\end{aligned}$$

The right-hand side obtains its minimum value when $c = \text{OPT}$, which further yields

$$\frac{\text{APP}}{\text{OPT}} \leq \ln O_{\text{OPT}} - \ln \text{OPT} + 1 + \frac{1}{\text{OPT}}.$$

Finally, because $\ln n > 1 + 1/n$ for all integers $n \geq 4$, we only need to check separately the cases $\text{OPT} = 3$ and $\text{OPT} = 2$ by substituting them and $c = 1$ to equation (3), to obtain the desired result

$$\frac{\text{APP}}{\text{OPT}} \leq \ln O_{\text{OPT}} < H(O_{\text{OPT}}) = H(\lceil pm \rceil).$$

The above derived harmonic bound $H(\lceil pm \rceil)$ is the tight bound for *weighted* Partial Cover [13], but Slavík's [14] greedy numbers technique yields the tight bound $\ln m - \ln \ln m + \Theta(1)$ for the *unweighted* problem. He obtains the tighter bound through a detailed analysis of the involved functions.

## 5 Further Application of the Analysis

Above we were eager to approximate the additive terms away in order to reach the harmonic bound for Partial Cover. Let us briefly consider what happens if they are not abstracted away.

Let

$$d = \ln \left(1 - \frac{1}{\text{OPT}}\right)^{-\text{OPT}}$$

and substitute it to equation (3) to obtain

$$\frac{rd}{\text{OPT}} = \ln O_{\text{OPT}} - \ln c.$$

Recalling that $\text{APP} \leq r + c + 1$ yields

$$\begin{aligned}\frac{\text{APP}}{\text{OPT}} &\leq \frac{r+c+1}{\text{OPT}} \\ &= \frac{\ln O_{\text{OPT}} - \ln c}{d} + \frac{c+1}{\text{OPT}}.\end{aligned}$$

The right-hand side of this inequality will obtain its minimum value when $c = \text{OPT}/d$. Thus,

$$\frac{\text{APP}}{\text{OPT}} \leq \frac{\ln O_{\text{OPT}}}{d} + \frac{1 + \ln d - \ln \text{OPT}}{d}. \tag{5}$$

This bound has the unfortunate property of not being independent of the value of OPT. Let us, thus, denote by $d_{\text{OPT}}$ the value of $d$ depending on the value of OPT. For instance, $d_2 = \ln 4 \approx 1.386$ and asymptotically $d_\infty = 1$.

How does the bound (5) behave in comparison to the harmonic one derived above? Assuming that $\ln O_{\text{OPT}}$ is large, for small values of OPT bound (5) will be tighter than the harmonic one. However, $d_{\text{OPT}}$ approaches one as the value of OPT increases and, thus, this bound eventually loses its advantage over the harmonic bound. Moreover, as the value of OPT is unknown, this performance guarantee is not a very practical one.

## 6 Related Work

The covering analysis of the greedy set covering algorithm (Lemma 1) has been settled in case $r = k$ by Hochbaum [22,23,24]. The result is relatively well known as Max Cover. The derivation of this result does not differ significantly from that given in this paper.

Slavík was not the first author to show the tight harmonic approximation bound for partial cover. This result is already contained in the more general result of Wolsey from 1982 [25], although it is not a widespread fact.

The direct link between the covering analysis of the greedy set covering algorithm and its performance for partial cover (Theorem 1) is, to the best of our knowledge, an original contribution.

The cover of Minimum Set Cover $S = \{ S_1, \ldots, S_n \}$ can be seen as a hypergraph over the vertices from the ground set $U$. Its dual problem is Minimum Vertex Cover (VC) over the dual of this hypergraph, which inverts the roles of hyperedges and vertices. The greedy approach and other algorithms for different variations of the partial VC problem have been studied extensively in recent years [26,27,28,29].

Several new variants of the Minimum Set Cover problem have been proposed and analyzed lately. Let us just mention a few of them. In the *red-blue* set cover [30] the ground set $U$ contains red and blue elements and the aim is to cover all of the blue elements and as few as possible of the red elements. This is a strongly inapproximable problem [31]. A generalization of the red-blue set cover, *positive-negative* partial set cover was introduced by Miettinen [32]. In *multicover* problems the requirement is to cover each element a prescribed number of times. Also in this extension of Minimum Set Cover the greedy algorithm and its variants yield good approximation results [33,34,35].

## 7 Conclusion

We have shown that covering analysis of the greedy algorithm for the Minimum Set Cover problem quite easily yields the harmonic bound $H(\lceil pm \rceil)$ for the

$p$-Partial Cover. This makes the connection between the classical problem and its generalization explicit. The obtained bound is not the tightest one known to hold for the unweighted problem. Nevertheless, it is clearly better than the one that comes out of the analysis that bounds the sizes of the resulting sets.

As future work we leave studying whether the tighter performance guarantee for the unweighted Partial Cover could be reached by means of covering analysis. Also, the potential of this line of analysis for the weighted Partial Cover was not explored in this work.

## Acknowledgements

This work has been supported by the Academy of Finland.

## References

1. Karp, R.M.: Reducibility among combinatorial problems. In: Miller, R., Thatcher, J. (eds.) Complexity of Computer Computations, pp. 85–103. Plenum Press, New York (1972)
2. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman & Co, New York (1979)
3. Johnson, D.S.: Approximation algorithms for combinatorial problems. Journal of Computer and System Sciences 9(3), 256–278 (1974)
4. Lovász, L.: On the ratio of optimal integral and fractional covers. Discrete Mathematics 13, 383–390 (1975)
5. Chvátal, V.: A greedy heuristic for the set-covering problem. Mathematics of Operations Research 4(3), 233–235 (1979)
6. Kearns, M.J.: The Computational Complexity of Machine Learning. MIT Press, Cambridge (1990)
7. Kivinen, J., Mannila, H., Ukkonen, E.: Learning hierarchical rule sets. In: Proceedings of the Fifth Annual ACM Conference on Computational Learning Theory, pp. 37–44. ACM Press, New York (1992)
8. Brazma, A., Jonassen, I., Ukkonen, E., Vilo, J.: Discovering patterns and subfamilies in biosequences. In: States, D.J., Agarwal, P., Gaasterland, T., Hunter, L., Smith, R. (eds.) Proceedings of the Fourth International Conference on Intelligent Systems for Molecular Biology, pp. 34–43. AAAI Press, Menlo Park (1996)
9. Brazma, A., Ukkonen, E., Vilo, J.: Discovering unbounded unions of regular pattern languages from positive examples. In: Nagamochi, H., Suri, S., Igarashi, Y., Miyano, S., Asano, T. (eds.) ISAAC 1996. LNCS, vol. 1178, pp. 95–104. Springer, Heidelberg (1996)
10. Li, M.: Towards a DNA sequencing theory. In: Proceedings of the Thirty-First Annual Symposium on Foundations of Computer Science, pp. 125–134. IEEE Computer Society, Los Alamitos (1990)
11. Kivioja, T., Arvas, M., Saloheimo, M., Penttilä, M., Ukkonen, E.: Optimization of cDNA-AFLP experiments using genomic sequence data. Bioinformatics 21(11), 2573–2579 (2005)
12. Rantanen, A., Mielikäinen, T., Rousu, J., Maaheimo, H., Ukkonen, E.: Planning optimal measurements of isotopomer distributions for estimation of metabolic fluxes. Bioinformatics 22(10), 1198–1206 (2006)

13. Slavík, P.: Improved performance of the greedy algorithm for partial cover. Information Processing Letters 64(5), 251–254 (1997)
14. Slavík, P.: A tight analysis of the greedy algorithm for set cover. Journal of Algorithms 25(2), 237–254 (1997)
15. Feige, U.: A threshold of ln n for approximating set cover. Journal of the ACM 45(4), 634–652 (1998)
16. Slavík, P.: Approximation algorithms for set cover and related problems. PhD thesis, State University of New York at Buffalo, Department of Computer Science, Buffalo, NY (April 1998)
17. Srinivasan, A.: Improved approximation of packing and covering problems. In: Proceedings of the Twenty-Seventh Annual ACM Symposium on Theory of Computing, pp. 268–276. ACM Press, New York (1995)
18. Motwani, R., Raghavan, P.: Randomized Algorithms. Cambridge University Press, Cambridge (1995)
19. Halldórsson, M.M.: Approximating set cover via local search. Technical Report IS-RR-95-0002F, Japan Advanced Institute of Science and Technology (1995)
20. Halldórsson, M.M.: Approximating discrete collections via local improvements. In: Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 160–169. ACM Press, New York (1995)
21. Duh, R., Fürer, M.: Approximation of $k$-set cover by semi-local optimization. In: Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing, pp. 256–264. ACM Press, New York (1997)
22. Hochbaum, D.S.: Approximating covering and packing problems: set cover, vertex cover, independent set, and related problems. In: Hochbaum, D.S. (ed.) Approximation Algorithms for NP-hard Problems, pp. 94–143. PSW Publishing, Boston (1997)
23. Hochbaum, D.S., Pathria, A.: Analysis of the greedy approach of maximum k-coverage. Naval Research Quarterly 45, 615–627 (1998)
24. Hochbaum, D.S.: Approximation algorithms for the set covering and vertex cover problems. SIAM Journal on Computing 11(3), 555–556 (1982)
25. Wolsey, L.A.: An analysis of the greedy algorithm for the submodular set covering problem. Combinatorica 2(4), 385–393 (1982)
26. Hochbaum, D.S.: The $t$-vertex cover problem: Extending the half integrality framework with budget constraints. In: Jansen, K., Hochbaum, D.S. (eds.) APPROX 1998. LNCS, vol. 1444, pp. 111–122. Springer, Heidelberg (1998)
27. Bar-Yehuda, R.: Using homogeneous weights for approximating the partial cover problem. Journal of Algorithms 39(2), 137–144 (2001)
28. Gandhi, R., Khuller, S., Srinivasan, A.: Approximation algorithms for partial covering problems. Journal of Algorithms 53(1), 55–84 (2004)
29. Sümer, Ö.: Partial covering of hypergraphs. In: Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 572–585. SIAM, Philadelphia (2005)
30. Carr, R.D., Doddi, S., Konjevod, G., Marathe, M.V.: On the red-blue set cover problem. In: Proceedings of the Eleventh ACM-SIAM Symposium on Discrete Algorithms, pp. 345–353. SIAM, Philadelphia (2000)
31. Peleg, D.: Approximation algorithms for the label-cover$_{\max}$ and red-blue set cover problems. J. Discrete Algorithms 5(1), 55–64 (2007)

32. Miettinen, P.: On the positive-negative partial set cover problem. Information Processing Letters 108(4), 219–221 (2008)
33. Fujito, T., Kurahashi, H.: A better-than-greedy algorithm for $k$-set multicover. In: Erlebach, T., Persinao, G. (eds.) WAOA 2005. LNCS, vol. 3879, pp. 176–189. Springer, Heidelberg (2005)
34. Berman, P., DasGupta, B., Sontag, E.D.: Randomized approximation algorithms for set multicover problems with applications to reverse engineering of protein and gene networks. Discrete Applied Mathematics 155(6-7), 733–749 (2007)
35. Chekuri, C., Clarkson, K.L., Har-Peled, S.: On the set multi-cover problem in geometric settings. In: Hershberger, J., Fogel, E. (eds.) Proceedings of the 25th ACM Symposium on Computational Geometry, pp. 341–350. ACM, New York (2009)

# From Nondeterministic Suffix Automaton to Lazy Suffix Tree

Kimmo Fredriksson

Department of Computer Science, University of Eastern Finland,
P.O. Box 1627, 70211 Kuopio, Finland
kimmo.fredriksson@uef.fi

**Abstract.** Given two strings, a pattern $P$ of length $m$ and a text $T$ of length $n$ over some alphabet $\Sigma$ of size $\sigma$, we consider the exact string matching problem, i.e. we want to report all occurrences of $P$ in $T$. The well-known Backward-Nondeterministic-DAWG-Matching (BNDM) algorithm is one of the most efficient algorithm for short to moderate length patterns. In this paper – as a prelude – we take the underlying nondeterministic suffix automaton and apply it to the text instead of to the pattern. The resulting algorithm is surprisingly simple, and efficient for relatively short patterns and small alphabet sizes in practice. We then show how the algorithm can be easily adapted to construct the suffix tree of $T$ in a lazy manner. Both of the algorithms are efficient if the text is static but the patterns are given on-line (without possibility to batch the queries). We discuss various variants of the algorithms, and conclude with some experimental results.

## 1 Introduction

We address the well studied exact string matching problem. The problem is to search the occurrences of the pattern $P = p_0p_1p_2 \dots p_{m-1}$ from the text $T = t_0t_1t_2 \dots t_{n-1}$, where the symbols of $P$ and $T$ are taken from some finite alphabet $\Sigma$ of size $\sigma$. Numerous efficient algorithms solving the problem have been obtained. The first $O(n)$ time algorithm was given in [23], and the first sublinear expected time algorithm in [6]. The sublinearity is obtained by skipping some characters of the input text by *shifting* the pattern over some text positions by using the information obtained by matching only a few characters of the pattern. An average optimal $O(n \log_\sigma(m)/m)$ time algorithm (BDM) is obtained e.g. in [9]. It is also possible to obtain slightly sublinear worst case time [5].

*Bit-parallelism* has been shown to lead to the most efficient algorithms for relatively short patterns, in practice. The first algorithm in this class was Shift-Or [4,35], which runs in time $O(n\lceil m/w \rceil)$ time, where $w$ is the number of bits in computer word. Currently one of the fastest algorithms in practice (for $m \leq w$) is BNDM [27] and its many variants (see e.g. [25,29]). BNDM is bit-parallel version of BDM, and shares its optimal $O(n \log_\sigma(m)/m)$ average case, as well as $O(nm)$ worst case time. This is possible to improve to $O(n)$ in a number of ways [8,2,27,19]. For more references see e.g. [28,10].

T. Elomaa et al. (Eds.): Ukkonen Festschrift, LNCS 6060, pp. 114–129, 2010.

Another line of work is indexing. In this case the text is available for preprocessing, so that the subsequent queries for one or more patterns can be executed efficiently. One such data structure is a *suffix tree*. Suffix tree for the text can be built in $O(n)$ time [34,24,33,12], and then the queries take $O(m + occ)$ worst case time each, where *occ* denotes the number of occurrences. However, the $O(n)$ building (and space) cost is in practice so high that it does not amortize [16] for searching a moderate number of patterns (in such a case e.g. Aho-Corasick automaton [1] is usually a better alternative). One method to alleviate this is to use lazy suffix trees [16], so that the suffix tree is (partially) built as needed.

There are also several *succinct* full text indexes that take space close to the information theoretical minimum. However, the construction can be intricate and have high cost in practice. Various query costs are also higher than for a suffix tree, both in theory and practice. We do not go into the details, the interested reader is referred to [26].

Suffix trees have a myriad of other applications as well [3], e.g. the text book [18] has about 70 pages devoted to the suffix tree applications only.

**Model of Computation.** We assume word RAM model of computation. In this model addressing a memory location and standard arithmetic and bit-wise operations on $O(\log(n))$ bit integers take $O(1)$ time, where $n$ is the input size. Hence the theoretical model imposes that the word length is $w = \Omega(\log(n))$ bits. The practical view is that $w = 32$ or $w = 64$ in current typical CPU architectures, and growing: e.g. the multimedia extensions, such as the widely available SSE instruction set introduced in 1999 with Intel Pentium III have word size of $w = 128$, and the Intel's upcoming AVX extensions will at first double this. Thus it is expected that bit-parallelism will become even more competitive approach in the future. We note also that graphics processing units (GPUs) can offer even higher (bit-)parallelism. Standard desktop computers can contain GPUs having hundreds of processing units (with $w = 32$), and these can be used to get considerable speed-ups for bit-parallel string matching algorithms [11].

We note that the wide word assumption occurs more and more often in algorithmics (outside of string matching) [31,32,22]. This model is called *broad word computation* by D. Knuth [22].

**Our Contributions in Context.** Nondeterministic suffix automaton can be used to recognize all suffixes (and factors, substrings) of the string it is built on [27]. The automaton for a string of length $m$ can be simulated in $O(\lceil m/w \rceil)$ time per input character, by using bit-parallelism. Each state is represented as one bit, and $w$ states can be updated in one shot. Using this to replace the suffix automaton in the BDM algorithm results in a very simple and in practice very efficient BNDM algorithm [27]. The algorithm runs in $O(\lceil m/w \rceil n \log_\sigma(m)/m)$ average time, which is optimal for short patterns ($\lceil m/w \rceil = O(1)$).

In this work we build the nondeterministic suffix automaton for the *text*, and derive two new string matching algorithms. However, applying it to the text one cannot no longer assume that $n = O(w)$, and hence "indexing" the text in this manner does not seem to be a good idea. However, this is only true if one

compares the resulting algorithm against indexing; comparing it against on-line string matching yields a different conclusion.

The first of our algorithms basically just uses the nondeterministic suffix automaton for the text and feeds the pattern to it. We show how this can be simulated efficiently by maintaining only the active states of the automaton. The preprocessing time per a given text is $O(\min(\sigma, m)\lceil n/w \rceil + n)$ and the search for each subsequent pattern then takes $O(\lceil n/w \rceil \log_\sigma(w) + m + occ)$ average and $O(\lceil n/w \rceil m)$ worst case time. We discuss various trade-offs between the preprocessing and search times.

The second algorithm takes the first one a step further. As the patterns are searched we save each of the search states, so that if another pattern with the same prefix is searched later on, we just use the precomputed state. This process can be viewed as a lazy evaluation of the suffix tree of $T$. Another point of view is that the process determinizes (but does not minimize) a nondeterministic suffix automaton (which in turn can be viewed as a "transposed" Shift-And algorithm [4]). We take the suffix tree point of view, as that is what the end result is and it is more useful to the analysis. The preprocessing and worst case search times are as for the first algorithm, as well as the average time for searching the *first* pattern. However, the average search time approaches $O(m + occ)$ as the number of queries grows, so that the construction is amortized over the queries.

Both of the algorithms are simple to implement and have simple main loops that run efficiently in modern processors. Both also have $\lceil n/w \rceil$ terms in their complexities, which should be compared against the "standard" $\lceil m/w \rceil$ terms in most of the bit-parallel string matching algorithms. That is, for relatively short patterns our algorithms parallelize more efficiently. Our experimental results show that these traits together make the algorithms very competitive.

We note that there are other ways to save bits, e.g. for multiple string matching by packing more patterns in a word [20], so that searching $r$ patterns (of any length) takes $O(\lceil r \log_\sigma(w)/w \rceil n + occ)$ average case time. When applicable, this is superior to our approach (the first algorithm). Likewise, when searching a very large pattern set e.g. plain Aho-Corasick achieves $O(n + rm + occ)$ worst case time, which is the same as if first building a suffix tree and then searching each pattern separately. However, traditional suffix tree construction is complicated and slow in practice, and as in [16] we consider the situation where the queries are on-line, so that the searches cannot be batched for multiple matching algorithms. In particular, our algorithms are very efficient for small alphabets and relatively short patterns. Both of these conditions can be somewhat relaxed if enough queries are executed.

## 2 Preliminaries

Let the pattern $P = p_0 p_1 p_2 \ldots p_{m-1}$ and the text $T = t_0 t_1 t_2 \ldots t_{n-1}$ be strings over alphabet $\Sigma = \{0, 1, \ldots, \sigma - 1\}$. The pattern has an occurrence in some text position $j$, if $p_i = t_{j+i}$ for $i = 0 \ldots m - 1$. We want to report all such text positions $j$. String $p_{0 \ldots i}$ is a *prefix* of $P$, string $p_{i \ldots m-1}$ is a *suffix* of $P$, and $p_{i \ldots j}$

is a *substring* (*factor*) of $P$. Any of these can be also an empty string. We use $xy$ to denote the concatenation of strings $x$ and $y$.

Let $w$ denote the number of bits in computer word. We number the bits from the least significant bit (0) to the most significant bit $(w-1)$. C–like notation is used for the bit-wise operations of words; & is bit-wise `and`, | is `or`, $\sim$ negates all bits, << is shift to left, and >> shift to right, both with zero padding. We sometimes use the notation $z_{[i]}$ to denote the $i$th bit of the word $z$.

**Suffix Trie, Tree and Automaton.** A trie [13] is a tree storing a set of strings. Each node of the tree corresponds to a prefix of the (sub)set of the strings. The root node represents an empty string. If no string is a prefix of another string, then each leaf corresponds to exactly one string in the set. Each node has (at most) $\sigma$ children, and the edges are labeled by symbols in $\Sigma$. Thus a path from the root to some node spells out a prefix of a string in the set. Compacted trie is a regular trie except that each unary path is compacted into a single edge, labeled with a string obtained by concatenating all the symbols on the original path. Fig. 1 illustrates both structures.

Suffix trie of a string $T$ is then just a trie storing all the suffixes of $T$, and suffix tree is compacted suffix trie. When building a suffix tree (trie) the string is usually appended with some special symbol that does not occur anywhere else in the string. This guarantees that no suffix is a prefix of another suffix. Hence each leaf of the tree corresponds to exactly one suffix. The labels in suffix tree edges are represented by pointers to the original text string, so that each edge takes only $O(1)$ space and the whole tree takes $O(n)$ space. Finally, suffix automaton (a.k.a. DAWG, Directed Acyclic Word Graph) is basically a trie interpreted as a finite state automaton, and then minimized. DAWG can also be compacted to form a CDAWG.

Given the suffix tree (or trie) for the text $T$, and a pattern $P$, clearly all suffixes that have $P$ as a prefix can be found in $O(m + occ)$ time, where $occ$ is the number of matching suffixes. We note that the actual search cost, as well as the building cost and space depend on how the nodes are stored. For constant size alphabets one can use a table of size $O(\sigma)$ in each node to represent the pointers to children, so that each child can be found in $O(1)$ time. On the other hand, the space grows accordingly. Another solution is to use space only for the children that actually exist, so that it does not depend on $\sigma$, but accessing a child node takes $O(\log(\sigma))$ time. Using perfect hashing addresses both problems,

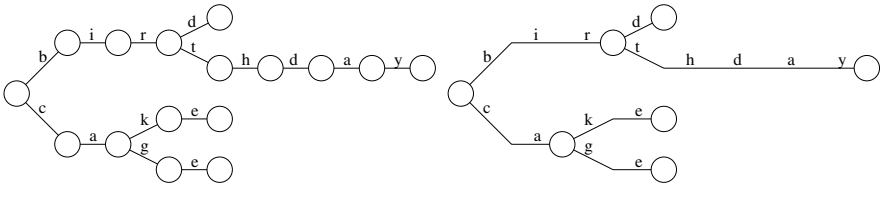



**Fig. 1.** Trie and compacted trie storing strings `birthday, cake, bird` and `cage`

but the construction cost is multiplied by $O(\sigma^2)$ [30]. There are a lot of work to reduce the space in practice, but even the best of them take about $10n$ bytes of space, see e.g. [16] for a brief review.

Although the construction cost can be linear, in practice it is very high, and the algorithms are intricate to implement. One possible solution to both problems is to build the tree in a lazy manner, so that only the parts of the tree that are needed for the queries are built incrementally [16]. This approach is experimentally shown to be superior to the other alternatives. We also take this approach in Sec. 4.

**BNDM.** Backward DAWG Matching algorithm (BDM for short) [8] is an average optimal string matching algorithm. The algorithm needs a method to recognize all factors of the reverse *pattern*, such as a DAWG, or a suffix tree. The algorithm is based on a sliding window of $m$ symbols over the text. The window is scanned backwards with the automaton, recognizing the reverse pattern suffixes. If a suffix of a length $m$ is found, then an occurrence is found. The matching pattern factors also give a powerful method for shifting the window, resulting in $O(n \log_\sigma(m)/m)$ average time, which is optimal [36]. BNDM [27] works exactly as BDM, but the automaton is nondeterministic and is simulated using bit-parallelism. This results in much simpler and more efficient implementation in practice for short patterns. We do not go into the details of the algorithm. The automaton simulation part (backward matching) is basically the same as covered in detail in Sec. 3.

## 3 Basic Algorithm

We now take a different view of the problem. Assume that we had a suffix tree (or suffix automaton) for the text $T$. Then searching pattern occurrences can be done in $O(m + occ)$ time. If the suffix tree is not available, it can be built in $O(n)$ time when needed, so that the total complexity is $O(n + m + occ)$. This method can also be implemented bit-parallely, which leads into an interesting hybrid between on-line searching and indexing. That is, to search the pattern occurrences, we first build a *nondeterministic* suffix automaton (or tree) for the *text*, and then simulate the standard suffix tree traversal bit-parallely to search the pattern occurrences.

The automaton is similar to that in BNDM. We have *states* $q_0 \dots q_n$, and there is a transition from state $q_i$ to state $q_{i+1}$ with the character $t_i$, denoted as $q_i \xrightarrow{t_i} q_{i+1}$. In addition, we have an initial state ('R', *root*) that has an $\varepsilon$-transition to every other state. The state $q_n$ is the accepting state. Again, iff the state $q_n$ is active, then some suffix of $T$ is recognized (including an empty string). Fig. 2 illustrates.

This automaton can be easily simulated with bit-parallelism if $n \leq w$. This assumption is obviously unreasonable, but we first make it to simplify the presentation, and then show the unrestricted version. The simulation is basically as in standard Shift-And algorithm [4]. The preprocessing algorithm builds a

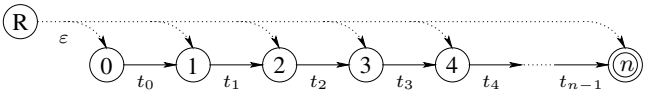



**Fig. 2.** Non-deterministic automaton recognizing the suffixes of the text

table $B$, having one bit-mask entry for each $c \in \Sigma$. For $0 \leq i \leq n-1$, the mask $B[c]$ has $i$th bit set to 1, iff $t_i = c$, i.e. $B[t_i]_{[i]} \equiv q_i \xrightarrow{t_i} q_{i+1}$. We also need a bit-vector $D$ of $n$ bits for the states of the automaton: iff $D_{[i]} = 1$ then the state $q_i$ is active. Note that this does not include the state 0, which will be handled implicitly. Initially each bit is set to 1 to simulate the $\varepsilon$-transitions from root state R. The automaton is simulated in two steps as follows. For each subsequent pattern symbol $p_i$ the vector is first updated by the formula

$$D \leftarrow D \;\&\; B[p_i].$$

The & operation leaves every 1 bit in $D$ to 1 iff there was a corresponding transition with character $p_i$ in the automaton. Thus for each surviving 1 bit the next state should be activated, which can be simply done using the shift operation:

$$D \leftarrow D << 1.$$

Hence we can simulate suffix tree traversal with the automaton, by executing the simulation step for symbols $p_0 \ldots p_{m-1}$, or until $D$ runs out of 1 bits (active states). If after the last step $D$ is not all zeros, then each 1 bit in $D$ indicates a pattern occurrence. That is, iff $D_{[i]} = 1$, then $P$ occurs in $T[i-m \ldots i-1]$. Alg. 1 shows the complete pseudo code. This is much simpler than BNDM, and in fact can be seen as a particular implementation of Shift-And algorithm.

### 3.1 The Real Algorithm

If $n \leq w$, then the above algorithm runs in $O(\sigma + n + m)$ worst case time. As per usual, it is quite easy to simulate longer machine words by simply allocating $\lceil n/w \rceil$ words and doing the & and << operations in $\lceil n/w \rceil$ steps. The algorithm then runs in $O(\sigma\lceil n/w \rceil + n + \lceil n/w \rceil m)$ worst case time. This should be contrasted to the $O(n\lceil m/w \rceil)$ worst case time of BNDM[1]; i.e. if $m \ll w$, then much of the parallelism is effectively lost. However, the search complexity $O(\lceil n/w \rceil m)$ assumes that each $w$-bit piece of $D$ is needed at each step. In practice most of the pieces become all zeros after a few (we make this precise shortly) steps. To this end, let us define an ordered set $L$:

**Definition 1.** $L = \{j \mid D[j] \neq 0 \text{ OR } D[j-1]_{[w-1]} \neq 0\}$.

Here $D[j]$ is the $j$th $w$-bit piece of the vector $D$. The set $L$ thus contains the indexes of the pieces that need to be updated in the next step of the simulation.

[1] The "text-book" implementation actually runs in $O(nm\lceil m/w \rceil)$ worst case time, but this can be improved with little additional complicacy [27,19].

The rationale is that if $D[j] = 0$, its value cannot change unless the highest bit of $D[j-1]$ is set to 1, as the shift operation may bring that bit (if it survives the & operation) into the lowest bit of $D[j]$.

The preprocessing step initializes $L$ to $\{0 \dots \lceil n/w \rceil - 1\}$ in $O(\lceil n/w \rceil)$ time. Then in each simulation step the & and << operations update only the pieces $D[j]$ such that $j \in L$. The time is thus $O(\ell)$ per step, where $\ell = |L|$. After the simulation step, the set $L$ must be updated, which is easy to do in $O(\ell)$ time. The code can be somewhat simplified by assuming that $m \leq w$. As is shown later, our algorithm is in any case competitive only for relatively short patterns. Consider now an alternative definition for $L$:

**Definition 2.** $L = \{j \mid D[j] \neq 0 \text{ OR } D[j-1] \neq 0\}$.

The assumption $m \leq w$ means that no 1-bit can be shifted more than $w$ steps, and together with the looser definition of the set $L$ it follows that $\ell$ can only decrease during the simulation, which in turn allows (slightly) simpler and more efficient set updating. The drawback is that now $L$ may contain some $j$ such that $D[j]$ cannot change in the next simulation step (but it is possible that it will change in later steps). We will later denote the search state by the pair $(L_s, D_s)$, i.e. $L$ and $D$ after the prefix $s$ of $P$ has been processed. The initial state then corresponds to $(L_\varepsilon, D_\varepsilon)$. Alg. 5 shows the complete pseudo code implementing everything described above. The worst case time is not affected.

### 3.2 Final Touches

Let us look at the preprocessing. Computing the table $B$ costs $O(\sigma\lceil n/w \rceil + n)$ time. The first term comes from the need to clear $\sigma$ bit-vectors. However, for large alphabets many of these vectors may also remain all zeros, and hence need not be explicitly represented. That is, the vectors need to be computed only for alphabet symbols that actually occur in the pattern. If some text symbol $c$ does not occur in the pattern, we immediately know that $B[c]$ is never accessed, and thus need not to be initialized. This brings the preprocessing cost down to $O(\min(\sigma, m)\lceil n/w \rceil + n + m)$. Another simple observation is that if we forget that improvement, then the preprocessing does not depend on the pattern at all, and thus the preprocessing needs to be done only once per given text, and its possibly high cost is quickly amortized.

The linear $O(n)$ term of the preprocessing comes from scanning the text once and setting one bit to 1 in $B$ for each text symbol. It is possible to parallelize this work somewhat by manipulating the bit patterns of the symbols with some bit trickery. This would give $O(\min(\sigma, m)\lceil \log(\sigma) n/w \rceil)$ time, which can be faster than the simple method for small alphabets. However, for constant size alphabets both are asymptotically the same. We omit the details; similar technique can be found in [14]. Yet another method is suggested in Sec. 4.2.

Interestingly, it is also possible to avoid the whole preprocessing. The trick is (as already observed in [27]) to treat $P$ and $T$ as binary vectors of lengths $m\lceil \log_2(\sigma) \rceil$ and $n\lceil \log_2(\sigma) \rceil$, respectively. Hence we have reduced the alphabet

size to $\sigma = 2$ by making $P$ and $T$ longer by a factor of $\lceil \log_2(\sigma) \rceil$. The real benefit is that now by definition $B[1] = T$, and $B[0] = \sim T$, and thus we do not need any preprocessing. In other words $T$ (in binary form) implicitly represents its own suffix automaton. The search algorithm is not affected, except that we accept only well aligned matches, i.e. every real occurrence must start in bit position of the form $i\lceil \log_2(\sigma) \rceil$.

### 3.3 Average Case Time

Consider now the average case time of Alg. 5. To this end, we assume uniformly random text. This model is reasonably good e.g. for DNA and protein sequences. For illustrative purposes we present the analysis by drawing parallels between a suffix trie and the nondeterministic suffix automaton of the text. We relate the search process in a suffix trie and nondeterministic suffix automaton by their state: if we have matched a pattern prefix $s$, we have ended up in some node $v$ in the trie, such that the path from the root to $v$ spells out $s$; likewise the suffix automaton is in state $(L_s, D_s)$. We make this connection more explicit in Sec. 4.

In the uniform model, (roughly) all strings of length $\leq h$ (or equivalently, all nodes at depth $\leq h$) exist in the suffix trie, for $h = \log_\sigma(n)$. In other words, the number of nodes at depth $i$ in the trie is $\Theta(\min(\sigma^i, n))$. The root node of the trie corresponds to all suffixes of the text, which in our nondeterministic automaton means the search state $(L_\varepsilon, D_\varepsilon)$, and $|L_\varepsilon| = \lceil n/w \rceil$. The $\sigma$ children of the root node each correspond to approximately $n/\sigma$ suffixes, their children to $n/\sigma^2$ suffixes, and so on, until depth $h$, where only one suffix remains. That is to say that for a search state $(L, D) = (L_{p_0 \ldots p_i}, D_{p_0 \ldots p_i})$, $D$ has $\Theta(n/\sigma^i)$ bits set to 1, assuming that $i \leq h$. However, $|L|$ is not always decreased even if the number of set bits decrease, as zeroing a particular bit may not make the corresponding word all zeros. The number of set bits per word of $D$ decreases to $O(1)$ after $O(\log_\sigma(w))$ steps. The search complexity of the algorithm up to this is therefore $O(\lceil n/w \rceil \log_\sigma(w))$. After this, $|L|$ decreases exponentially, and summing up, the rest of the search takes at most

$$O\left(\sum_{i=\log_\sigma(w)}^{\log_\sigma(n)} \lceil n/w \rceil \;/\; \sigma^{i-\log_\sigma(w)}\right) \quad = \quad O(\lceil n/w \rceil).$$

time on average. To summarize, we have obtained:

**Theorem 1.** *Alg. 5 takes $O(\sigma\lceil n/w \rceil + n + r(\lceil n/w \rceil \log_\sigma(w) + m) + occ)$ expected time to search $r$ patterns.*

## 4 Lazy Suffix Tree

As already mentioned in Sec. 3.2 we can obtain a rudimentary indexing algorithm by noticing that the preprocessing is needed only once per a given text. However,

**Alg 1.** NDIMa$(T, n, P, m)$ ▶ Assumes that $n \leq w$

```
1  for i ← 0 to σ − 1 do B[i] ← 0
2  for i ← 0 to n − 1 do B[t_i] ← B[t_i] | (1 << i)
3  D ← ∼0; i ← 0
4  while i < m AND D ≠ 0 do
5      D ← (D & B[p_i]) << 1
6      i ← i + 1
7  for i ← 0 to n − 1 doif D & (1 << i) ≠ 0 then report occurrence at i − m
```

**Alg 2.** SparseAnd$(D, B, L, \ell)$

```
1  for i ← 0 to ℓ − 1 do D[L[i]] ← D[L[i]] & B[L[i]]
```

**Alg 3.** SparseShl$(D, L, \ell)$

```
1  if L[0] ≠ 0 then z ← 0; else z ← 1
2  for i ← ℓ − 1 downto z do D[L[i]] ← (D[L[i]] << 1) | (D[L[i] − 1] >> (w − 1))
3  if z = 1 then D[0] ← D[0] << 1
```

**Alg 4.** UpdateList$(D, L, \ell)$

```
1  if D[L[0]] ≠ 0 then j ← 1 else j ← 0
2  for i ← 1 to ℓ − 1 doif (D[L[i]] | D[L[i] − 1]) ≠ 0 then {L[j] ← L[i]; j ← j + 1}
3  ℓ ← j
```

**Alg 5.** NDIM$(T, n, P, m)$

```
1  if the text T is not seen yet then
2      for i ← 0 to σ − 1 dofor j ← 0 to ⌈n/w⌉ − 1 do B[i][j] ← 0
3      for i ← 0 to n − 1 do B[t_i][⌊i/w⌋] ← B[t_i][⌊i/w⌋] | (1 << (i mod w))
4  for j ← 0 to ⌈n/w⌉ − 1 do { D[j] ← ∼0; L[j] ← j }
5  i ← 0; ℓ ← ⌈n/w⌉
6  while i < m AND ℓ ≠ 0 do
7      SparseAnd(D, B[p_i], L, ℓ)
8      SparseShl(D, L, ℓ)
9      UpdateList(D, L, ℓ)
10     i ← i + 1
11 for i ← 0 to ℓ − 1 do for each set bit in D[L[i]] report occurrence
```

Alg. 5 can be easily adapted to compute the suffix trie (or tree) of $T$, and we do it lazyly, i.e. build only the nodes of the tree that are actually needed when searching the patterns. The method was already hinted in Sec. 3.3. We describe the method for building the suffix trie. Algorithm for suffix tree follows easily.

As the preprocessing step, we compute the (global) table $B$ and initialize the root node. Each node has one outgoing edge for each alphabet symbol, and these are initialized to NIL.

**Definition 3.** *A node $v$ is* found *when we compute the edge $u \xrightarrow{c} v$ from an already found node $u$. Root node is found. We call a found node $u$* unevaluated*, if not all the edges to its children are not yet computed, and* evaluated *otherwise.*

Each unevaluated node $u$ (only) stores its state $(L_{\overline{u}}, D_{\overline{u}})$, where $\overline{u}$ denotes the string spelled by the path from the root to $u$. The root then stores $(L_\varepsilon, D_\varepsilon)$.

The search goes as in any suffix trie, i.e. follows the edges corresponding to the pattern symbols. In our case, when matching some symbol $p_i$, we may run into unevaluated node $u$, such that the edge $u \xrightarrow{p_i} v$ is not computed. Such a node stores $(L_{\overline{u}}, D_{\overline{u}})$, and hence the state $(L_{\overline{v}}, D_{\overline{v}})$ for node $v$ can be computed simply as

$$D_{\overline{v}} \leftarrow (D_{\overline{u}} \mathbin{\&} B[p_i]) << 1,$$

and then computing $L_{\overline{v}}$. Both steps can be done in $O(|L_{\overline{u}}|)$ time as detailed in Sec. 3. If $|L_{\overline{v}}| > 0$, then $(L_{\overline{v}}, D_{\overline{v}})$ is stored to the newly found node $v$. Either case, $(L_{\overline{u}}, D_{\overline{u}})$ can be discarded if $u$ became evaluated. We can do this only for $p_i$ ($lazier$ = TRUE in Alg. 11), or for all $\sigma$ alphabet symbols at once ($lazier$ = FALSE). The latter alternative requires $O(\sigma|L_{\overline{u}}|)$ time, but has the benefit that $u$ becomes evaluated in one shot, and hence $(L_{\overline{u}}, D_{\overline{u}})$ can be immediately discarded, reducing the space usage significantly. Note that it is possible to use a hybrid approach too, e.g. to use the latter method near the root only.

As the $(L_{\overline{u}}, D_{\overline{u}})$ pairs need to be stored to all found unevaluated nodes, some care is needed to store them efficiently. In particular, the previous representation stored all $O(\lceil n/w \rceil)$ words of $D$ even if most words were zeros. We fix this as follows. $L_{\overline{u}}$ is defined as previously: $L_{\overline{u}} = \{j \mid D_{\overline{u}}[j] \neq 0 \text{ OR } D_{\overline{u}}[j-1] \neq 0\}$. However, instead of storing $D_{\overline{u}}$, we store $D'_{\overline{u}}$ defined as:

$$D'_{\overline{u}}[i] = D_{\overline{u}}[L_{\overline{u}}[i]],$$

i.e. the non-zero words of $D_{\overline{u}}$ stored consecutively. Mapping bits to suffixes is still simple:

**Lemma 1.** *If $D'_{\overline{u}}[i]_{[j]} = 1$, then $\overline{u} = t_k \dots t_{k+|\overline{u}|-1}$, where $k = L_{\overline{u}}[i] \cdot w + j - |\overline{u}|$.*

With this arrangement, and assuming that $lazier$ = FALSE, the total space for all $(L, D')$ pairs in any phase of the construction is at most $O(n)$. Computing the search state is still simple.

However, if $lazier$ = TRUE, the space can be $O(\lceil n/w \rceil m)$. This can be easily mitigated as follows. First the vector $D_{\overline{v}}$ is computed just as detailed above, given $D_{\overline{u}}$ and $p_i$; then the state $(L_{\overline{u}}, D_{\overline{u}})$ is updated by first computing

$$D_{\overline{u}} \leftarrow D_{\overline{u}} \mathbin{\&} {\sim}B[p_i],$$

and then updating $L_{\overline{u}}$ accordingly. In other words, the suffixes recorded in the found children of $u$ are not duplicated in $D_{\overline{u}}$. I.e. node $u$ and its found children nodes together use at most as much space as the children of $u$ would use when $u$ becomes fully evaluated. This increases the time only by a factor of two. W.l.o.g., for the sequel we assume that $lazier$ = FALSE unless otherwise stated.

### 4.1 From Suffix Trie to Suffix Tree

The above described algorithm computes the suffix trie of $T$, that is, the unary paths are not compacted. First note that handling the case where the unary path leads to a leaf node is easy to handle. Assume that we just found a node $v$ (and the corresponding edge, $u \xrightarrow{c} v$), such that $|L_{\overline{v}}| = 1$, and $D'_{\overline{v}}$ has exactly one bit (let this be the $j$th bit) set to 1. This means that there is only one suffix left (corresponding to node $v$), which in turn means that $v$ is in fact a leaf in a suffix tree (but not necessarily in suffix trie) and hence the edge $u \xrightarrow{c} v$ can be replaced by an edge $u \xrightarrow{cs} v$, where $s = t_i \dots t_{n-1}$, and $i = L_{\overline{v}}[0] \cdot w + j - |\overline{v}|$. The string $cs$ can be represented in a standard way, i.e. buy using a pointer to the text itself.

This technique alone makes the average space complexity $O(n)$, under the same assumptions as in Sec. 3.3. Furthermore, the leaves need not be explicitly represented, and we can make the method even more lazier by just stopping adding new children to any node $u$ when $|L_{\overline{u}}| \leq \mathit{threshold}$, for some $\mathit{threshold} = O(1)$, as it is then possible (and simple) to use just the **while**-loop from Alg. 5 to compute them in $O(1)$ time on the fly. Alg. 11 shows the pseudo code.

Consider now the unary paths that do not end up in a leaf node. These can be also computed in a lazy way as follows. Assume that we have computed an edge $u \xrightarrow{s} v$, for some string $s$. At first $|s| = 1$, and $v$ is unevaluated. Assume that the search has entered the node $v$, and that the next (pattern) symbol to be matched is $c$. Thus one needs to compute the new search state given $(L_{\overline{v}}, D_{\overline{v}})$ and $c$. We have two possibilities: (1) either we need to add a new edge $v \xrightarrow{c} q$ (unless it is there already); or (2) if $c$ is in a unary path, we need to extend the previous edge to $u \xrightarrow{sc} v$. This is easy to notice:

**Lemma 2.** *Iff $D_{\overline{v}} = D_{\overline{v}} \,\&\, B[c]$, then the set of matching suffixes for $\overline{v}$ and $\overline{v}c$ are the same, and $c$ is in unary path.*

This makes the space complexity $O(n)$ in the worst case. The time complexity of the lazy construction depends on the number of searches. The first search takes the same time as Alg. 5 (possibly multiplied by $O(\sigma)$, depending on the variant) on average, but approaches $O(m + occ)$ time in the worst case per search later on. This bound is achieved when the whole tree is evaluated.

Finally, the algorithm can be easily converted to compute the whole suffix tree of $T$ in "eager" manner by pre-order depth-first traversal of the tree, in each node computing all the $\sigma$ children before entering any of them. For the eager construction the time is $O(n \log n)$ on average, but in the worst case $O(n\lceil n/w \rceil)$.

### 4.2 Sparse Suffix Trees

Sparse suffix trees can be used to reduce the space requirements. The algorithm in [21] constructs *evenly spaced* sparse suffix tree, so that the tree stores only every $q$th suffix. Their construction time remains $O(n)$, but the space is reduced to only $O(n/q)$ (for the tree itself; the original text must be kept as well). This space saving comes with a cost: the search becomes more complicated and slower. We

**Alg 6.** SparseStAnd($D, B, L, \ell$)

```
1  for i ← 0 to ℓ − 1 do D[i] ← D[i] & B[L[i]]
```

**Alg 7.** SparseStShl($D, L, \ell$)

```
1  for i ← ℓ − 1 downto 1 do
2      D[i] ← D[i] << 1
3      if L[i] − 1 = L[i − 1] then D[i] ← D[i] | (D[i − 1] >> (w − 1))
4  D[0] ← D[0] << 1
```

**Alg 8.** UpdateStList($D, L, \ell$)

```
1  if D[0] ≠ 0 then j ← 1 else j ← 0
2  for i ← 1 to ℓ − 1 doif D[i] ≠ 0 OR (D[i − 1] ≠ 0 AND L[i] − 1 = L[i − 1]) then
3      L[j] ← L[i]; D[j] ← D[i]; j ← j + 1
4  ℓ ← j
```

**Alg 9.** PartialEvalNode($ST, v, i$)

```
1  v.c[i] ← NIL; D ← v.D; L ← v.L; ℓ ← v.ℓ
2  SparseStAnd(D, ST.B[i], L, ℓ)
3  SparseStShl(D, L, ℓ)
4  UpdateStList(D, L, ℓ)
5  if ℓ > 0 then
6      v.c[i] ← NewStNode
7      v.c[i].L ← L; v.c[i].ℓ ← ℓ; v.c[i].D ← D
8  v.pevaluated[i] ← TRUE
9  if ∀c ∈ Σ : v.pevaluated[c] then
10     v.L ← NIL; v.D ← NIL; v.evaluated ← TRUE
```

**Alg 10.** NewSt($T, n$)

```
1  ST.root ← NewStNode
2  ST.root.ℓ ← ⌈n/w⌉ − 1
3  for i ← 0 to σ − 1 dofor j ← 0 to ST.root.ℓ do ST.B[i][j] ← 0
4  for i ← 0 to n − 1 do ST.B[t_i][⌊i/w⌋] ← ST.B[t_i][⌊i/w⌋] | (1 << (i mod w))
5  for j ← 0 to ST.root.ℓ do { ST.root.D[j] ← ∼0; ST.root.L[j] ← j }
```

**Alg 11.** Lazy($ST, P, m, threshold, lazier$)

```
1  v ← ST.root; i ← 0
2  while i < m AND v ≠ NIL do
3      if v.ℓ ≤ threshold then
4          continue as in Alg. 5 while-loop, using v.L and v.D
5          break
6      if NOT v.pevaluated[p_i] then
7          if lazier then PartialEvalNode(ST, v, p_i)
8          elsefor c ← 0 to σ − 1 do PartialEvalNode(ST, v, c)
9      v ← v.c[p_i]
10     i ← i + 1
11 if v ≠ NIL then report occurrences
```

briefly sketch here another method that has the same space reduction factor, but our construction cost is also reduced by the same factor. The search algorithm becomes slower for a fully built suffix tree, but combined with the lazy evaluation we can get significantly faster *average* search times as well, for a moderate number of patterns. However, the method is applicable only if $(m-q+1)/q \geq 1$ and useful only if $m/q > c \log_\sigma(n)$ for some $c > 1$ and thus can be applied only when the minimum $m$ is known beforehand.

We borrow the idea from [15], used for on-line string matching. Conceptually, we build the tree for the string $T'[i] = T[iq]$, i.e. $T'$ is a subsequence of $T$, containing only every $q$th text symbol. (A somewhat similar idea was used in [7], but their subsequence is based an sampling the alphabet, resulting in irregular subsequences.) Thus $|T'| = \lfloor n/q \rfloor$. Note that this is different from [21]; their suffixes still have $O(n)$ length.

Consider now the search. As $T'$ contains only every $q$th symbol of the original text, we also use only every $q$th symbol of $P$. This means that a matching subsequence must be verified, using the original $P$ and $T$. However, this finds only the matches that are correctly aligned with respect to $q$. Hence we must generate all $q$ possible alignments of $P$, and search each separately. That is, we search patterns of the form $P^j[i] = P[iq+j]$ for $0 \leq j < q$. The following is then immediate:

**Lemma 3.** *If $P$ occurs at $T[i \ldots i+m-1]$ then: (i) $P^j[h] = T[i+j+hq]$, where $j = i \mod q$; and (ii) $P^j$ occurs at $T'[\lceil i/q \rceil \ldots \lceil i/q \rceil + \lfloor m/q \rfloor - 1]$, where $j = q - 1 - (i+q-1) \mod q$.*

As both the text and the pattern (pieces) are shorter than before, the lazy construction will create less nodes, and the nodes are created faster. The result is that doing $q$ searches for patterns of length $O(m/q)$, plus some verifications, is faster than doing one search with a pattern of length $m$, provided that the suffix tree is only partially built, and $m/q$ is large enough. More precisely, the number of verifications per search is on average $(n/q)/\sigma^{m/q}$, the average time per verification is $O(1)$, and we execute $q$ searches. Thus we want to have $q(n/q)/\sigma^{m/q} < 1$, i.e. $q < m/\log_\sigma(n)$. Note also that the combined length of the pieces is $m$, so that for our $q$ the search time tends to $O(m + occ)$.

The method obviously works for Alg. 5 as well. In this case the expected search time is as stated in Theorem 1, when one just substitutes "$n$" with "$n/q$", and taking that $q < m/\log_\sigma(n)$:

**Theorem 2.** *Alg. 5 can be made to run in $O(\sigma\lceil(n\log_\sigma(n)/m)/w\rceil + n\log_\sigma(n)/m + r(\lceil n/w \rceil \log_\sigma(w)+m)+occ)$ expected time to search $r$ patterns, each of length $m$.*

In other words, the preprocessing cost can be reduced while keeping the average search cost the same.

## 5 Preliminary Experimental Results

We have implemented "quick-and-dirty" prototypes of the algorithms in C. We ran the experiments in 3.0GHz Intel Core2 with 2GB RAM, 4MB L2 cache,

**Table 1.** Left: the minimum number of patterns ($r$) for some $m$ where Alg. 5 starts to beat BNDM, using $q = 1$ and the optimal $q$. Right: (time for Alg. 11) / (time for WOTD) $\times$ 100%, to search $r$ patterns of length $m$. Both for 10MB DNA sequence.

| $m$ | 2 | 4 | 8 | 16 | 32 |
|---|---|---|---|---|---|
| $(q = 1)$ $r \geq$ | 1 | 2 | 3 | 7 | 21 |
| (opt $q$) $r \geq$ | 1 | 2 | 2 | 4 | 10 |

| | $r = 10$ | $r = 100$ | $r = 1000$ | $r = 10000$ | $r = 100000$ |
|---|---|---|---|---|---|
| $m = 8$ | 26% | 45% | 77% | 112% | 149% |
| $m = 16$ | 21% | 31% | 47% | 69% | 145% |
| $m = 32$ | 17% | 20% | 27% | 39% | 100% |

running GNU/Linux 2.6.23. We compared against BNDM [27] and WOTD [16] (lazy suffix tree construction, their implementation). As for our algorithms, we implemented the versions given in Alg. 5 and Alg. 11. In particular, we did not implement the advanced preprocessing techniques, and we (effectively) compact only the unary paths leading to the leaves in the suffix trie. On the other hand we implemented the sparse tree technique from Sec. 4.2 (pseudo code not given).

We ran experiments on DNA, protein and English text. The patterns were randomly picked from the text, so each pattern has at least one occurrence. Table 1 summarizes the results for DNA. In general Alg. 5 is more competitive for shorter patterns; for long patterns the preprocessing can always be amortized by doing many enough queries. On the other hand, Alg. 11 is more competitive for long patterns. In general it is better than WOTD when $r/m$ is not "too large". Alg. 11 is never slower than Alg. 5. For proteins and English text Alg. 5 is not very attractive, but Alg. 11 still is; the large alphabet makes the sparse tree technique of Sec. 4.2 useful, and the node evaluation is still efficient if done only partially (parameter *lazier* in Alg. 11). Still it becomes relatively worse as the alphabet grows. We leave a proper implementation and experiments for a future work.

## 6 Final Remarks

Our techniques can be applied to many (bit-parallel) approximate matching algorithms as well. In fact, the approximate matching algorithm under Levenshtein distance in [14] can be seen an example of this (albeit the method was not presented like this). Other possibilities include e.g. the Shift-Add algorithm [4] (or its more efficient variant [17]) for Hamming distance. One could in principle even build a "suffix tree" like structure with it, although the space complexity would be very high, unless some cut-off threshold is used to limit the number of nodes.

**Acknowledgments.** We wish to thank Szymon Grabowski and the anonymous reviewers for several useful comments.

## References

1. Aho, A.V., Corasick, M.J.: Efficient string matching: an aid to bibliographic search. Communications of the ACM 18(6), 333–340 (1975)
2. Allauzen, C., Raffinot, M.: Simple optimal string matching. J. of Algorithms 36, 102–116 (2000)

3. Apostolico, A.: The myriad virtues of suffix trees. In: Apostolico, A., Galil, Z. (eds.) Combinatorial Algorithms on Words. NATO Advanced Science Institutes, Series F, vol. 12, pp. 85–96. Springer, Heidelberg (1985)
4. Baeza-Yates, R.A., Gonnet, G.H.: A new approach to text searching. Communications of the ACM 35(10), 74–82 (1992)
5. Bille, P.: Fast searching in packed strings. In: Kucherov, G., Ukkonen, E. (eds.) CPM 2009. LNCS, vol. 5577, pp. 116–126. Springer, Heidelberg (2009)
6. Boyer, R.S., Moore, J.S.: A fast string searching algorithm. Communications of the ACM 20(10), 762–772 (1977)
7. Claude, F., Navarro, G., Peltola, H., Salmela, L., Tarhio, J.: Speeding up pattern matching by text sampling. In: Amir, A., Turpin, A., Moffat, A. (eds.) SPIRE 2008. LNCS, vol. 5280, pp. 87–98. Springer, Heidelberg (2008)
8. Crochemore, M., Czumaj, A., Gąsieniec, L., Jarominek, S., Lecroq, T., Plandowski, W., Rytter, W.: Speeding up two string matching algorithms. Algorithmica 12(4/5), 247–267 (1994)
9. Crochemore, M., Rytter, W.: Text algorithms. Oxford University Press, Oxford (1994)
10. Crochemore, M., Rytter, W.: Jewels of Stringology. World Scientific, Singapore (2002)
11. Deorowicz, S.: Computing the longest common transposition-invariant subsequence with GPU. In: Proceedings of Man-Machine Interactions, Advances in Intelligent and Soft Computing, vol. 59, pp. 551–559. Springer, Heidelberg (2009)
12. Farach, M.: Optimal suffix tree construction with large alphabets. In: Proceedings of FOCS 1997, pp. 137–143. IEEE, Los Alamitos (1997)
13. Fredkin, E.: Trie memory. Communications of the ACM 3(9), 490–499 (1960)
14. Fredriksson, K.: Row-wise tiling for the myers' bit-parallel approximate string matching algorithm. In: Nascimento, M.A., de Moura, E.S., Oliveira, A.L. (eds.) SPIRE 2003. LNCS, vol. 2857, pp. 66–79. Springer, Heidelberg (2003)
15. Fredriksson, K., Grabowski, S.: Average-optimal string matching. J. Discrete Algorithms 7(4), 579–594 (2009)
16. Giegerich, R., Kurtz, S., Stoye, J.: Efficient implementation of lazy suffix trees. Softw., Pract. Exper. 33(11), 1035–1049 (2003)
17. Grabowski, S., Fredriksson, K.: Bit-parallel string matching under Hamming distance in $O(n\lceil m/w\rceil)$ worst case time. Information Processing Letters 105(5), 182–187 (2008)
18. Gusfield, D.: Algorithms on strings, trees and sequences: computer science and computational biology. Cambridge University Press, Cambridge (1997)
19. He, L., Fang, B.: Linear nondeterministic dawg string matching algorithm. In: Apostolico, A., Melucci, M. (eds.) SPIRE 2004. LNCS, vol. 3246, pp. 70–71. Springer, Heidelberg (2004)
20. Hyyrö, H., Fredriksson, K., Navarro, G.: Increased bit-parallelism for approximate and multiple string matching. ACM J. of Experimental Algorithmics 10(2.6), 1–27 (2005)
21. Kärkkäinen, J., Ukkonen, E.: Sparse suffix trees. In: Cai, J.-Y., Wong, C.K. (eds.) COCOON 1996. LNCS, vol. 1090, pp. 219–230. Springer, Heidelberg (1996)
22. Knuth, D.: The art of computer programming: Combinatorial algorithms. Prefascicle 1a. Draft of section 7.1.3: Bitwise tricks and techniques (2008)
23. Knuth, D.E., Morris Jr, J.H., Pratt, V.R.: Fast pattern matching in strings. SIAM Journal on Computing 6(1), 323–350 (1977)
24. McCreight, E.M.: A space-economical suffix tree construction algorithm. J. Algorithms 23(2), 262–272 (1976)

25. Navarro, G.: NR-grep: a fast and flexible pattern matching tool. Softw. Pract. Exp. 31, 1265–1312 (2001)
26. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Computing Surveys 39(1), article 2 (2007)
27. Navarro, G., Raffinot, M.: Fast and flexible string matching by combining bit-parallelism and suffix automata. ACM J. of Experimental Algorithmics 5(4) (2000)
28. Navarro, G., Raffinot, M.: Flexible Pattern Matching in Strings – Practical on-line search algorithms for texts and biological sequences. Cambridge University Press, Cambridge (2002)
29. Peltola, H., Tarhio, J.: Alternative algorithms for bit-parallel string matching. In: Nascimento, M.A., de Moura, E.S., Oliveira, A.L. (eds.) SPIRE 2003. LNCS, vol. 2857, pp. 80–94. Springer, Heidelberg (2003)
30. Raman, R.: Priority queues: Small, monotone and trans-dichotomous. In: Díaz, J. (ed.) ESA 1996. LNCS, vol. 1136, pp. 121–137. Springer, Heidelberg (1996)
31. Thorup, M.: Combinatorial power in multimedia processors. SIGARCH Comput. Archit. News 31(4), 5–11 (2003)
32. Thorup, M.: On $AC^0$ implementations of fusion trees and atomic heaps. In: Proceedings of SODA 2003, pp. 699–707. SIAM, Philadelphia (2003)
33. Ukkonen, E.: On-line construction of suffix trees. Algorithmica 14(3), 249–260 (1995)
34. Weiner, P.: Linear pattern matching algorithm. In: Proceedings of the 14th Annual IEEE Symposium on Switching and Automata Theory, pp. 1–11 (1973)
35. Wu, S., Manber, U.: Fast text searching allowing errors. Communications of the ACM 35(10), 83–91 (1992)
36. Yao, A.C.: The complexity of pattern matching for a random string. SIAM Journal on Computing 8(3), 368–387 (1979)

# Clustering the Normalized Compression Distance for Influenza Virus Data

Kimihito Ito[1], Thomas Zeugmann[2,⋆], and Yu Zhu[2]

[1] Research Center for Zoonosis Control
Hokkaido University, N-20, W-10 Kita-ku, Sapporo 001-0020, Japan
itok@czc.hokudai.ac.jp
[2] Division of Computer Science
Hokkaido University, N-14, W-9, Sapporo 060-0814, Japan
{thomas,yuar}@mx-alg.ist.hokudai.ac.jp

**Abstract.** The present paper analyzes the usefulness of the normalized compression distance for the problem to cluster the hemagglutinin (HA) sequences of influenza virus data for the HA gene in dependence on the available compressors. Using the CompLearn Toolkit, the built-in compressors `zlib` and `bzip2` are compared.

Moreover, a comparison is made with respect to hierarchical and spectral clustering. For the hierarchical clustering, `hclust` from the R package is used, and the spectral clustering is done via the kLine algorithm proposed by Fischer and Poland (2004).

Our results are very promising and show that one can obtain an (almost) perfect clustering. It turned out that the `zlib` compressor allowed for better results than the `bzip2` compressor and, if all data are concerned, then hierarchical clustering is a bit better than spectral clustering via kLines.

## 1 Introduction

The similarity between objects is a fundamental notion in everyday life. It is also fundamental to many data mining and machine learning algorithms, and, in particular to clustering algorithms. Often the similarity between objects is measured by a domain-specific distance measure based on features of the objects. For defining the right domain-specific distance measure one needs special knowledge about the application domain for extracting the relevant features beforehand. Such an approach does not only cause difficulties, but includes a certain danger or risk of being biased.

If one is pursuing the approach to design data mining algorithms based on domain knowledge, then the resulting algorithms tend to have many parameters. By using these parameters, one can then control the algorithms' sensitivity to certain features. Determining how relevant particular features are is often difficult and may require a certain amount of guessing. Expressing this differently,

⋆ Supported by MEXT Grand-in-Aid for Scientific Research on Priority Areas under Grant No. 21013001.

T. Elomaa et al. (Eds.): Ukkonen Festschrift, LNCS 6060, pp. 130–146, 2010.

one has to tune the algorithms which is requiring domain knowledge and a larger amount of experience. Furthermore, it may be expensive, error prune and time consuming to arrive at a suitable tuning.

However, as a radically different approach, the paradigm of parameter-free data mining has emerged (cf. Keogh *et al.* [11]). The main idea of parameter-free data mining is the design of algorithms that have no parameters and that are universally applicable in all areas.

The problem is whether or not such an approach can be realized at all. It is only natural to ask how an algorithm can perform well if it is not based on extracting the important features of the data and if we are not allowed to adjust its parameters until it is doing the right thing. As expressed by Vitányi *et al.* [21], *if we a priori know the features, how to extract them, and how to combine them into exactly the distance measure we want, we should do just that. For example, if we have a list of cars with their color, motor rating, etc. and want to cluster them by color, we can easily do that in a straightforward way.*

So the approach of parameter-free data mining is aiming at scenarios where we are not interested in a certain similarity measure but in *the* similarity between the objects themselves.

The main goal of the present paper is to test the usefulness of this approach in the domain of influenza viruses. Our data are gene sequences for the hemagglutinin of influenza viruses. The hemagglutinin of influenza viruses is important, since it is responsible for binding the virus to the cell it infects. So far, 16 subtypes of influenza hemagglutinin are known. More details are given in Subsection 3.1. The definite method used by biologists to determine the subtype of the influenza hemagglutinin is based on the antiserum that prevent the docking of the virus. So intuitively, the similarity between the gene sequences for the hemagglutinin of influenza viruses should be large if they have the same subtype and small if the have a different subtype. Therefore, it seems justified to test the paradigm of parameter-free data mining in this domain.

The most promising approach to this paradigm uses Kolmogorov complexity theory [13] as its basis. The key ingredient to this approach is the so-called *normalized information distance* (NID) which was developed by various researchers during the past decade in a series of steps (cf., e.g., [4, 12, 9]). The idea behind it is quite intuitive. If two objects are similar then there should be a simple description of how to transform each one of them into the other one. And conversely, if all descriptions for transforming each one of them into the other one are complex, then the objects should be dissimilar.

More formally the *normalized information distance* between two strings $\mathtt{x}$ and $\mathtt{y}$ is defined as

$$NID(\mathtt{x},\mathtt{y}) = \frac{\max\{\mathsf{K}(\mathtt{x}|\mathtt{y}),\, \mathsf{K}(\mathtt{y}|\mathtt{x})\}}{\max\{\mathsf{K}(\mathtt{x}),\, \mathsf{K}(\mathtt{y})\}}\ , \tag{1}$$

where $\mathsf{K}(\mathtt{x}|\mathtt{y})$ is the length of the shortest program that outputs $\mathtt{x}$ on input $\mathtt{y}$, and $\mathsf{K}(\mathtt{x})$ is the length of the shortest program that outputs $\mathtt{x}$ on the empty input. It is beyond the scope of the present paper to discuss the technical details of the definition of the NID. We refer the reader to Vitányi *et al.* [21].

The NID has nice theoretical properties, the most important of which is universality. The NID is called *universal*, since it accounts for the dominant difference between two objects (cf. Li *et al.* [12] and Vitányi *et al.* [21] and the references therein).

In a sense, the NID captures all computational ways in which the features needed in the traditional approach could be defined. Since its definition involves the Kolmogorov complexity $\mathtt{K}(\cdot)$, the NID cannot be computed. Therefore, to apply this idea to real-world data mining tasks, standard compression algorithms, such as `gzip`, `bzip2`, or `PPMZ`, have been used as approximations of the Kolmogorov complexity. This yields the *normalized compression distance* (NCD) as approximation of the NID (cf. Definition 1).

In a typical data mining scenario we are given some objects as input. The pairwise NCDs for all objects in question form a distance matrix. This matrix can be processed further until finally standard algorithms, e. g., clustering algorithms can be applied. This has been done in a variety of typical data mining scenarios with remarkable success. Works of literature and music have been clustered according to genre or author; evolutionary trees of mammals have been derived from their mitochondrial genome; language trees have been derived from several linguistic corpora (cf., e.g., [9, 11, 6, 7, 3]).

As far as virus data are concerned, Cilibrasi and Vitányi [8] used the SARS TOR2 draft genome assembly 120403 from Canada's Michael Smith Genome Sciences Centre and compared it to other viruses by using the NCD. They used the `bzip2` compressor and applied their quartet tree heuristic for hierarchical clustering. The resulting ternary tree showed relations very similar to those shown in the definitive tree based on medical-macrobiological genomics analysis which was obtained later (see [8] for details).

In the present paper we aim at a detailed analysis of the general method outlined above in the domain of influenza viruses. More specifically, we are interested in learning whether or not specific gene data for the hemagglutinin of influenza viruses are *correctly* classifiable by using the concept of the NCD. For this purpose we have chosen a set of 106 gene sequences from the National Center for Biotechnology Information for which the correct classification of the hemagglutinin is known. As explained in Section 3, there are 16 subtypes commonly called H1, ..., H16. For these 106 gene sequences (or subsets thereof) we then compute the NCD by using the CompLearn Toolkit (cf. [5]) as done in [8].

This computation returns a symmetric matrix $\mathtt{D}$ such that $\mathtt{d}_{\mathtt{ij}}$ is the NCD between the data entries `i` and `j` (henceforth called distance matrix). Furthermore, we study the influence of the compressor chosen and restrict ourselves here to the `zlib` and `bzip2` compressors which are the standard two built-in compressors for the CompLearn Toolkit.

The next step is the clustering. Here of course the variety of possible algorithms is large. Note that the CompLearn Toolkit contains also an implementation of quartet tree heuristic for hierarchical clustering. However, this heuristic is computationally quite expensive and does currently not allow to handle a matrix of

dimension $106 \times 106$. Therefore, we have decided to try the *hierarchical clustering* algorithm from the R package (called `hclust`) with the average option. In this way we obtain a rooted tree showing the relations among the input data.

The second clustering algorithm used is *spectral clustering* via kLines (cf. Fischer and Poland [10]). We have successfully applied this method before (cf. [19, 18]) in settings where the NID is approximated by the so-called Google distance or Web distance. In such settings we are given non-literal objects, i.e., essentially names and not the the literal objects themselves as in the present paper. The Web distance is then based on computing probabilities by determining the frequency of web pages for the individual names and those containing simultaneously two of the given names. We refer the reader to [21] for a comprehensive explanation.

It should be noted that spectral clustering generally requires the transformation of the distance matrix into an adjacency matrix of pairwise similarities (henceforth called similarity matrix). The clustering is then done by analyzing its spectrum.

The results obtained for our data are generally very promising. Since we know the true subtype of the hemagglutinin from the description of the gene sequences used, we could determine the quality of the clustering obtained. Quite often, we arrived at a *perfect* clustering independently of the compressor and of the clustering method used. On the other hand, when including all data or a rather large subset thereof, the clustering obtained is not perfect but the number of errors made is still sufficiently small to make the results interesting. Without going into details here, it can be said that the `zlib` compressor seems more suitable in this setting than the `bzip2` compressor (see Subsection 3.2 for details).

## 2 Background and Theory

As explained in the Introduction, the theoretical basis for computing the distance matrix is deeply based in Kolmogorov complexity theory. In the following we assume the definition of the NID as shown in Equation (1). The definition of the NID depends on the function $\mathsf{K}$ which is *uncomputable*. Thus, the NID is *uncomputable*, too.

Using a real-word compressor, one can approximate the NID by the NCD (cf. Definition 1). Again, we omit details and refer the reader to [21].

**Definition 1.** *The* normalized compression distance *between two strings* $\mathtt{x}$ *and* $\mathtt{y}$ *is defined as*

$$NCD(\mathtt{x},\mathtt{y}) = \frac{\mathsf{C}(\mathtt{xy}) - \min\{\mathsf{C}(\mathtt{x}),\ \mathsf{C}(\mathtt{y})\}}{\max\{\mathsf{C}(\mathtt{x}),\ \mathsf{C}(\mathtt{y})\}}\ ,$$

*where* $\mathsf{C}$ *is any given data compressor.*

Common data compressors are `gzip`, `bzip2`, `zlib`, etc. Note that the compressor $\mathsf{C}$ has to be computable and *normal* in order to make the NCD a useful approximation. This can be stated as follows.

**Definition 2 ([21]).** *A compressor* $C$ *is said to be* normal *if it satisfies the following axioms for all strings* $x$, $y$, $z$ *and the empty string* $\lambda$.

(1) $C(xx) = C(x)$ *and* $C(\lambda) = 0$; (identity)
(2) $C(xy) \geqslant C(x)$; (monotonicity)
(3) $C(xy) = C(yx)$; (symmetry)
(4) $C(xy) + C(z) \leqslant C(xz) + C(yz)$; (distributivity)

*up to an additive* $O(\log n)$ *term, with* $n$ *the maximal binary length of a string involved in the (in)equality concerned.*

These axioms are in various degrees satisfied by good real-world compressors like `bzip2`, `PPMZ` and `gzip`, where the latter did not perform so well, as informal experiments have shown (cf. [9]). Also note that in all cases the compressor-specific window or block size determines the maximum usable length of the arguments. As a matter of fact, for our data these axioms seem to be fulfilled.

For our investigations we used the built-in compressors `bzip2` and `zlib` and the `ncd` function from the CompLearn Toolkit (cf. [5]). After having done this step, we have a distance matrix $D = \left(d^{ncd}(x, y)\right)_{x,y \in X}$, where $X = (x_1, \ldots, x_n)$ is the relevant data list.

Next, we turn our attention to clustering. First, we shortly outline the hierarchical clustering as provided by the R package, i.e., by the program `hclust` (cf. [2]). Input is the $(n \times n)$ distance matrix $D$. The program uses a measure of dissimilarity for the objects to be clustered. Initially, each object is assigned to its own cluster and the program proceeds iteratively. In each iteration the two most similar clusters are joint, and the process is repeated until only a single cluster is left. Furthermore, in every iteration the distances between clusters are recomputed by using the Lance–Williams dissimilarity update formula for the particular method used.

The methods differ in the way in which the distances between clusters are recomputed. Provided are the *complete linkage method*, the *single linkage method*, and the *average linkage clustering*. In the first case, the distance between any two clusters is equal to the greatest similarity from any member of one cluster to any member of the other cluster. This method works well for compact clusters but causes sensitivity to outliers. The second method pays attention solely to the area where the two clusters come closest to one another. The more distant parts of the clusters and the overall structure of the clusters is not taken into account. If the total number of clusters is large, a messy clustering may result.

The *average linkage clustering* defines the distance between any two clusters to be the average of distances between all pairs of objects from any member of one cluster to any member of the other cluster. As a result, the average pairwise distance within the newly formed cluster, is minimum.

Heuristically, the average linkage clustering should give the best results in our setting, and thus we have chosen it (see also Manning *et al.* [14] for a thorough exposition). Note that for hierarchical clustering the number $k$ of clusters does *not* to be known in advance.

Next, the *spectral clustering* algorithm used is shortly explained. Spectral clustering is an increasingly popular method for analyzing and clustering data by using only the matrix of pairwise similarities. It was invented more than 30 years ago for partitioning graphs (cf., e.g., Spielman and Teng [20] for a brief history and Luxburg [22] for a tutorial). Formally, spectral clustering can be related to approximating the normalized min-cut of the graph defined by the adjacency matrix of pairwise similarities [24]. Finding the exactly minimizing cut is an NP-hard problem.

The transformation of the distance matrix into a similarity matrix is done by using a suitable kernel function. In our experiments we have used the Gaussian kernel function, i.e.,

$$k(x,y) = \left(\exp(-\frac{1}{2}d(x,y)^2/(2\cdot\sigma^2))\right) , \tag{2}$$

where $\sigma$ is the kernel width. As pointed out by Perona and Freeman [17], there is nothing magical with this function. Moreover, it is most commonly used. An advantage of using the Gaussian kernel function is that the resulting similarity matrix is positive definite.

So, the remaining problem is a suitable choice for $\sigma$. Unfortunately, the performance of spectral clustering heavily depends on this $\sigma$. In the experiments, we compute the mean value of the entries of the distance matrix $D$ and then set $\sigma = \mathtt{mean}(D)/\sqrt{2}$. In this way, the kernel is most sensitive around $\mathtt{mean}(D)$. Though we are not aware of a theoretical result supporting this choice, it worked remarkably well and further studies are needed to explore the properties of this choice.

The final spectral clustering algorithm for a known number of clusters $k$ is stated below.

**Algorithm:** *Spectral Clustering*
*Input*: data list $X = (x_1, x_2, \ldots, x_n)$, number of clusters $k$
*Output*: clustering $c \in \{1 \ldots k\}^n$

1. for $x, y \in X$, compute the distance matrix $D = \left(d^{ncd}(x,y)\right)_{x,y\in X}$
2. compute $\sigma = \mathtt{mean}(D)/\sqrt{2}$
3. compute the similarity matrix $A = \left(\exp(-\frac{1}{2}d(x,y)^2/(2\cdot\sigma^2))\right)$
4. compute the Laplacian $L = S^{-\frac{1}{2}}AS^{-\frac{1}{2}}$, where $S_{ii} = \sum_j A_{ij}$ and $S_{ij} = 0$ for $i \neq j$
5. compute top $k$ eigenvectors $V \in \mathbb{R}^{n\times k}$
6. cluster $V$ using kLines [10]

## 3 Experiments and Results

In this section we describe the data used, the experiments performed and the results obtained.

### 3.1 Influenza Viruses – The Data Set

We shortly describe the data set used. For any relevant background concerning the biological aspects of the influenza viruses we refer the reader to Palese and Shaw [16] and Wright *et al.* [23].

Influenza viruses were probably a major cause of morbidity and mortality world wide. Large segments of the human population are affected every year. The family of *Orthomyxoviridae* is defined by viruses that have a negative-sense, single-stranded, and segmented RNA genome. There are five different genera in the family of *Orthomyxoviridae*: the influenza viruses A, B and C; *Thogotovirus*; and *Isavirus*. Influenza A viruses have a complex structure and possess a lipid membrane derived from the host cell (cf. Figure 1).

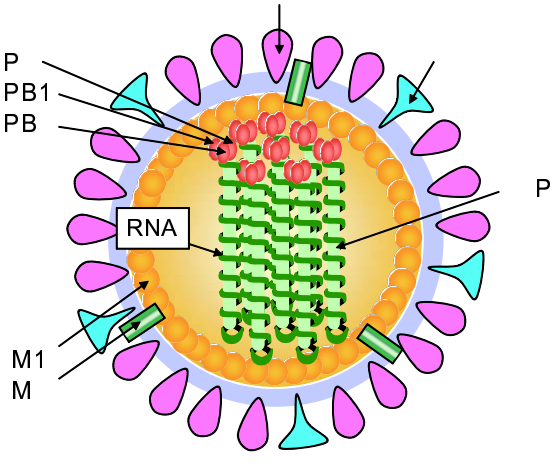



**Fig. 1.** Influenza A virus

Biologists classify influenza A viruses primarily by their hemagglutinin (HA) subtypes and neuraminidase (NA) subtypes. So far, 16 subtypes of HA are known and commonly denoted by H1, . . . , H16. In addition to these HA types, biologists distinguish 9 NA subtypes denoted by N1, . . . , N9.

Influenza A viruses of all 16 hemagglutinin (H1-H16) and 9 neuraminidase (N1-N9) subtypes are maintained in their nature host, i.e., the duck. Of these duck viruses, H1N1, H2N2 and H3N2 subtypes jumped into human population, and caused three pandemics in the last century. Therefore, in the experiments performed we have exclusively selected data of influenza viruses that have been obtained from viruses hosted by the duck.

The complete genome of these influenza viruses has 8 segmented-genes. Of these 8 genes, here we are only interested in their HA gene, since HA is the major target of antibodies that neutralize viral infectivity, and responsible for binding the virus to the cell it infects. The corresponding gene is found on segment 4.

Each datum consists of a sequence of roughly 1800 letters from the alphabet {A, T, G, C}, e.g., looking such as

AAAAGCAGGGGAATTTCACAATTAAA . . . TGTATATAATTAGCAAA.

These gene sequences are publicly available from the National Center for Biotechnology Information (NCBI) which has one of the largest collections of such sequences (cf. [15]).

When analyzed by biologists the definite method to determine the correct HA subtype is based on the antiserum that prevent the docking of the virus. Sometimes biologists also compare the actual sequence to already analyzed sequences and produce a guess based on the Hamming distance of the new sequence to the analyzed ones.

As explained in the Introduction, the primary goal of the investigations undertaken is to cluster the sequences correctly with respect to their HA subtype. In order to achieve this goal with collected from each subtype up to 8 examples. The reason for choosing at most 8 sequences from each type has been caused by their availability. While for some subtypes there are many sequences, there are also subtypes for which only very few sequences are available. The extreme case is the subtype H16 for which only one sequence is in the data base. Figure 2 shows the number of sequences chosen.

It should be noted that most of these sequences are marked as `complete cds`, but some are also marked as `partial cds` by the NCBI. For a complete list of the data description we refer the reader to

`http://www-alg.ist.hokudai.ac.jp/106Data_description.html` .

For the ease of presentation, below we use the following abbreviation for the data entries. Instead of giving the full description, e.g.,

>gi|113531192|gb|AB271117| /Avian/4 (HA)/H10N1/Hong Kong/1980/// Influenza A virus (A/duck/Hong Kong/938/80(H10N1)) HA gene for hemagglutinin, complete cds.

We refer to this datum as H10N1AB271117 for short.

Among the available files, there were two files containing only a very short partial sequence of the gene, i.e., H7N1AM157391 and H10N4AM922160 (483 and 80 letters, respectively). So, we did not consider these two files, since they do not seem to contain enough information.

### 3.2 Results

All experiments have been performed under SuSE Linux. As already mentioned, for the hierarchical clustering we used the open source R package (cf. [2]).

The Algorithm *Spectral Clustering* from Section 2 has been realized by performing Step 1 via the CompLearn function `ncd` (cf. [5]). Steps 2 through 6 have

| H1 | H2 | H3 | H4 | H5 | H6 | H7 | H8 | H9 | H10 | H11 | H12 | H13 | H14 | H15 | H16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 8 | 8 | 8 | 8 | 8 | 8 | 8 | 7 | 8 | 8 | 8 | 8 | 2 | 4 | 4 | 1 |

**Fig. 2.** Number of sequences for each subtype

been implemented in `GNU Octave`, version 2.1.72 (cf. [1]). It should be noted that `ncd` assigns 0.000000 to all elements on the main diagonal of the distance matrix (Version 1.1.5).

By performing our experiments we aimed to answer the following questions. First, does the NCD provide enough information to obtain a correct clustering for the virus data? Second, does the rather large number of clusters (recall that we 16 HA types) cause any problems? Third, do the answers to the first and second question depend on the compressor and clustering, respectively, chosen?

To get started and for the sake of comparison, we used the subset containing all data belonging to H1, H2, and H3, i.e., a total of 24 sequences (cf. Figure 2).

Using the `maketree` program from the CompLearn Toolkit, we get the following clustering (cf. Figures 3 and 4). As Figures 3 and 4 show, the data are clearly and correctly separated into three clusters. However, the intra-cluster dissimilarities clearly differ from inter-cluster dissimilarities in Figure 3, i.e., for the `zlib` compressor, while there is no such clear difference for the `bzip2` compressor (cf. Figure 4).

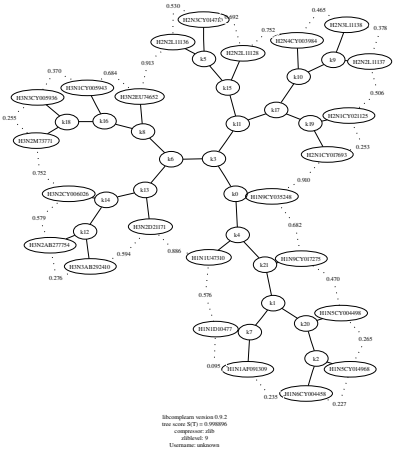



**Fig. 3.** Classification of HA sequences; compr.: `zlib`

**Fig. 4.** Classification of HA sequences; compr.: `bzip2`

Using `hclust` we obtained the trees shown in Figure 5 and 6 for the matrix D computed for the compressor `zlib` and `bzip2`, respectively. As Figures 5 and 6 show, we obtained a correct clustering into three clusters independently of the compressor used.

Next, we tried our algorithm *Spectral Clustering* for the same data set. After having computed the matrix D, we get the following order of the data

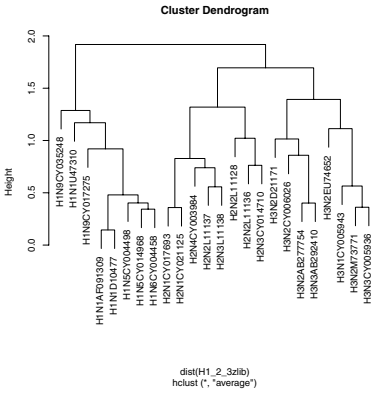



**Fig. 5.** Clustering all HA sequences for H1 through H3 via `hclust`; compr.: `zlib`

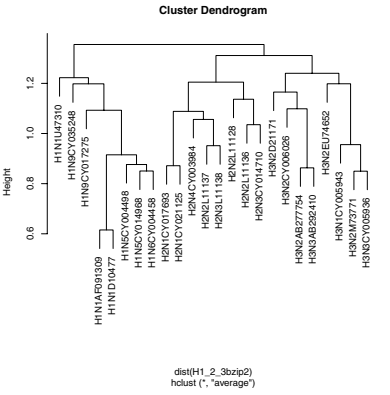



**Fig. 6.** Clustering all HA sequences for H1 through H3 via `hclust`; compr.: `bzip2`

`H2N4CY003984, H3N1CY005943, H3N2AB277754, H1N9CY017275, H1N9CY035248, H3N3CY005936, H2N2L11128, H2N2L11136, H2N2L11137, H2N1CY017693, H2N1CY021125, H2N3L11138, H1N6CY004458, H1N1D10477, H2N3CY014710, H3N2EU74652, H3N2CY006026, H1N1AF091309, H1N1U47310, H3N3AB292410, H3N2D21171, H3N2M73771, H1N5CY004498, H1N5CY014968`

Since spectral clustering is a hard clustering method, it has to return for each data entry just one class label. Assigning canonically the clusters 1, 2, and 3 to the HA subtypes `H1N...`, `H2N...`, and `H3N...`, respectively, we therefore should get the sequence

2 3 3 1 1 3 2 2 2 2 2 2 1 1 2 3 3 1 1 3 3 3 1 1

which was indeed returned for both compressors. Note that $\sigma = 0.56078$ and $\sigma = 0.57329$ for the `zlib` and `bzip2` compressor, respectively.

Next, we tried all HA sequences for H1 through H8 and from H9 through H16. The reason for this partition has been caused by the different number of sequences available. Recall that there are only two sequences for H13 and only one sequence for H16 (cf. Figure 2).

For H1 through H8 the hierarchical clustering was error free for the `zlib` compressor but not for `bzip2` compressor (1 error) (see Figures 8 and 9 in the Appendix). Interestingly, for H9 through H16 the tree obtained for the `zlib` compressor contains 4 errors, while the one obtained for `bzip2` compressor has only one error.

Our spectral clustering algorithm returned a perfect clustering for all HA sequences for H1 through H8 for both compressors. On the other hand, for all sequences from H9 through H16 the results differed with respect to the compressor used.

| | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| c0 = | 7 | 7 | 14 | 2 | 11 | 12 | 12 | 3 | 7 | 10 | 10 | 5 | 9 | 9 | 9 | 3 | 1 | 1 | 9 | 11 |
| sp = | 7 | 7 | 14 | 2 | 11 | 12 | 12 | 3 | 7 | 10 | 10 | 5 | 9 | 9 | 9 | 3 | 1 | 1 | 9 | 11 |
| c0 = | 11 | 5 | 3 | 7 | 5 | 2 | 2 | 2 | 4 | 10 | 5 | 8 | 12 | 2 | 2 | 4 | 4 | 4 | 11 | 9 |
| sp = | 11 | 5 | 3 | 7 | 5 | 2 | 2 | 2 | 4 | 10 | 5 | 8 | 12 | 2 | 2 | 4 | 4 | 4 | 11 | 9 |
| c0 = | 10 | 2 | 6 | 6 | 6 | 5 | 1 | 1 | 4 | 10 | 7 | 4 | 8 | 15 | 2 | 9 | 9 | 16 | 10 | 14 |
| sp = | 10 | 2 | 13 | 13 | 6 | 5 | 1 | 1 | 4 | 10 | 7 | 4 | 8 | 15 | 2 | 9 | 9 | 3 | 10 | 14 |
| c0 = | 14 | 7 | 7 | 6 | 14 | 7 | 8 | 8 | 12 | 12 | 11 | 15 | 3 | 15 | 5 | 11 | 3 | 1 | 1 | 8 |
| sp = | 14 | 7 | 7 | 6 | 14 | 7 | 8 | 8 | 12 | 12 | 11 | 15 | 3 | 15 | 5 | 11 | 3 | 1 | 1 | 8 |
| c0 = | 4 | 3 | 3 | 6 | 12 | 10 | 4 | 5 | 3 | 6 | 13 | 13 | 12 | 1 | 1 | 11 | 12 | 8 | 11 | 10 |
| sp = | 4 | 3 | 3 | 6 | 12 | 10 | 4 | 5 | 3 | 6 | 13 | 13 | 12 | 1 | 1 | 11 | 12 | 8 | 11 | 10 |
| c0 = | 5 | 9 | 15 | 8 | 6 | 6 | | | | | | | | | | | | | | |
| sp = | 5 | 9 | 15 | 8 | 13 | 13 | | | | | | | | | | | | | | |

**Fig. 7.** Clustering all HA sequences via *Spectral Clustering*; compr.: `zlib`

For the `zlib` compressor we obtained 5 errors and for the `bzip2` compressor the number of errors was 7 when using for $\sigma$ the mean as described above. However, it is well-known that spectral clustering is quite sensitive to the kernel width $\sigma$. So, we also tried to vary it a bit around the mean by rounding it to two decimal digits and then changing the second one. For `zlib` the mean was 0.60873 and after two variations we found $\sigma = 0.59$ which resulted in just one error, i.e., H16 was classified as H13. For the `bzip2` compressor such an improvement could not be obtained.

As a possible explanation we conjecture that one needs a certain minimum of available sequences in order to arrive at a correct spectral clustering. Trying all HA sequences for H1 through H12 kind of confirmed this conjecture, since we again obtained a perfect spectral clustering for both compressors.

For the hierarchical clustering, the tree obtained for the `zlib` compressor is correct, but the the one obtained for the `bzip2` compressor has one error. These trees are shown in the Appendix.

Finally, we tried all data. Again hierarchical clustering was best for the `zlib` compressor and showed only 2 errors. For the `bzip2` compressor, we obtained 3 errors (see the Appendix for details). On the other hand, the best result we could obtain for spectral clustering had 5 errors (for both compressors). In Figure 7 we show the clustering obtained for the `zlib` compressor for $\sigma = 0.63$, where `c0` is the desired classification and `sp` the one returned from the spectral clustering algorithm (partitioned into six groups).

So, the errors occur at positions 43, 44, 58, 105, and 106 and affect H6 which is four times assigned to H13 and one time H16 which got in the H3 cluster. We omit further details due to the lack of space.

Note that one can also compute the sum square error (s.s.e.) of all eigenvalues with respect to their means in order to determine quite reliably from the eigenvalues of the Laplacian the number `k` of clusters (cf. Poland and Zeugmann [19] for details).

## 4 Conclusions

The usefulness of the normalized compression distance for clustering the HA type of virus data for the HA gene for it (segment 4) has been demonstrated. Though we just used the built-in compressors `zlib` and `bzip2` the results are (almost) correct when clustering the resulting distance matrix for the whole data set with `hclust` or spectral clustering via kLines. What is also remarkable in this context is the robustness with respect to the completeness of the data. As mentioned above, some data contain only a partial cds but this did not influence the quality of the clustering as the results, e.g., H1N1U47310 and H3N2D21171 have only 1000 letters.

We have not reported the running time here, since it is still in the range of several seconds. Though the quartet tree algorithm by Cilibrasi and Vitányi [8] returns a high quality classification, it lacks scalability, since it tries to optimize a quality function, a task which is NP-hard. So, even for the small example including the 24 data for H1, H2, and H3 resulting in $(24 \times 24)$ distance matrix, it took hours to find the resulting (very good) clustering. In contrast, the clustering algorithms used in this study scale nicely at least up to the amount of data for which the distance matrix is efficiently computable, since they have almost the same running time as the `ncd` algorithm.

## References

[1] GNU Octave, `http://www.gnu.org/software/octave/`
[2] The R project for statistical computing, `http://www.r-project.org/`
[3] Benedetto, D., Caglioti, E., Loreto, V.: Language trees and zipping. Phys. Rev. Lett. 88(4), 048702–1–048702–4 (2002)
[4] Bennett, C.H., Gács, P., Li, M., Vitányi, P.M.B., Zurek, W.H.: Information distance. IEEE Transactions on Information Theory 44(4), 1407–1423 (1998)
[5] Cilibrasi, R.: The CompLearn Toolkit (2003), `http://www.complearn.org/`
[6] Cilibrasi, R., Vitányi, P.M.B.: Automatic meaning discovery using Google. CWI, Amsterdam (2006)
[7] Cilibrasi, R., Vitányi, P.M.B.: Similarity of objects and the meaning of words. In: Cai, J.-Y., Cooper, S.B., Li, A. (eds.) TAMC 2006. LNCS, vol. 3959, pp. 21–45. Springer, Heidelberg (2006)
[8] Cilibrasi, R., Vitányi, P.M.B.: A new quartet tree heuristic for hierarchical clustering. In: Arnold, D.V., Jansen, T., Vose, M.D., Rowe, J.E. (eds.) Theory of Evolutionary Algorithms. Dagstuhl Seminar Proceedings, Schloss Dagstuhl, Germany. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), vol. (06061) (2006)
[9] Cilibrasi, R., Vitányi, P.M.B.: Clustering by compression. IEEE Transactions on Information Theory 51(4), 1523–1545 (2005)
[10] Fischer, I., Poland, J.: New methods for spectral clustering. Technical Report IDSIA-12-04, IDSIA/USI-SUPSI, Manno, Switzerland (2004)

[11] Keogh, E., Lonardi, S., Ratanamahatana, C.A.: Towards parameter-free data mining. In: KDD 2004: Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining, pp. 206–215. ACM Press, New York (2004)
[12] Li, M., Chen, X., Li, X., Ma, B., Vitányi, P.M.B.: The similarity metric. IEEE Transactions on Information Theory 50(12), 3250–3264 (2004)
[13] Li, M., Vitányi, P.: An Introduction to Kolmogorov Complexity and its Applications, 3rd edn. Springer, Heidelberg (2008)
[14] Manning, C.D., Raghavan, P., Schütze, H.: Introduction to Information Retrieval. Cambridge University Press, Cambridge (2008)
[15] National Center for Biotechnology Information. Influenza Virus Resource, information, search and analysis, http://www.ncbi.nlm.nih.gov/genomes/FLU/FLU.html
[16] Palese, P., Shaw, M.L.: Orthomyxoviridae: The viruses and their replication. In: Knipe, D.M., Howley, P.M., et al. (eds.) Fields' Virology, 5th edn., pp. 1647–1689. Lippincott Williams & Wilkins, Philadelphia (2007)
[17] Perona, P., Freeman, W.: A factorization approach to grouping. In: Burkhardt, H., Neumann, B. (eds.) ECCV 1998. LNCS, vol. 1406, pp. 655–670. Springer, Heidelberg (1998)
[18] Poland, J., Zeugmann, T.: Clustering pairwise distances with missing data: Maximum cuts versus normalized cuts. In: Todorovski, L., Lavrač, N., Jantke, K.P. (eds.) DS 2006. LNCS (LNAI), vol. 4265, pp. 197–208. Springer, Heidelberg (2006)
[19] Poland, J., Zeugmann, T.: Clustering the google distance with eigenvectors and semidefinite programming. In: Knowledge Media Technologies, First International Core-to-Core Workshop. Diskussionsbeiträge, Institut für Medien und Kommunikationswisschaft, vol. 21, pp. 61–69. Technische Universität Ilmenau (2006)
[20] Spielman, D.A., Teng, S.-H.: Spectral partitioning works: Planar graphs and finite element meshes. In: Proceedings of the 37th Annual IEEE Conference on Foundations of Computer Science, pp. 96–105. IEEE Computer Society, Los Alamitos (1996)
[21] Vitányi, P.M.B., Balbach, F.J., Cilibrasi, R.L., Li, M.: Normalized information distance. In: Information Theory and Statistical Learning, pp. 45–82. Springer, New York (2008)
[22] von Luxburg, U.: A tutorial on spectral clustering. Statistics and Computing 17(4), 395–416 (2007)
[23] Wright, P.F., Neumann, G., Kawaoka, Y.: Orthomyxoviruses. In: Knipe, D.M., Howley, P.M., et al. (eds.) Fields Virology, 5th edn., pp. 1691–1740. Lippincott Williams & Wilkins, Philadelphia (2007)
[24] Yu, S.X., Shi, J.: Multiclass spectral clustering. In: Proceedings of the Ninth IEEE International Conference on Computer Vision, vol. 2, pp. 313–319. IEEE Computer Society, Los Alamitos (2003)

## Appendix

Here we show the results obtained for the remaining data.

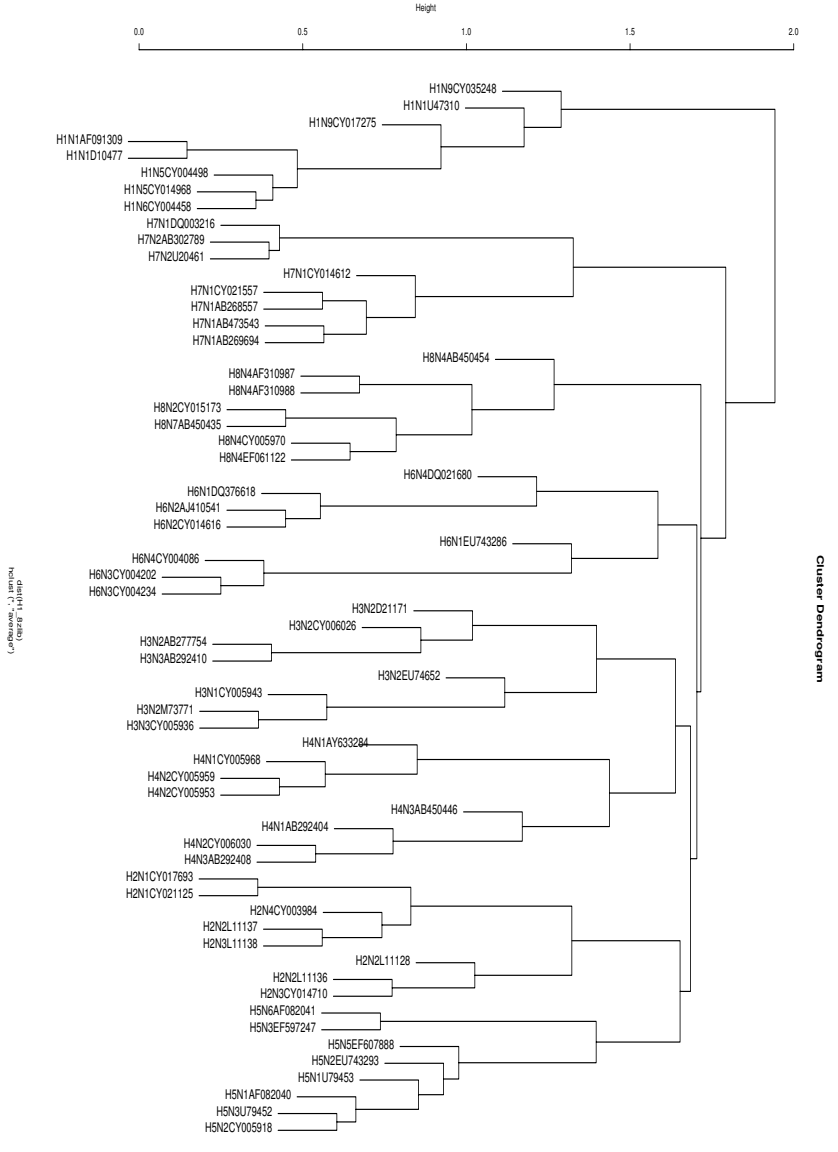



**Fig. 8.** Clustering of all HA sequences for H1 through H8 via `hclust`; compr.: `zlib`

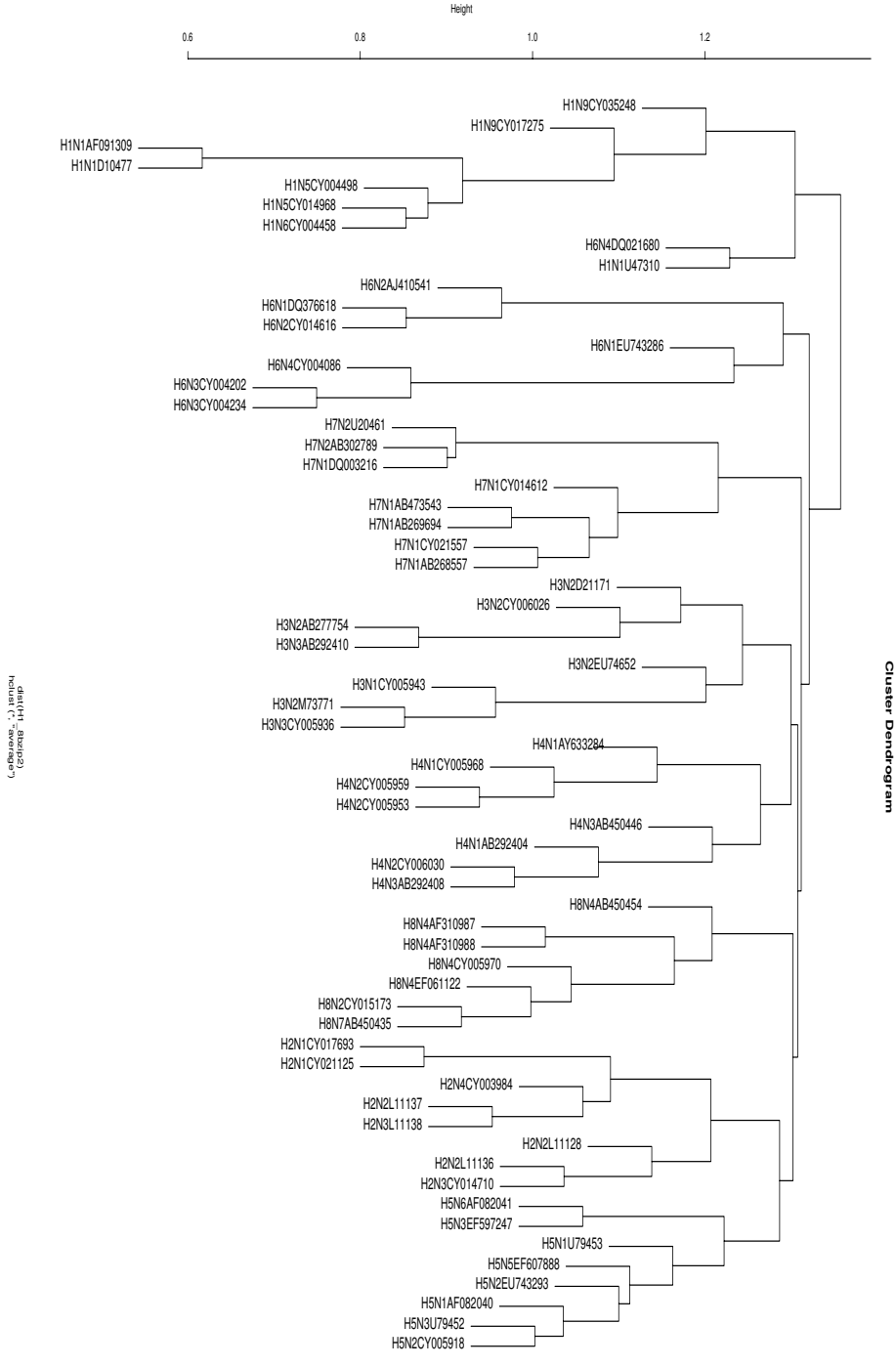



**Fig. 9.** Clustering of all HA sequences for H1 through H8 via `hclust`; compr.: `bzip`

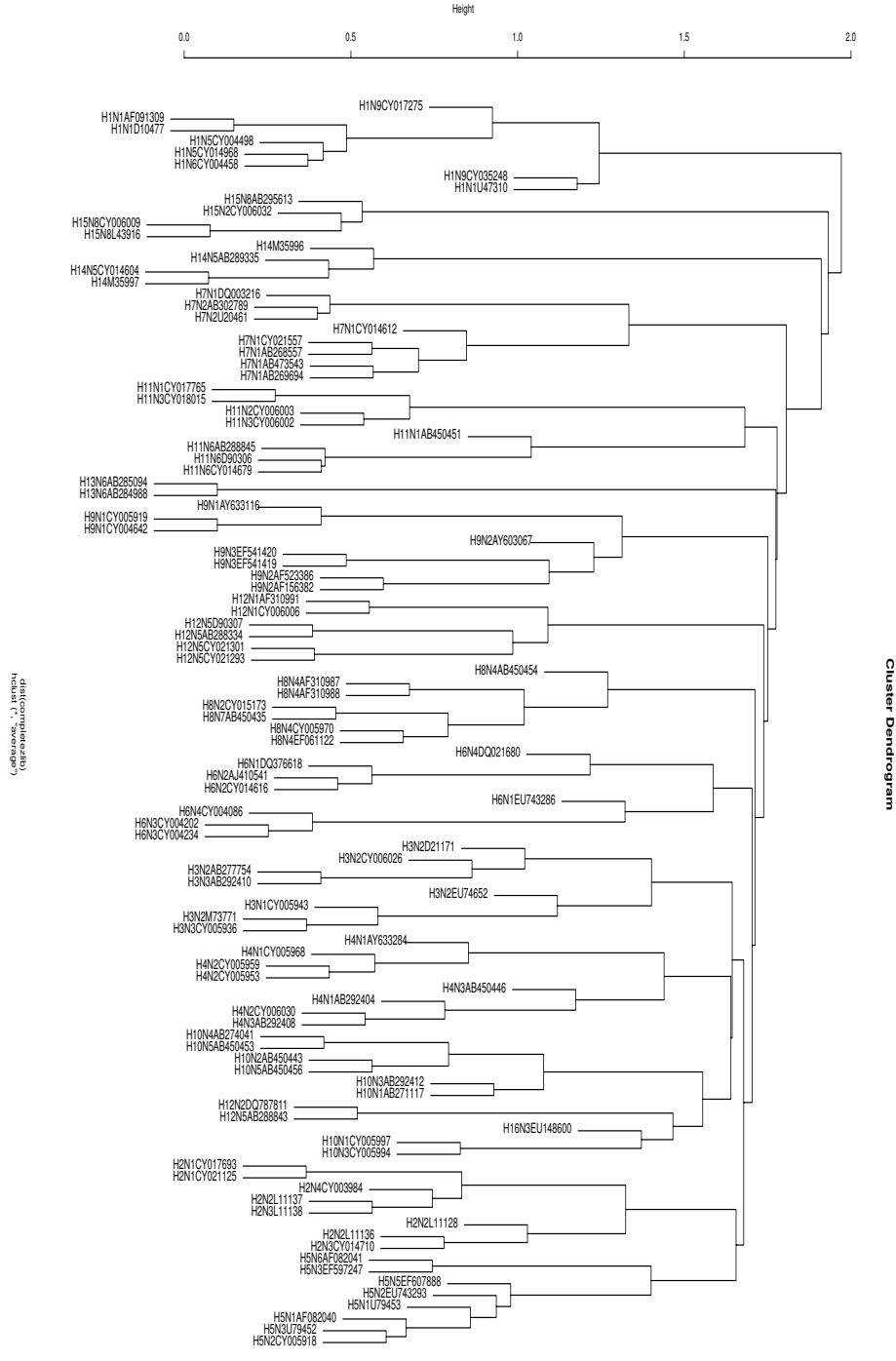



**Fig. 10.** Clustering of all HA sequences via `hclust`; compr.: `zlib`

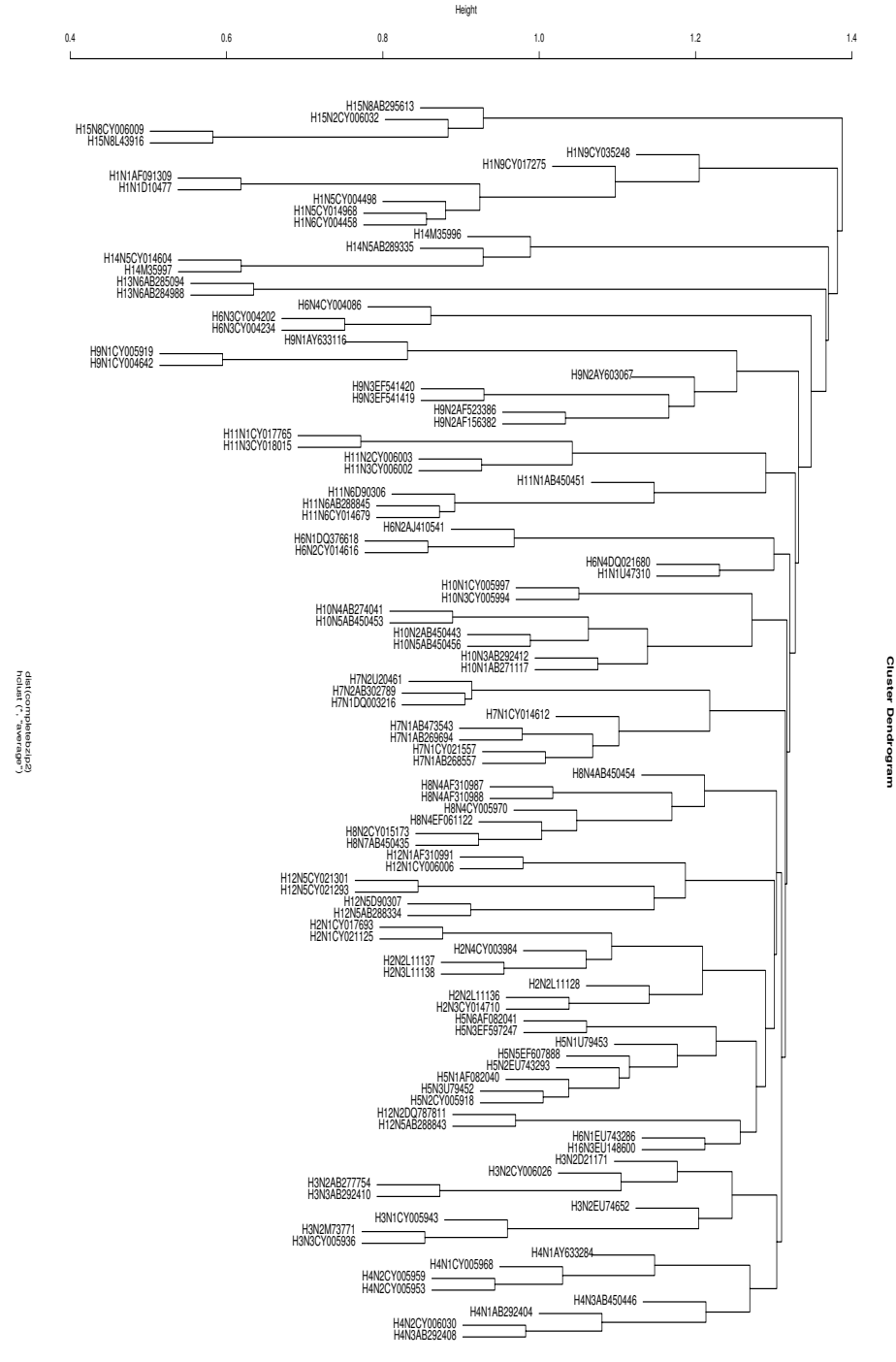



**Fig. 11.** Clustering of all HA sequences via `hclust`; compr.: `bzip`

# An Evolutionary Model of DNA Substring Distribution

Meelis Kull[1,2], Konstantin Tretyakov[1], and Jaak Vilo[1,2]

[1] Institute of Computer Science, University of Tartu,
Liivi 2, 50409 Tartu, Estonia
[2] Quretec Ltd. Ülikooli 6a, 51003 Tartu, Estonia
{meelis.kull,konstantin.tretjakov,jaak.vilo}@ut.ee
http://biit.cs.ut.ee/

**Abstract.** DNA sequence analysis methods, such as motif discovery, gene detection or phylogeny reconstruction, can often provide important input for biological studies. Many of such methods require a *background model*, representing the expected distribution of short substrings in a given DNA region. Most current techniques for modeling this distribution disregard the evolutionary processes underlying DNA formation. We propose a novel approach for modeling DNA $k$-mer distribution that is capable of taking the notions of evolution and natural selection into account. We derive a computionally tractable approximation for estimating $k$-mer probabilities at genetic equilibrium, given a description of evolutionary processes in terms of fitness and mutation probabilities. We assess the goodness of this approximation via numerical experiments. Besides providing a generative model for DNA sequences, our method has further applications in motif discovery.

## 1 Introduction

From the very early days of bioinformatics, the computational analysis of DNA sequences has been one of its primary focuses. Genomic sequence is believed to literally define most of the key aspects of each organism's life and development. However, the sheer size of genomic data makes a manual human analysis practically impossible.

The information in the DNA can be viewed and analyzed at various levels of abstraction, from the large modules such as chromosomes and genes corresponding to perceivable phenotypic traits, down to short codes or motifs guiding the low-level chemical processes. In this work we turn our attention to the latter, and address the problem of modeling the distribution of short substrings (*i.e.* $k$-mers) in the genomic regulatory regions.

Understanding and modeling the distribution of short substrings is often the key element in the analysis of DNA regulatory regions, because it provides a concise description of the most relevant "regulatory codes" exploited by the organism. For example, some short substrings are generally known to directly induce the expression of the nearby genes. Others act as repressors or indirect

T. Elomaa et al. (Eds.): Ukkonen Festschrift, LNCS 6060, pp. 147–157, 2010.

switches [1]. Therefore, the problem of modeling the low-level substring distribution is an important step for further analysis, such as motif discovery and phylogenetics. This problem is most commonly referred to as *background modeling* and so far quite often overlooked in favor of rather simplistic approaches limited to 1-mer (*i.e.* single-nucleotide) frequencies only.

So far most of the probabilistic $k$-mer models have been based on either purely phenomenological ideas (*i.e.* HMMs), or loosely related to chemical binding energy models (*i.e.* PWMs) [2]. Despite the undoubted practical usefulness, the abovementioned models are, however, incapable of incorporating the question of *how could such a distribution arise* in the regulatory region in the first place.

In this work we present a novel modeling framework for relating the process of evolution and natural selection to the $k$-mer distribution expected to arise in the corresponding population as a result of this process. More precisely, we examine the situation where the *fitness*, *i.e.* the expected number of offspring of the individual with a given regulatory sequence is related to the features present in the sequence, such as the *counts of certain substrings*. We then derive a computationally tractable way of inferring the expected $k$-mer distribution for the given fitness function and *sequence mutation rates*.

The corresponding model is reasonably general and can be used to incorporate more complex evolutionary assumptions into various DNA analysis methods. The reasons for including these assumptions are twofold. Firstly, introducing a strong inductive bias into low-level models (background) can result in better precision of the higher-level pattern analysis and motif discovery algorithms [3]. Secondly, the background and higher-level patterns can be handled by a single model as both are products of evolutionary processes. By matching the model to data it is possible to learn something about these processes. Mustonen and Lässig provide a cross-species model with transcription factor binding sites under selection and background in neutral evolution [4]. Similar methods have been used for analysing binding site turnover [5,6,7]. Our approach uses a single species but allows for more complicated evolutionary models by defining the fitness and mutation functions.

The incorporation of evolution, even in its most simple form, can lead to computationally expensive procedures. We propose simplifications, which make the computations tractable and yet still provide a close approximation when tested numerically.

We believe our model provides a novel view on the problem of modeling DNA substring distribution and has a potential for further development and applications.

## 2 Methods

### 2.1 Evolutionary Model of DNA Regulatory Regions

It is known that the formation of DNA sequence is mainly driven by evolution. Genomic sequence mutates from generation to generation, and the less successful variants tend to stage out in favor of the more successful ones. We consider the

influence of this process on a single regulatory region, *e.g.* a promoter of some gene. We assume that the *fitness* of a given region (*i.e.,* its expected number of offspring) is largely determined by a number of certain functional elements in the region. In this case the evolutionary process will necessarily impose some nontrivial $k$-mer distribution on the corresponding DNA region in the whole population. Our goal here is to compute this distribution from the information about the important features and their influence on promoter's fitness, taking into account the sequence mutation rates.

Formally, let us fix a promoter region of length $n$. Suppose we know that the expected average number of offspring for an individual with sequence $s$ in this promoter region is $f_s$, for all $s \in A^n$, where $A = \{\mathsf{A}, \mathsf{C}, \mathsf{G}, \mathsf{T}\}$. We shall refer to $f$ as the *fitness function*. Suppose we also know the probability $m_{t\to s}$ of sequence $t$ at this region mutating into a sequence $s$ within one generation. Let the expected proportion of individuals with promoter sequence $s$ in a population be $p_s$, for all $s \in A^n$. We say that this population is in genetic equilibrium, if the expected proportion of individuals with sequence $s$ in the offspring population $p'_s$ is equal to $p_s$. Note that we assume reproduction to be performed before mutation. All of the following results could also be proven for the opposite order, yet the equilibrium probabilities would be different.

### 2.2 The Equilibrium Distribution

We shall now prove that the equilibrium of the promoter sequence distribution exists and is uniquely determined under very general assumptions. To do that we first derive a formula to calculate $p'$ from $p$. For a population of size $i$, the expected number of individuals with sequence $t$ is $i \cdot p_t$. The expected number of offspring for these individuals is $i \cdot p_t \cdot f_t$. As any sequence $t$ can mutate into sequence $s$ with probability $m_{t\to s}$, the expected number of offspring with sequence $s$ is $\sum\limits_{t\in A^n} ip_t f_t m_{t\to s}$. In order to get the expected proportion of sequence $s$ in the offspring population, we have to divide by the size of the new population:

$$p'_s = \frac{\sum\limits_{t\in A^n} ip_t f_t m_{t\to s}}{\sum\limits_{u\in A^n}\sum\limits_{t\in A^n} ip_t f_t m_{t\to u}} = \frac{i\sum\limits_{t\in A^n} p_t f_t m_{t\to s}}{i\sum\limits_{t\in A^n} p_t f_t \sum\limits_{u\in A^n} m_{t\to u}} = \frac{\sum\limits_{t\in A^n} p_t f_t m_{t\to s}}{\sum\limits_{t\in A^n} p_t f_t},$$

where $\sum\limits_{u\in A^n} m_{t\to u} = 1$ because it is the probability of $t$ mutating into any other sequence, including itself. We can now prove the following theorem.

**Theorem 1.** *Let $n \in \mathbb{N}$, $f_s > 0$ for all $s \in A^n$, and $m_{t\to s} > 0$ for all $s, t \in A^n$. Then there exists a unique probability distribution $p^{\mathsf{EQ}} = (p_s^{\mathsf{EQ}})_{s\in A^n}$ with $p_s^{\mathsf{EQ}} > 0$ for all $s \in A^n$, and $\sum\limits_{u\in A^n} p_u^{\mathsf{EQ}} = 1$, such that the equilibrium condition holds:*

$$p_s^{\mathsf{EQ}} = \frac{\sum\limits_{t\in A^n} p_t^{\mathsf{EQ}} f_t m_{t\to s}}{\sum\limits_{t\in A^n} p_t^{\mathsf{EQ}} f_t}, \tag{1}$$

*Proof.* Let us convert the formula (1) to a matrix form. For that we define an $|A^n| \times |A^n|$ matrix $E = (e_{st})_{s,t \in A^n}$ with $e_{st} = f_t \cdot m_{t \to s}$. Now (1) is equivalent to

$$\lambda p^{\mathsf{EQ}} = E p^{\mathsf{EQ}}, \tag{2}$$

where

$$\lambda = \sum_{t \in A^n} p_t^{\mathsf{EQ}} f_t. \tag{3}$$

It remains to prove that the matrix $E$ has a unique eigenvector $p^{\mathsf{EQ}}$ with positive components, having the sum of components equal to 1, and the corresponding eigenvalue $\lambda$ satisfies the constraint (3).

As all the elements of matrix $E$ are positive, we can apply the Perron-Frobenius theorem, stating that real matrices with positive entries have a unique largest real eigenvalue and that the corresponding eigenvector has strictly positive components. Furthermore, it states that this eigenvector is the only eigenvector with strictly positive components. After scaling this eigenvector so that its components would sum up to 1, we have obtained the required $p^{\mathsf{EQ}}$. It remains to prove that the corresponding eigenvalue $\lambda$ satisfies (3). This can be shown by summing up the components of vectors on both sides of the equation (2). On the left we get $\lambda$ as the components of $p^{\mathsf{EQ}}$ sum up to 1. On the right we get the required expression:

$$\sum_{s \in A^n} \sum_{t \in A^n} e_{st} p_t^{\mathsf{EQ}} = \sum_{s \in A^n} \sum_{t \in A^n} p_t^{\mathsf{EQ}} f_t m_{t \to s} = \sum_{t \in A^n} p_t^{\mathsf{EQ}} f_t \sum_{s \in A^n} m_{t \to s} = \sum_{t \in A^n} p_t^{\mathsf{EQ}} f_t,$$

because $\sum_{s \in A^n} m_{t \to s} = 1$. □

The following theorem expresses $p^{\mathsf{EQ}}$ in terms of the fitness function and mutation probabilities.

**Theorem 2.** *Let $n \in \mathbb{N}$, $f_s > 0$, $m_{s \to t} > 0$, $p_s^{(0)} \geq 0$ for all $s, t \in A^n$, and $\sum_{u \in A^n} p_u^{(0)} = 1$. Further, let $p_s^{(i)}$ be defined for each $i \in \mathbb{N}$ and $s \in A^n$ as follows:*

$$p_s^{(i)} = \frac{\sum_{t \in A^n} p_t^{(i-1)} f_t m_{t \to s}}{\sum_{t \in A^n} p_t^{(i-1)} f_t}. \tag{4}$$

*Then the limit $p^{\mathsf{EQ}} = \lim_{i \to \infty} p^{(i)}$ exists and satisfies the equilibrium condition (1).*

*Proof.* As in the proof of Theorem 1 we represent the equilibrium problem as the eigenvector problem for matrix $E$. We have to prove that our iterative process converges to the only positive eigenvector of $E$, which gives us the equilibrium distribution according to Theorem 1. Since by Perron-Frobenius theorem the positive eigenvector corresponds to the largest eigenvalue, we can apply the

power iteration method to find vector $p^{\mathsf{EQ}}$. At step $i$ the original power iteration method divides the vector $Ep^{(i)}$ by its length, whereas in our iterative definition (4) we divide it by the sum of its components to obtain a probability distribution. Both methods are just different normalizations, and reach the same equilibrium eigenvector, up to a multiplicative constant. Thus, the limit distribution $p^{\mathsf{EQ}}$ exists and satisfies the equilibrium condition (1). □

### 2.3 Substring Distribution at Equilibrium

For studying the substring distribution we first introduce some notation. For two strings $a$ and $b$ we denote their concatenation by $a \cdot b$. For any sequence $s$ let $s_j^k$ denote its substring of length $k$ starting at location $j$. We regard sequences as cyclic, that is, the substring that reaches the end of the sequence wraps to continue from the beginning. Further we define the shift operator "$\gg$", such that $s \gg i$ denotes the sequence obtained from $s$ by removing the last $i$ nucleotides and inserting these at the beginning.

In order to express the substring distribution in a usable form, we need to make assumptions about the fitness function $f$ and mutation probabilities $m$. Namely, for $f$ we assume shift invariance, that is $f_s = f_{s \gg i}$ for all $s \in A^n$ and $i \in \mathbb{N}$. This holds, for example, if fitness is measured by the number of occurences of some substring in the sequence. For $m$ we assume that $m_{a \cdot s \to b \cdot t} = m_{a \to b} \cdot m_{s \to t}$ for all $1 \le k \le n$, $a, b \in A^k$ and $s, t \in A^{n-k}$. In other words, we assume that mutations at different parts of the sequence are independent. From this it follows that $m$ is also shift invariant. As the proportions in the equilibrium population are computed directly from the fitness function and mutation probabilities, these must also be shift invariant, that is, $p_s^{\mathsf{EQ}} = p_{s \gg i}^{\mathsf{EQ}}$.

Suppose we now pick a substring of length $k$ from a random location in the sequence of a random individual from the equilibrium population. The probability to get substring $a$ can be calculated as follows:

$$\Pr(a) = \sum_{s \in A^n} p_s^{\mathsf{EQ}} \sum_{j=1}^{n} \frac{1}{n} \cdot [s_j^k = a] = \sum_{s \in A^n} \frac{1}{n} \sum_{j=1}^{n} p_s^{\mathsf{EQ}} \cdot [s_j^k = a],$$

where $[s_j^k = a]$ is defined as 1 if $s_j^k = a$ and 0 otherwise. Note that $s_j^k = a$ if and only if $s = (a \cdot t) \gg j$ for some $t \in A^{n-k}$. The above equality can now be rewritten:

$$\Pr(a) = \sum_{t \in A^{n-k}} \frac{1}{n} \sum_{j=1}^{n} p_{(a \cdot t) \gg j}^{\mathsf{EQ}} = \sum_{t \in A^{n-k}} \frac{1}{n} \sum_{j=1}^{n} p_{a \cdot t}^{\mathsf{EQ}} = \sum_{t \in A^{n-k}} p_{a \cdot t}^{\mathsf{EQ}}.$$

In other words, the probability of $k$-mer $a$ in the equilibrium substring distribution is equal to the total proportion of all sequences starting with $a$. Due to this fact we introduce the following notation:

$$p_a^{\mathsf{EQ}} := \Pr(a) = \sum_{t \in A^{n-k}} p_{a \cdot t}^{\mathsf{EQ}}. \tag{5}$$

Computing the equilibrium substring distribution directly from (4) and (5) is intractable as the time complexity is exponential in $n$. Therefore, we propose an alternative method for estimating this distribution. For each $k$-mer $a$, we can write:

$$p_a^{\mathsf{EQ}} \overset{(5)}{=} \sum_{s\in A^{n-k}} p_{a\cdot s}^{\mathsf{EQ}} \overset{(1)}{=} \sum_{s\in A^{n-k}} \frac{\sum_{t\in A^n} p_t^{\mathsf{EQ}} f_t m_{t\to a\cdot s}}{\sum_{t\in A^n} p_t^{\mathsf{EQ}} f_t} \overset{(t=b\cdot u)}{=} \frac{\sum_{s\in A^{n-k}} \sum_{b\in A^k} \sum_{u\in A^{n-k}} p_{b\cdot u}^{\mathsf{EQ}} f_{b\cdot u} m_{b\cdot u\to a\cdot s}}{\sum_{b\in A^k} \sum_{u\in A^{n-k}} p_{b\cdot u}^{\mathsf{EQ}} f_{b\cdot u}} =$$
$$= \frac{\sum_{s\in A^{n-k}} \sum_{b\in A^k} \sum_{u\in A^{n-k}} p_{b\cdot u}^{\mathsf{EQ}} f_{b\cdot u} m_{b\to a} m_{u\to s}}{\sum_{b\in A^k} \sum_{u\in A^{n-k}} p_{b\cdot u}^{\mathsf{EQ}} f_{b\cdot u}} = \frac{\sum_{b\in A^k} m_{b\to a} \sum_{u\in A^{n-k}} p_{b\cdot u}^{\mathsf{EQ}} f_{b\cdot u} \sum_{s\in A^{n-k}} m_{u\to s}}{\sum_{b\in A^k} \sum_{u\in A^{n-k}} p_{b\cdot u}^{\mathsf{EQ}} f_{b\cdot u}} =$$
$$= \frac{\sum_{b\in A^k} m_{b\to a} \sum_{u\in A^{n-k}} p_{b\cdot u}^{\mathsf{EQ}} f_{b\cdot u}}{\sum_{b\in A^k} \sum_{u\in A^{n-k}} p_{b\cdot u}^{\mathsf{EQ}} f_{b\cdot u}}.$$

We now approximate

$$\sum_{u\in A^{n-k}} p_{b\cdot u}^{\mathsf{EQ}} f_{b\cdot u} \approx \frac{p_b^{\mathsf{EQ}}}{|A^{n-k}|} \sum_{u\in A^{n-k}} f_{b\cdot u}\,, \tag{6}$$

which essentially means replacing all terms $p_{b\cdot u}^{\mathsf{EQ}}$ within the sum by their average over $u \in A^{n-k}$. Although not strongly supported by theoretical considerations, the numerical experiments indicate that this approximation is quite good. We further denote $f_b := \frac{1}{|A^{n-k}|} \sum_{u\in A^{n-k}} f_{b\cdot u}$, as this is the average fitness of all sequences starting with substring $b$. Note that this does not conflict with the notation of the original fitness function, as for the full sequence $s$ the sum on the right has only one element, $f_s$ itself. The approximation (6) can now we be written down as follows:

$$p_a^{\mathsf{EQ}} \approx \frac{\sum_{b\in A^k} m_{b\to a} p_b^{\mathsf{EQ}} f_b}{\sum_{b\in A^k} p_b^{\mathsf{EQ}} f_b}.$$

which matches exactly the original equilibrium condition (1), yet now it is for substrings of length $k$. According to Theorem 1 there exists a unique distribution $p^{\mathsf{EQ}_k}$, which for any $k$-mer $a$ satisfies the following condition:

$$p_a^{\mathsf{EQ}_k} = \frac{\sum_{b\in A^k} p_b^{\mathsf{EQ}_k} \cdot f_b m_{b\to a}}{\sum_{b\in A^k} p_b^{\mathsf{EQ}_k} f_b}.$$

It is therefore natural to use $p_a^{\mathsf{EQ}_k}$ as an approximation to $p_a^{\mathsf{EQ}}$.

The practical calculation of $p^{\mathsf{EQ}_k}$ can now be performed in two steps:

- estimating $f_b = \frac{1}{|A^{n-k}|} \sum_{u \in A^{n-k}} f_{b \cdot u}$ for all $b \in A^k$ by random sampling over $u$;
- finding $p^{\mathsf{EQ}_k}$ using Theorem 2, *i.e.* using power iteration on a $|A|^k \times |A|^k$ matrix.

For the case of DNA sequences ($|A| = 4$) this calculation is realistic up to $k = 8$ or so.

## 3 Experiments

In order to check the applicability of our approximations we have performed experiments to compare the distributions $p^{\mathsf{EQ}}$ and $p^{\mathsf{EQ}_k}$. As calculating the exact distribution $p^{\mathsf{EQ}}$ is computationally very demanding, we restricted ourselves to the alphabet of size 2, *i.e.* $A = \{\mathsf{A}, \mathsf{C}\}$, and to the promoters of length $n = 8$. At each site independently, the probability of a mutation from $\mathsf{A}$ to $\mathsf{C}$ or from $\mathsf{C}$ to $\mathsf{A}$ was $r$, we tested values $r = 10^{-0.4}, 10^{-0.6}, 10^{-0.8}, \ldots 10^{-3.0}$. As explained later, smaller values of $r$ lead to very similar results due to convergence of the distribution. Because of independent point-mutations the probability of a sequence $s$ mutating into sequence $t$ was $m_{s \to t} = r^{\Delta(s,t)}(1-r)^{n-\Delta(s,t)}$, where $\Delta(s,t)$ is the number of positions where $s$ and $t$ have a different nucleotide. The fitness of a sequence was dependent on the number of times a certain substring $q$ occurred in the sequence, we tested $q = \mathsf{AC}, \mathsf{AAC}, \mathsf{AACA}$. Altogether we used nine different measures $f$ as for each of the substrings $q$ we tested the following three strategies:

(S1) sequences with $i$ occurrences of $q$ had fitness $i + 1$;
(S2) sequences with 0 or 1 occurrences of $q$ had fitness 1, others had fitness 2;
(S3) sequences with 0 occurrences of $q$ had fitness 2, others had fitness 1.

For each mutation rate $r$ (14 values), each fitness function $f$ (9 values), and each $k = 1, 2, \ldots, 6$ we found the $k$-mer distribution for the exact equilibrium $p^{\mathsf{EQ}}$ and the approximation $p^{\mathsf{EQ}_k}$, altogether $14 \cdot 9 \cdot 6 = 756$ pairs of distributions.

To evaluate the approximated distribution we found the Pearson correlation coefficient with the exact distribution as well as the Kullback-Leibler divergence per position ($\mathsf{KLdpp}$) defined as follows:

$$\mathsf{KLdpp} = \frac{D_{\mathsf{KL}}(p^{\mathsf{EQ}} || p^{\mathsf{EQ}_k})}{k} = \frac{1}{k} \sum_{a \in A^k} p_a^{\mathsf{EQ}} \log \frac{p_a^{\mathsf{EQ}}}{p_a^{\mathsf{EQ}_k}}.$$

The results did not show significant dependence of approximation error on the choice of the fitness function and value $k$. The precision of approximation was mainly dependent on the mutation probability $r$. Figure 1 plots the correlation and $\mathsf{KLdpp}$ for all experiments for different values of $r$.

To get some idea about how the exact and approximate distributions change with $r$ we plotted the distributions for $q = \mathsf{AAC}$, fitness strategy (S2), $n = 8$,

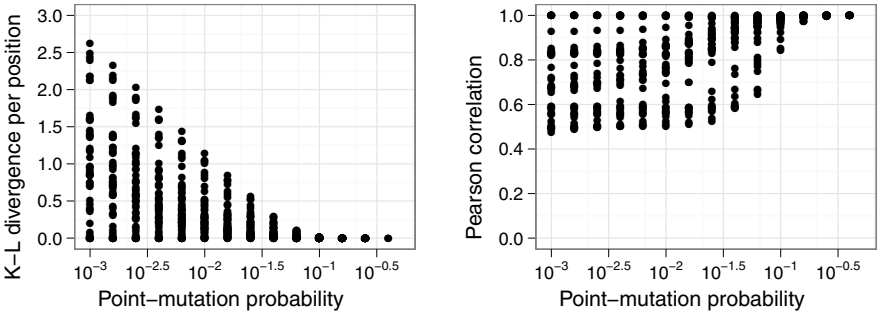



**Fig. 1.** The effect of point-mutation rate $r$ on the approximation quality measured as Kullback-Leibler divergence per position and correlation between the exact and approximated distributions. Each circle denotes an experiment with a different set of parameters.

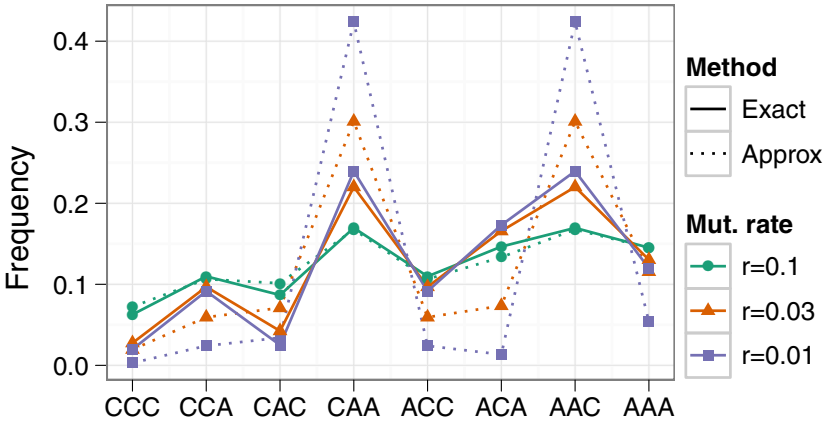



**Fig. 2.** The exact and approximate 3-mer distributions for point-mutation rates $r = 0.1, 0.03, 0.01$ where fitness is defined with strategy (S2) for substring $q = \mathsf{AAC}$

$k = 3$, and $r = 0.1, 0.03, 0.01$ (see Figure 2). For mutation rate $r = 0.1$ the approximation is almost perfect and is gradually becoming worse with decreasing $r$. Still, the correlation between the exact and approximate distribution remains quite high, which is confirmed in Figure 1 where all correlations are above 0.4. High Kullback-Leibler divergence for small values of $r$ is apparently caused by the substrings with moderate true frequency but very low approximated frequency, such as ACA in Figure 2.

Figure 2 also illustrates the convergence of the exact distributon with decreasing $r$, as the distributions of $r = 0.03$ and $r = 0.01$ are highly similar. The same holds for the approximated distributions, explaining why approximation errors for $r = 10^{-2.8}$ and $r = 10^{-3}$ have extremely similar patterns in Figure 1.

## 4 Discussion

In this work we presented a novel approach to modeling DNA substring distribution that is based on an evolutionary model. We have derived a computationally tractable approximation for estimating the $k$-mer distribution from the description of the process, and verified that the approximation is fairly precise. This allows to use the model in generative settings as well as for significance computations in motif discovery procedures.

The major merit of the approach lies in the fact that it provides a sound way of incorporating evolutionary assumptions and prior knowledge into further analyses. Introduction of such *inductive bias* opens up novel possibilities of application for sequence analysis algorithms. To be more precise, consider the case of motif discovery from promoter sequences. This task is often solved by searching given DNA sequence data for short *significantly overrepresented* substrings [8]. Significance here denotes a measure of deviation of observed substring frequencies from a certain *null-model* – a presumed distribution of substrings, which could be explained using prior knowledge only. For example, if we presume that a given DNA region is inherently rich in `CG`-pairs, we shall not be surprised to find that a substring `CGCGCG` is frequent. On the other hand, detecting a similarly frequent substring `ATATAT` might be interpreted as a presence of something, which cannot be explained using previous knowledge only, *i.e.* an overrepresented motif.

Many contemporary motif discovery methods are rather unsophisticated in the way of modeling prior knowledge, using just the single- or di-nucleotide distribution for their *background model* [9]. This may result in spurious discoveries, such as detecting multiple versions of a single motif or just some generic sequence features. Other methods use the set of *background sequences* [10,11], or a higher-order HMM [12,3] to model prior knowledge. The drawback of this approach is that it requires many sequences "of the same kind" to estimate the model. Yet it is often not clear which regulatory sequences may be modeled as being from the same kind. Therefore further assumptions, such as co-regulation or co-expression must be made. In our method we essentially provide a set of *evolutionary* assumptions, which may be used instead. Given these assumptions only, a background model of $k$-mer distribution can be computed, incorporating the information about which sequence features are already known to be significant.

Another natural way of regarding our method is just as a purely generative model for DNA sequences. Indeed, the marginal $k$-mer distribution can be straightforwardly extended to a generator of arbitrary-length substrings satisfying this distribution [3]. The need for such generators of "random" DNA sequences arises often in connection with testing and analysis of various algorithms, and a number of tools have already been developed for this purpose. Some of these proceed by simulating evolution [13], some focus on simulating alignments [14], and yet others propose ways of planting randomized motifs [15]. Our model can account for motifs and evolutionary aspects simultanously through the fitness and mutation functions. For modelling gene promoter regions our model can in principle aggregate such information as the transcription factor binding motifs and their combinations [1], nucleosome positioning code [16] and CpG mutation

rates [17]. While defining the fitness function is mostly a biological endeavor, the mutation function can require more mathematical effort, as exemplified in the Experiments section.

The open problem which yet remains to be solved is the question of efficient estimation of model parameters from data, as this could open new possibilities and application areas both in sequence analysis as well as the study of DNA evolution. Or in other words, what information can we extract from the substring distributions, assuming genetic equilibrium? Another interesting question is related to the possibility of improving the precision of the approximation (6), especially for smaller mutation rates, perhaps by incorporating higher-order terms, and yet still keeping the computations tractable.

## References

1. Davidson, E.H.: The regulatory genome: gene regulatory networks in development and evolution. Academic Press, San Diego (2006)
2. Stormo, G.D.: DNA binding sites: representation and discovery. Bioinformatics 16(1), 16–23 (2000)
3. Thijs, G., Lescot, M., Marchal, K., Rombauts, S., Moor, B.D., Rouzé, P., Moreau, Y.: A higher-order background model improves the detection of promoter regulatory elements by Gibbs sampling. Bioinformatics 17(12), 1113–1122 (2001)
4. Mustonen, V., Lässig, M.: Evolutionary population genetics of promoters: predicting binding sites and functional phylogenies. Proc. Natl. Acad. Sci. USA 102(44), 15936–15941 (2005)
5. Moses, A.M., Pollard, D.A., Nix, D.A., Iyer, V.N., Li, X.Y., Biggin, M.D., Eisen, M.B.: Large-scale turnover of functional transcription factor binding sites in Drosophila. PLoS Comput. Biol. 2(10), e130 (2006)
6. Doniger, S.W., Fay, J.C.: Frequent gain and loss of functional transcription factor binding sites. PLoS Comput. Biol. 3(5), e99 (2007)
7. Huang, W., Nevins, J.R., Ohler, U.: Phylogenetic simulation of promoter evolution: estimation and modeling of binding site turnover events and assessment of their impact on alignment tools. Genome. Biol. 8(10), R225 (2007)
8. Brazma, A., Jonassen, I., Vilo, J., Ukkonen, E.: Predicting gene regulatory elements in silico on a genomic scale. Genome. Res. 8(11), 1202–1215 (1998)
9. Das, M.K., Dai, H.K.: A survey of DNA motif finding algorithms. BMC Bioinformatics 8(Suppl. 7), S21 (2007)
10. Redhead, E., Bailey, T.: Discriminative motif discovery in DNA and protein sequences using the DEME algorithm. BMC Bioinformatics 8(1), 385 (2007)
11. Vilo, J.: Pattern discovery from biosequences. Thesis PhD (2002)
12. Wang, G., Yu, T., Zhang, W.: WordSpy: identifying transcription factor binding motifs by building a dictionary and learning a grammar. Nucleic Acids Res. 33(Web Server issue), W412–W416 (2005)
13. Cartwright, R.A.: DNA assembly with gaps (Dawg): simulating sequence evolution. Bioinformatics 21(Suppl. 3), iii31–iii38 (2005)
14. Varadarajan, A., Bradley, R., Holmes, I.: Tools for simulating evolution of aligned genomic regions with integrated parameter estimation. Genome. Biol. 9(10), R147 (2008)

15. Rouchka, E.C., Hardin, C.T.: rMotifGen: random motif generator for DNA and protein sequences. BMC Bioinformatics 8, 292 (2007)
16. Segal, E., Fondufe-Mittendorf, Y., Chen, L., Thåström, A., Field, Y., Moore, I.K., Wang, J.P.Z., Widom, J.: A genomic code for nucleosome positioning. Nature 442(7104), 772–778 (2006)
17. Saxonov, S., Berg, P., Brutlag, D.L.: A genome-wide analysis of CpG dinucleotides in the human genome distinguishes two distinct classes of promoters. Proc. Natl. Acad. Sci. USA 103(5), 1412–1417 (2006)

# Indexing a Dictionary for Subset Matching Queries

Gad M. Landau[1,⋆], Dekel Tsur[2], and Oren Weimann[3]

[1] Department of Computer Science, University of Haifa, Haifa 31905, Israel
and
Department of Computer Science and Engineering, NYU-Poly,
Six MetroTech Center, Brooklyn, NY 11201-3840
landau@cs.haifa.ac.il
[2] Department of Computer Science, Ben-Gurion University, Beer-Sheva, Israel
dekelts@cs.bgu.ac.il
[3] Faculty of Mathematics and Computer Science, Weizmann Institute of Science,
Rehovot, Israel
oren.weimann@weizmann.ac.il

**Abstract.** We consider a subset matching variant of the *Dictionary Query* problem. Consider a dictionary $D$ of $n$ strings, where each string location contains a set of characters drawn from some alphabet $\Sigma = \{1, ..., |\Sigma|\}$. Our goal is to preprocess $D$ so when given a query pattern $p$, where each location in $p$ contains a single character from $\Sigma$, we answer if $p$ matches to $D$. $p$ is said to match to $D$ if there is some $s \in D$ where $|p| = |s|$ and $p[i] \in s[i]$ for every $1 \leq i \leq |p|$.

To achieve a query time of $O(|p|)$, we construct a compressed trie of all possible patterns that appear in $D$. Assuming that for every $s \in D$ there are at most $k$ locations where $|s[i]| > 1$, we present two constructions of the trie that yield a preprocessing time of $O(nm+|\Sigma|^k n \log(\min\{n, m\}))$, where $n$ is the number of strings in $D$ and $m$ is the maximum length of a string in $D$. The first construction is based on divide and conquer and the second construction uses ideas introduced in [2] for text fingerprinting. Furthermore, we show how to obtain $O(nm + |\Sigma|^k n + |\Sigma|^{k/2} n \log(\min\{n, m\}))$ preprocessing time and $O(|p| \log\log|\Sigma| + \min\{|p|, \log(|\Sigma|^k n)\} \log\log(|\Sigma|^k n))$ query time by cutting the dictionary strings and constructing two compressed tries.

Our problem is motivated by haplotype inference from a library of genotypes [13, 16]. There, $D$ is a known library of genotypes ($|\Sigma| = 2$), and $p$ is a haplotype. Indexing all possible haplotypes that can be inferred from $D$ as well as gathering statistical information about them can be used to accelerate various haplotype inference algorithms.

## 1 Introduction

In the *Dictionary Query* problem, one is given a set $D$ of strings $s_1, \ldots, s_n$ and subsequent queries ask whether a given query pattern $p$ appears in $D$. In [5],

⋆ Partially supported by the National Science Foundation Award 0904 246, Israel Science Foundation grants 35/05 and 347/09, the Israel-Korea Scientific Research Cooperation, Yahoo, Grant No. 2008217 from the United States-Israel Binational Science Foundation (BSF) and DFG.

T. Elomaa et al. (Eds.): Ukkonen Festschrift, LNCS 6060, pp. 158–169, 2010.

this paradigm was broaden to allow a bounded number of mismatches, or allow a bounded number of "don't care" characters. We further extend dictionary queries to support a restricted version of *subset matching*. In subset matching, the characters are subsets of some alphabet $\Sigma$. A pattern $p$ is said to match a string $s$ of the same length if $p[i] \subseteq s[i]$ for every $1 \leq i \leq |p|$. The subset matching problem of finding all occurrences of a pattern string $p$ in a text string $t$ was solved in $O(N \log^2 N)$ deterministic time [6] and $(N \log N)$ randomized time [21], where $N$ is the sum of sizes of the sets in $p$ and $t$.

In this paper we consider the problem of indexing a dictionary for subset matching queries. We focus on a relaxed version of subset matching requiring that the query pattern is over single characters from $\Sigma$ rather than subsets of $\Sigma$. Formally, the problem we consider is defined as follows. We are given a dictionary $D$ of strings $s_1, \ldots, s_n$ where each string character is a subset of some alphabet $\Sigma$. A query $p$ is a string over the alphabet $\Sigma$, and we say that $p$ matches to $s_i$ if $|p| = |s_i|$ and $p[j] \in s_i[j]$ for every $1 \leq j \leq |p|$. Our goal is to preprocess $D$ for queries of the form "does $p$ match to a string in $D$?".

Let $m$ denote the length of the longest string in $D$ and let $D'$ be the set of all strings that match to a string in $D$. For example, if $D$ contains two strings, $ab\{c,d\}$ and $ab\{c,d\}g\{a,b,c\}ad$, then $D' = \{abc, abd, abcgaad, abcgbad, abcgcad, abdgaad, abdgbad, abdgcad\}$. Notice that $|D'|$ is bounded by $O(|\Sigma|^k n)$. By storing the dictionary $D'$ in a trie we can efficiently answer membership queries in $O(|p|)$ time for a pattern $p$. A *compressed trie* (i.e. a trie whose internal nodes all have more than one child and whose edges correspond to strings rather than single characters) can be naively constructed in $O(|\Sigma|^k nm)$ time and $O(|\Sigma||D'|)$ space, assuming every $s \in D$ has at most $k$ locations in which $|s[i]| > 1$. The techniques of Cole et al. [5] can be used to solve the problem with $O(nm \log(nm) + n \log^k n/k!)$ preprocessing time, and $O(m + \log^k n \log\log n)$ query time. For small $|\Sigma|$, this approach is less efficient than the compressed trie approach.

In Sections 2 and 3 we present two faster constructions of the trie. The first construction is based on divide and conquer and requires $O(nm + |\Sigma|^k n \log n)$ preprocessing time. The second construction uses ideas introduced in [2] for text fingerprinting and requires $O(nm + |\Sigma|^k n \log m)$ preprocessing time. The space complexity is $O(|\Sigma||D'|)$, and it can be reduced to $O(|D'|)$ by using suffix tray [7] ideas. Intuitively, a suffix tray is a combination of a suffix tree and a suffix array where in some suffix tree nodes we store and array of length $|\Sigma|$ of children pointers and in some nodes we store two pointers to appropriate intervals in the suffix array. The save in space comes at the cost of $O(|p| + \log\log|\Sigma|)$ query time. In Section 4 we show that by cutting the dictionary strings and constructing two tries we can obtain $O(nm + |\Sigma|^k n + |\Sigma|^{k/2} n \log(\min\{n, m\}))$ preprocessing time at the cost of $O(|p| \log\log|\Sigma| + \min\{|p|, \log|D'|\} \log\log|D'|) = O(|p| \log\log|\Sigma| + \min\{|p|, \log(|\Sigma|^k n)\} \log\log(|\Sigma|^k n))$ query time.

An important feature of our first two trie constructions is that they can calculate the number of appearances in $D$ of each pattern in $D'$ (i.e., which is most common?

which is least common? etc.). This feature is useful in the application of *Haplotype Inference* that we next describe according to the presentation of Gusfield [12].

### 1.1 A Haplotype Trie from a Genotype Dictionary

In diploid organisms such as humans, there are two non-identical copies of each chromosome (except for the sex chromosome). A description of the data from a single copy is called a *haplotype* while a description of the conflated (mixed) data on the two copies is called a *genotype*. The underlying data that forms a haplotype is either the full DNA sequence in the region, or more commonly the values of only DNA positions that are *Single Nucleotide Polymorphisms* (SNP's). A SNP is a position in the genome at which exactly two (of four) nucleotides occur in a large percentage of the population. If we consider only the SNP positions, each position can have one of two nucleotides and a haplotype can thus be represented as a 0/1 vector. A genotype can be represented as a 0/1/2 vector, where 0 means that both copies contain the first nucleotide, 1 means that both copies contain the second nucleotide and 2 means that the two copies contain different nucleotides (but we don't know which copy contains which nucleotide).

The next high-priority phase of human genomics will involve the development and use of a full *Haplotype Map* of the human genome [20]. Unfortunately, it is prohibitively expensive to directly determine the haplotypes of an individual. As a result, almost all population data consists of genotypes and the haplotypes are currently inferred from raw genotype data. The input to the haplotype inference problem consists of $n$ genotypes (0/1/2 vectors), each of length $m$. A solution to the problem associates every genotype with a pair of haplotypes (binary vectors) as follows. For any genotype $g$, the associated binary vectors $v_1, v_2$ must both have value 0 (respectively 1) at any position where $g$ has value 0 (respectively 1); but for any position where $g$ has value 2, exactly one of $v_1, v_2$ must have value 0, while the other has value 1. The haplotypes inference problem has been studied extensively, e.g. [1, 4, 9, 10, 12, 15, 17, 19, 24, 26, 27, 32].

In our settings, the dictionary $D$ corresponds to the library of genotypes, where every genotype location that has the value 2 is replaced by the set $\{0, 1\}$. This way, $|\Sigma| = 2$ and $D'$ consists of all the possible haplotypes that can be part of a pair inferred from $D$. Our trie stores all haplotypes in $D'$ and we can calculate the number of appearances in $D$ of each such haplotype while constructing the trie. The trie can then be used to accelerate haplotype inference algorithms based on the "pure parsimony criteria", greedy heuristics such as "Clarks rule", and EM based algorithms.

## 2 An $O(nm + |\Sigma|^k n \log n)$ Time Construction

In this section we present an $O(nm + |\Sigma|^k n \log n)$ time construction for the compressed trie of $D'$. To simplify the presentation, for the rest of the paper we assume without loss of generality that all strings in $D$ have the same length $m$.

We say that string $s$ is the *longest common prefix* (LCP) of strings $x$ and $y$ if $s$ is the longest string that is a prefix of both $x$ and $y$.

We first describe an algorithm for merging two compressed tries $T_1$ and $T_2$.

1. If one of the tries $T_1$ or $T_2$ has a single vertex, then return a copy of the other trie.
2. If both the roots of $T_1$ and $T_2$ have degree 1, and the labels of the edges leaving the roots of $T_1$ and $T_2$ have a common first letter, then find the longest common prefix (LCP) $p$ of these labels. Remove the string $p$ from $T_1$, that is, if the label of the edge $e$ that leaves the root of $T_1$ is equal to $p$, remove the edge $e$ and the root from $T_1$, and otherwise remove $p$ from the label of $e$. Additionally, remove $p$ from $T_2$.

   Next, recursively merge the two modified tries $T_1$ and $T_2$, and let $T$ be the result of the merge. Add a new root $r$ to $T$ and connect it by an edge to the old root of $T$, where the label of the edge is $p$.
3. If the two cases above do not occur, then split the trie $T_1$ as follows. For every edge $e = (r, v)$ that leaves the root $r$ of $T_1$, create a new trie that contains $r$, $v$, and all the descendents of $v$ in $T_1$. This trie will be denoted $T_1^a$, where $a$ is the first letter in the label of $e$. Similarly, split the trie $T_2$ and create tries $T_2^a$.

   For each letter $a \in \Sigma$, recursively merge the tries $T_1^a$ and $T_2^a$ if these two tries exist. Finally, merge the roots of the merged tries.

If the LCP of two edge labels can be obtained in $O(1)$ time, then the time complexity of this algorithm is $O(|T_1| + |T_2|)$, where $|T|$ denotes the number of vertices in the compressed trie $T$. To perform such LCP queries in $O(1)$ time, we make use of *generalized suffix tree*.

Given a set $X$ of $n$ strings each of length bounded by $m$, a generalized suffix tree is a compressed trie containing all $O(nm)$ suffixes of the strings in $X$. strings in $X$. A generalized suffix tree can be built in $O(nm)$ time (e.g. [11,22,25,29,31]). By building a lowest common ancestor data structure (such as [18]) on the generalized suffix tree, we can support $O(1)$-time LCP queries between pairs of suffixes in $X$.

We now present the algorithm for building a compressed trie of $D'$.

1. For every string in $D$, replace every character that is a set of size greater than one with a new symbol $\phi$.
2. Build a generalized suffix tree $\hat{T}$ for $D$.
3. Build compressed tries $T_1, \ldots, T_n$, where $T_i$ is a compressed trie containing all the patterns that match $s_i$ (recall that $D = \{s_1, \ldots, s_n\}$).
4. Repeat $\lceil \log n \rceil$ times:
   (a) Partition the compressed tries into pairs, except at most one trie.
   (b) Merge each pair of tries into a single trie.

Constructing $\hat{T}$ requires $O(nm)$ time. Each edge label $b$ in some trie that is built during the algorithm, matches a substring $s_i[j..j + |b| - 1]$ of some string $s_i$ in $D$. It is important to notice that $|s_i[l]| = 1$ for every $j + 1 \leq l \leq j + |b| - 1$.

Using the generalized suffix tree $\hat{T}$, computing the LCP of two edge labels takes $O(1)$ time. Therefore, the merging of two compressed tries in the algorithm is performed in linear time. In each iteration of line 4, the total work is linear in the total sizes of the current tries, which is $O(|D'|) = O(|\Sigma|^k n)$. Thus, the overall time complexity of the algorithm is $O(nm + |\Sigma|^k n \log n)$.

## 3 An $O(nm + |\Sigma|^k n \log m)$ Time Construction

In this section we present an $O(nm + |\Sigma|^k n \log m)$ time construction for the compressed trie of $D'$. Consider the lexicographical ordering of all the strings in $D'$. Notice that if we knew this ordering and the length of the LCP of every adjacent strings in this ordering, then we could construct the trie in $O(|D'|) = O(|\Sigma|^k n)$ time by adding the strings in order. We next describe how to obtain the required ordering and LCP information in $O(nm + |\Sigma|^k n \log m)$ time.

We assign a unique integer name to every string in $D'$ such that the names preserve the lexicographical order of $D'$. The names are assigned using a fingerprinting technique [2, 8, 23]. The idea behind fingerprinting is that the name of a string $p$ can be computed fast from the name of a string $q$ that differs from $p$ only in one location.

A *naming table* of a string $p$ is a table of $1 + \log |p|$ rows, where the $i$-th row contains $2^{i-1}$ cells (without loss of generality $|p|$ is a power of two, otherwise, we can extend $p$ until $|p|$ is a power of two by concatenating to $p$ a string of a repeated new character). Each cell in the table is assigned a name. First, the cells in the last row are named by the characters of $p$. Next, the cells of the second last row are named. The name of a cell depends on the names $a_1$ and $a_2$ assigned to the two cells below it. If there was other cell in the current row such that the blocks below it were also named $a_1$ and $a_2$, then the name used for that cell is also given for the current cell. Otherwise, a new name is used. This process is continued with the other rows in the table. See Figure 1(a) for an example.

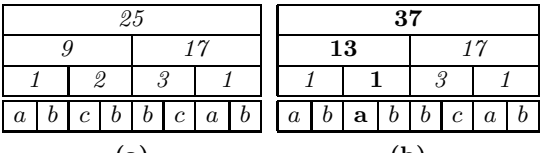



**Fig. 1.** Figure (a) shows a possible naming table for the string $p = abcbbcab$. Note that the first and last cell in the third row have the same name as the names of the cells below these cells are the same ($a$ and $b$). Figure (b) shows a possible naming table for the string $q = ababbcab$ that differs from $p$ in one location. The cells of the naming table of $q$ that differ from the corresponding cells of the naming table of $p$ are marked in bold.

The following property is what makes the naming technique appealing in our settings. Consider two strings $p$ and $q$ that differ only in one location. Then, the naming table of $p$ differs from the naming table of $q$ only in $1 + \log |p|$ cells (see Figure 1(b)).

Consider all the strings that match a specific string $s \in D$. It is possible to enumerate these strings in an order $s^{(1)}, s^{(2)}, \ldots, s^{(r)}$ in which two consecutive strings differ in exactly one location. This means that one can compute names for these strings in $O(m + r \log m)$ time as follows. First build the naming table of $s^{(1)}$ from bottom to top, using a two-dimensional table $B$ to store the names given so far. More precisely, $B[a, b]$ is the name given for the pair $(a, b)$, if the pair $(a, b)$ was named. Since checking whether a pair of names appeared before takes constant time, the time it takes to build the naming table is linear in the number of cells in the table, which is $m + m/2 + m/4 + \cdots + 1 = 2m - 1$. Next, we build the naming table of $s^{(2)}$ by updating $1 + \log m$ cells in the table of $s^{(1)}$, which takes $O(\log m)$ time. Then, we build the naming table of $s^{(3)}$ using the naming table of $s^{(2)}$, and so on.

Applying the naming procedure to all strings in $D$ takes $O(nm + |\Sigma|^k n \log m)$ time. The space complexity is $O((nm + |\Sigma|^k n \log m)^2)$ due to the table $B$. The space complexity can be reduced to $O(nm + |\Sigma|^k n \log m)$ as shown in [8]. The algorithm of [8] uses a different order of filling the naming tables. In the first step, the algorithm computes the names in the second row from the bottom of the naming tables of all strings in $D'$. This is done by taking all pairs of names encountered in the first row of each naming table, lexicographically sorting these pairs, and then naming the pairs. In the second step, the algorithm computes the names in the third row from the bottom of the naming tables of all strings in $D'$, and so on.

After naming all strings in $D'$, we sort these strings according to their names. As noted above, this gives the lexicographical ordering of $D'$. Furthermore, the LCP of any two strings in $D'$ can be computed in $O(\log m)$ time by comparing their naming tables top-down as noticed in [23]. Therefore, we can compute the length of the LCP of every two consecutive strings in the lexicographic ordering of $D'$ in $O(|\Sigma|^k n \log m)$ time, and then construct the trie in $O(|D'|) = O(|\Sigma|^k n)$ time by adding the strings in lexicographical order.

## 4 An $O(nm + |\Sigma|^k n + |\Sigma|^{k/2} n \log(\min\{n, m\}))$ Time Construction

In this section we present a different approach for solving the dictionary query problem. Instead of building one trie, we build two tries. This reduces the construction time, but gives a penalty in the query time.

Let $S$ be a set of integers. For an integer $x$, the *successor* of $x$ in $S$ is the minimal element $y \in S$ such that $y \geq x$. A *successor data-structure* for the set $S$ supports answering queries of the form "Given an integer $x$, what is the successor of $x$ in $S$?". A successor data-structure for a set $S \subseteq \{1, \ldots, U\}$ can be built in $O(|S|)$ time and space such that successor queries are answered in $O(\log \log U)$

time (such a construction is obtained, for example, by combining the van Emde Boas data-structure [30] with the static dictionary of Hagerup et al. [14]).

In order to build a dictionary query data-structure, we split every string in $D$ into two parts. For each $s_i \in D$ define $s'_i$ to be the longest prefix of $s_i$ that contains at most $\lceil k/2 \rceil$ sets of size greater than 1. Also, define $s''_i$ to be the prefix of $s_i^R$ (i.e. the string $s_i$ reversed) of length $m - |s'_i|$. For example, if $k = 2$ and $s_1 = ab\{c,d\}g\{a,b,c\}ad$ then $s'_1 = ab\{c,d\}g$ and $s''_1 = da\{a,b,c\}$.

Let $D_1 = \{s'_1, \ldots, s'_n\}$ and $D_2 = \{s''_1, \ldots, s''_n\}$. For $i = 1, 2$, let $D'_i$ be the set of all strings that match to one of the strings in $D_i$. We wish to reduce the problem of matching a string $p$ against the dictionary $D$ to matching a prefix $p'$ of $p$ against $D_1$, and matching a prefix $p''$ of $p^R$ against $D_2$, with $|p''| = m - |p'|$. However, there are two issues that need to be addressed: (1) It is possible that $p'$ matches a string $s'_i$, while $p''$ matches to a string $s''_j$ with $i \neq j$. This of course does not imply that $p$ matches to a string in $D$. (2) We do not know the length of $p'$, so we need to check all prefixes of $p$ that match to a string in $D_1$.

Let $T_1$ be a compressed trie for $D'_1$ and $T_2$ be a compressed trie for $D'_2$. For each vertex of $T_2$ assign a *name* which is an integer from the set $\{1, \ldots, |T_2|\}$. The name assigned to a vertex $v$ is denoted name($v$). For now we assume that all the names are distinct.

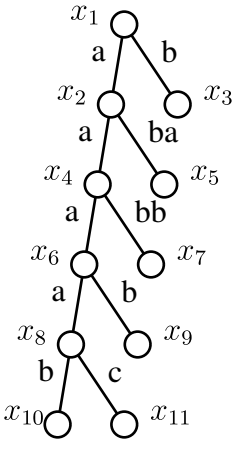
The string that *corresponds* to a vertex $v$ in a trie is the concatenation of the edge labels in the path from the root to $v$. The *depth* of a vertex $v$ in a trie is the length of the strings that corresponds to $v$. We say that the vertices $v \in T_1$ and $w \in T_2$ are *paired* if the sum of their depths is $m$. For a vertex $v$ in $T_1$ (respectively $T_2$) whose corresponding string is $s$, let $L_v$ be the set of all indices $i$ such that $s$ matches to $s'_i$ (respectively $s''_i$). For a vertex $v \in T_1$, let $S_v = \{\text{name}(w) | w \in T_2 \text{ and } L_v \cap L_w \neq \emptyset\}$. See Figure 2 for an example.

The data-structure for the dictionary query problem consists of the tries $T_1$ and $T_2$, and each vertex $v \in T_1$ has a successor data-structure on the set $S_v$. Answering a query is done as follows.

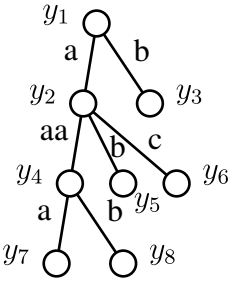
1. Find the longest path $P_1$ in $T_1$ that corresponds to a prefix of the query pattern $p$, and the longest path $P_2$ in $T_2$ that corresponds to prefix of $p^R$.
2. Find all paired vertices $v \in P_1, w \in P_2$ by traversing $P_1$ from top to bottom, while concurrently traversing $P_2$ from bottom to top (note that a vertex $v \in P_1$ is paired with at most one vertex $w \in P_2$).
3. Check whether name($w$) $\in S_v$ for some paired vertices $v \in P_1$ and $w \in P_2$ (by checking whether the successor of name($w$) in $S_v$ is equal to name($w$)).

Answering a dictionary query requires at most $|P_1| \leq m$ queries on the successor data-structures, where each such query takes $O(\log\log|D'|)$ time. Therefore, the time to answer a query is $O(m \log\log|D'|)$.

We now discuss the time complexity of building the tries. The tries $T_1$ and $T_2$ are built using the algorithms in Sections 2 and 3 in $O(nm + |\Sigma|^{k/2} n \log(\min(n, m)))$ time. In order to build the sets $S_v$ for all $v$, compute the intersections $L_v \cap L_w$ for all $v$ and $w$. This is done as follows. For each $i$ from 1 to $n$, go over all vertices $v \in T_1$ such that $i \in L_v$. For each such $v$, go over all $w \in T_2$ such that $i \in L_w$, and add the pair (name($w$), $i$) to a list $I_v$ that is stored at $v$. Then, for each




(a)




(b)

| $v$ | $L_v$ | $S_v$ (1) | $S_v$ (2) |
|---|---|---|---|
| $x_1$ | $\emptyset$ | $\emptyset$ | $\emptyset$ |
| $x_2$ | $\{1\}$ | $\{7,8\}$ | $\{1,2\}$ |
| $x_3$ | $\{1\}$ | $\{7,8\}$ | $\{1,2\}$ |
| $x_4$ | $\emptyset$ | $\emptyset$ | $\emptyset$ |
| $x_5$ | $\{2\}$ | $\{5,6\}$ | $\{3,4\}$ |
| $x_6$ | $\{2\}$ | $\{5,6\}$ | $\{3,4\}$ |
| $x_7$ | $\{3\}$ | $\{2,3\}$ | $\{1,5\}$ |
| $x_8$ | $\emptyset$ | $\emptyset$ | $\emptyset$ |
| $x_9$ | $\{3\}$ | $\{2,3\}$ | $\{1,5\}$ |
| $x_{10}$ | $\{4\}$ | $\{1\}$ | $\{1\}$ |
| $x_{11}$ | $\{5\}$ | $\{1\}$ | $\{1\}$ |

(c)

| $v$ | $L_v$ |
|---|---|
| $y_1$ | $\{4,5\}$ |
| $y_2$ | $\{3\}$ |
| $y_3$ | $\{3\}$ |
| $y_4$ | $\emptyset$ |
| $y_5$ | $\{2\}$ |
| $y_6$ | $\{2\}$ |
| $y_7$ | $\{1\}$ |
| $y_8$ | $\{1\}$ |

(d)

**Fig. 2.** An example of the data-structure for the input strings $s_1 = \{a,b\}\{a,b\}aaa$, $s_2 = a\{a,b\}a\{b,c\}a$, $s_3 = aa\{a,b\}b\{a,b\}$, $s_4 = aaaab$, and $s_5 = aaaac$. The tries $T_1$ and $T_2$ are shown in Figures (a) and (b), respectively. The sets $L_v$ for vertices of $T_1$ and $T_2$ are shown in Figures (c) and (d), respectively. Moreover, Figure (c) shows the sets $S_v$ for $v \in T_1$, where the naming of the vertices is done according to two naming schemes: (1) name$(y_i) = i$ (2) naming according to the heavy path decomposition $Q_1 = [y_1, y_2, y_4, y_8]$, $Q_2 = [y_7]$, $Q_3 = [y_5]$, $Q_4 = [y_6]$, and $Q_5 = [y_3]$ of $T_2$.

$v \in T_1$, lexicographically sort the list $I_v$ and obtain all the intersections involving $v$. Therefore, computing all the intersections and building the successor data-structures takes $O(|\Sigma|^k n)$ time. The total preprocessing time is $O(nm + |\Sigma|^k n + |\Sigma|^{k/2} n \log(\min\{n,m\}))$.

In order to speed up the query time, we use the technique of *fractional cascading* [3]. Fractional cascading is a method for efficiently searching for the same element in several successor data structures. Using a variant of this technique that is described in the next section, we can preprocess $T_1$ such that performing a successor query $x$ on all the successor data structures of the vertices of some path $P$ in $T_1$ is done in $O(|P| \log\log|\Sigma| + \log\log|D'|)$ time. Recall that in order to answer a query, we need to query for name$(w)$ in the successor data-structures of $v$ for every paired vertices $v \in P_1$ and $w \in P_2$. In order to use the fractional cascading speedup, we need to decrease the number of names assigned to the vertices of $P_2$. Note that we can assign the same name to several vertices of $T_2$ if their corresponding strings have different lengths. Thus, we partition the vertices of $T_2$ into paths $Q_1, \ldots, Q_r$ using a *heavy path decomposition* [18].

A heavy path decomposition $T_2$ is as follows. For each node $v$ define its size to be the size of the subtree rooted at $v$. For every internal node $v$ we pick a child of maximum size and classify it as heavy. The remaining children are light. An edge to a light child is a *light edge*. Removing the light edges we obtain the decomposition of $T_2$ into paths. This decomposition has the important property

that a path from some vertex of $T_2$ to the root passes through at most $\log |T_2|$ different paths in the decomposition.

We now assign names to the vertices of $T_2$ according to the heavy path decomposition: The name of a vertex $w$ is the index $i$ such that $w \in Q_i$.

Now, answering a query is done as follows.

1. Find the longest path $P_1$ in $T_1$ that corresponds to a prefix of the query pattern $p$, and the longest path $P_2$ in $T_2$ that corresponds to prefix of $p^R$.
2. For $i = 1, \ldots, r$, let $v_i^{\text{high}}$ (respectively $v_i^{\text{low}}$) be the highest (respectively lowest) vertex in $P_1$ that is paired with a vertex $w \in P_2 \cap Q_i$, if there is such a vertex.
3. For every $i$ such that $v_i^{\text{high}}$ is defined, let $P_{1,i}$ be the path from $v_i^{\text{high}}$ to $v_i^{\text{low}}$,
4. For every path $P_{1,i}$, perform a successor query with the integer $i$ on the successor data-structures of the vertices in $P_{1,i}$ using fractional cascading.

For example, consider the query $p = aaaaa$ on the structure in Figure 2. We have that $P_1 = [x_1, x_2, x_4, x_6, x_8]$ and $P_2 = [y_1, y_2, y_4, y_7]$. Moreover, $v_1^{\text{high}} = x_4$, $v_1^{\text{low}} = x_6$, and $v_2^{\text{high}} = v_2^{\text{low}} = x_2$, so $P_{1,1} = [x_4, x_6, x_8]$ and $P_{1,2} = [x_2]$.

We have that there are at most $\min\{m, \log |T_2|\} = O(\min\{m, \log |D'|\})$ different names assigned to the vertices of $P_2$. Therefore, the number of $P_{1,i}$ paths is $O(\min\{m, \log |D'|\})$. Since the $P_{1,i}$ paths are disjoint, it follows that the time to answer a dictionary query is $O(m \log\log |\Sigma| + \min\{m, \log |D'|\} \log\log |D'|)$.

### 4.1 Fractional Cascading

Let $T$ be a rooted tree of maximum degree $d$. Each vertex $v$ of $T$ has a set $C_v \subseteq \{1, \ldots, U\}$. The goal is to preprocess $T$ in order to answer the following queries "given a connected subtree $T'$ of $T$ and an integer $x$, find the successor of $x$ in $C_v$ for every $v \in T'$". The fractional cascading technique of [3] gives search time of $O(|T'| \log d + \log\log U)$, with linear time preprocessing. We now present a variant of fractional cascading that gives $O(|T'| \log\log d + \log\log U)$ search time (our construction is similar to the one in [28]).

The preprocessing of $T$ is as follows. Traverse the vertices of $T$ in postorder, and for each vertex $v$ of $T$ construct a list $A_v$ whose elements are kept in a non-decreasing order. For a leaf $v$, $A_v$ contains exactly the elements of $C_v$. For an internal vertex $v$, $A_v$ contains all the elements of $C_v$. Additionally, for every child $w$ of $v$, $A_v$ contains every second element of $A_w$. Each element of $A_v$ stores a pointer to its successor in the set $C_v$. An element of $A_v$ which came from a set $A_w$ keeps a pointer to its copy in $A_w$. This pointer is called a *$w$-bridge*. For every vertex $v$, the elements of $A_v$ are stored in a successor data-structures.

Handling a query $(T', x)$ is done by finding the successor of $x$ in each set $A_v$ for $v \in T'$. Then, using the successor pointers, the successor of $x$ in each set $C_v$ is obtained. Finding the successor of $x$ in each set $A_v$ for $v \in T'$ is done by traversing the vertices of $T'$ in postorder. For the root $r$ of $T'$, finding the successor of $x$ in $A_r$ is done by making a successor query on the successor structure of $A_r$. Suppose we have found the successor $y$ of $x$ in $A_v$ and we now wish to find the successor of $x$ in $A_w$, where $w$ is a child of $v$. Let $z$ be the

first element that appears after $y$ in $A_v$ that has a $w$-bridge, and let $z'$ be the elements in $A_w$ pointed to by the $w$-bridge of $z$. Then, the successor of $x$ in $A_w$ is either $z'$ or the element preceding $z'$ in $A_w$.

In order to efficiently find the first $w$-bridge after some element of $A_v$, perform additional preprocessing as follows. Partition the elements of each list $A_v$ into blocks $B_v^1, B_v^2, \ldots, B_v^{\lceil |A_v|/d \rceil}$ of $d$ consecutive elements each (except perhaps the last block). Let $w_1, \ldots, w_{d'}$ be the children of $v$. For each block $B_v^i$ build an array $L_v^i$, where $L_v^i[j]$ is the location of the first $w_j$-bridge that appears in the blocks $B_v^{i+1}, B_v^{i+2}, \ldots, B_v^{\lceil |A_v|/d \rceil}$. Moreover, for all $j$, build a successor data-structures $S_v^{i,j}$ that contains the indices of the elements of the block $B_v^i$ that have a $w_j$-bridge.

Find the first $w_j$-bridge after some element of $A_v$ takes $O(\log \log d)$ time. Therefore, the time complexity of answering a successor query is $O(|T'| \log \log d + \log \log U)$.

## 5 Conclusion and Open Problems

We have shown two solutions for the subset dictionary query problem: one based on building a trie for $D'$ and one based on building two tries. We conjecture that the trie of $D'$ can be built in $O(nm + |\Sigma|^k n)$ time.

## References


1. Abecasis, G.R., Martin, R., Lewitzky, S.: Estimation of haplotype frequencies from diploid data. American Journal of Human Genetics, 69(4 Suppl. 1):114 (2001)
2. Amir, A., Apostolico, A., Landau, G.M., Satta, G.: Efficient text fingerprinting via parikh mapping. J. of Discrete Algorithms 1(5-6), 409–421 (2003)
3. Chazelle, B., Guibas, L.J.: Fractional cascading: I. a data structuring technique. Algorithmica 1(2), 133–162 (1986)
4. Clark, A.G.: Inference of haplotypes from PCR-amplified samples of diploid population. Molecular Biology and Evolution 7(2), 111–122 (1990)
5. Cole, R., Gottlieb, L., Lewenstein, M.: Dictionary matching and indexing with errors and don't cares. In: Proc. 36th ACM Symposium on Theory of Computing (STOC), pp. 91–100 (2004)
6. Cole, R., Hariharan, R.: Verifying candidate matches in sparse and wildcard matching. In: Proc. 34th ACM Symposium on Theory of Computing (STOC), pp. 592–601 (2002)
7. Cole, R., Kopelowitz, T., Lewenstein, M.: Suffix trays and suffix trists: structures for faster text indexing. In: Bugliesi, M., Preneel, B., Sassone, V., Wegener, I. (eds.) ICALP 2006. LNCS, vol. 4051, pp. 358–369. Springer, Heidelberg (2006)
8. Didier, G., Schmidt, T., Stoye, J., Tsur, D.: Character sets of strings. J. of Discrete Algorithms 5(2), 330–340 (2007)
9. Excoffier, L., Slatkin, M.: Maximum-likelihood estimation of molecular haplotype frequencies in a diploid population. Molecular Biology and Evolution 12(5), 921–927 (1995)
10. Fallin, D., Schork, N.J.: Accuracy of haplotype frequency estimation for biallelic loci, via the expectation-maximization algorithm for unphased diploid genotype data. American Journal of Human Genetics 67(4), 947–959 (2000)

11. Farach-Colton, M., Ferragina, P., Muthukrishnan, S.: On the sorting-complexity of suffix tree construction. J. of the ACM 47(6), 987–1011 (2000)
12. Gusfield, D.: Haplotype inference by pure parsimony. In: Baeza-Yates, R., Chávez, E., Crochemore, M. (eds.) CPM 2003. LNCS, vol. 2676, pp. 144–155. Springer, Heidelberg (2003)
13. Gusfield, D., Orzack, S.H.: Haplotype inference. In: Aluru, S. (ed.) CRC handbook on bioinformatics (2005)
14. Hagerup, T., Miltersen, P.B., Pagh, R.: Deterministic dictionaries. J. of Algorithms 41(1), 69–85 (2001)
15. Hajiaghayi, M.T., Jain, K., Konwar, K., Lau, L.C., Mandoiu, I.I., Vazirani, V.V.: Minimum multicolored subgraph problem in multiplex PCR primer set selection and population haplotyping. In: Alexandrov, V.N., van Albada, G.D., Sloot, P.M.A., Dongarra, J. (eds.) ICCS 2006. LNCS, vol. 3992, pp. 758–766. Springer, Heidelberg (2006)
16. Halldórsson, B.V., Bafna, V., Edwards, N., Lippert, R., Yooseph, S., Istrail, S.: A survey of computational methods for determining haplotypes. In: Istrail, S., Waterman, M.S., Clark, A. (eds.) DIMACS/RECOMB Satellite Workshop 2002. LNCS (LNBI), vol. 2983, pp. 26–47. Springer, Heidelberg (2002)
17. Halperin, E., Karp, R.M.: The minimum-entropy set cover problem. In: Díaz, J., Karhumäki, J., Lepistö, A., Sannella, D. (eds.) ICALP 2004. LNCS, vol. 3142, pp. 733–744. Springer, Heidelberg (2004)
18. Harel, D., Tarjan, R.E.: Fast algorithms for finding nearest common ancestors. SIAM J. on Computing 13(2), 338–355 (1984)
19. Hawley, M.E., Kidd, K.K.: Haplo: A program using the em algorithm to estimate the frequencies of multi-site haplotypes. J. of Heredity 86, 409–411 (1995)
20. Helmuth, L.: Genome research: Map of human genome 3.0. Science 5530(293), 583–585 (2001)
21. Indyk, P.: Faster algorithms for string matching problems: Matching the convolution bound. In: Proc. 39th Symposium on Foundations of Computer Science (FOCS), pp. 166–173 (1998)
22. Kärkkäinen, J., Sanders, P., Burkhardt, S.: Linear work suffix array construction. J. of the ACM 53(6), 918–936 (2006)
23. Kolpakov, R., Raffinot, M.: New algorithms for text fingerprinting. In: Lewenstein, M., Valiente, G. (eds.) CPM 2006. LNCS, vol. 4009, pp. 342–353. Springer, Heidelberg (2006)
24. Long, J.C., Williams, R.C., Urbanek, M.: An E-M algorithm and testing strategy for multiple-locus haplotypes. American Journal of Human Genetics 56(2), 799–810 (1995)
25. McCreight, E.M.: A space-economical suffix tree construction algorithm. J. of the ACM 23, 262–272 (1976)
26. Rastas, P., Koivisto, M., Mannila, H., Ukkonen, E.: A hidden markov technique for haplotype reconstruction. In: Casadio, R., Myers, G. (eds.) WABI 2005. LNCS (LNBI), vol. 3692, pp. 140–151. Springer, Heidelberg (2005)
27. Rastas, P., Ukkonen, E.: Haplotype inference via hierarchical genotype parsing. In: Giancarlo, R., Hannenhalli, S. (eds.) WABI 2007. LNCS (LNBI), vol. 4645, pp. 85–97. Springer, Heidelberg (2007)
28. Shi, Q., JáJá, J.: Novel transformation techniques using Q-heaps with applications to computational geometry. SIAM J. on Computing 34(6), 1471–1492 (2005)

29. Ukkonen, E.: On-line construction of suffix trees. Algorithmica 14(3), 246–260 (1995)
30. van Emde Boas, P.: Preserving order in a forest in less than logarithmic time and linear space. Information Processing Letters 6(3), 80–82 (1977)
31. Weiner, P.: Linear pattern matching algorithm. In: Proc. 14th IEEE Symposium on Switching and Automata Theory, pp. 1–11 (1973)
32. Zhang, P., Sheng, H., Morabia, A., Gilliam, T.C.: Optimal step length EM algorithm (OSLEM) for the estimation of haplotype frequency and its application in lipoprotein lipase genotyping. BMC Bioinformatics 4(3) (2003)

# Transposition and Time-Scale Invariant Geometric Music Retrieval

Kjell Lemström

University of Helsinki
Department of Computer Science
klemstro@cs.helsinki.fi

**Abstract.** This paper considers how to adapt geometric algorithms, developed for content-based music retrieval of symbolically encoded music, to be robust against time deformations required by real-world applications. In this setting, music is represented by sets of points in plane. A matching, pertinent to the application, involves two such sets of points and invariances under translations and time scalings. We give an algorithm for finding exact occurrences, under such a setting, of a given query point set, of size $m$, within a database point set, of size $n$, with running time $O(mn^2 \log n)$; partial occurrences are found in $O(m^2n^2 \log n)$ time. The algorithms resemble the sweepline algorithm introduced in [1].

## 1 Introduction

Query-by-humming is a problem that has fascinated researchers working in the music-retrieval area for over fifteen years. First, the music under investigation was assumed to be monophonic (see Fig. 1) [2], later the term has been used with a wider meaning addressing problems where the task is to search for excerpts of music, resembling a given query pattern, in a large database. Moreover, both the query pattern and the database may be polyphonic, and the query pattern constitutes only a subset of instruments appearing in the database representing possibly a full orchestration of a musical work. Although current audio-based methods can be applied to rudimentary cases where queries are directed to clearly separable melodies, the general setting requires methods based on symbolic representation that are truly capable of dealing with polyphonic subset matching.

To this end, several authors have recently used geometric-based modeling of music [1,3,4,5]. Geometric representations usually also take into account another feature intrinsic to the problem: the matching process ignores extra intervening notes in the database that do not appear in the query pattern. Such extra notes are always present because of the polyphony, various noise sources and musical decorations. There is, however, a notable downside of the current geometric methods: they do not allow distortions in tempo (except for individual outliers that are not even discovered) that are inevitable in the application. Even if the query could be given exactly on tempo, the occurrences in the database would be time-scaled versions of the query (requiring *time-scale invariance*). If the query

T. Elomaa et al. (Eds.): Ukkonen Festschrift, LNCS 6060, pp. 170–181, 2010.

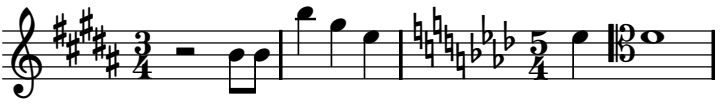


**Fig. 1.** An excerpt of a well-known melody in common music notation. Let us have a closer look at the last bar with a change in key and time signature: The first note is associated with pitch value "Es" (or E flat). It is followed by a c-clef, which looks like a letter "k" to this author. Note also the resemblance of the last note to the letter "o".

is to be given in a live performance, more or less local jittering will inevitably take place and a stronger invariance, namely the *time-warping invariance* [6], would be a desired property for the matching process.

In this paper, new time-scale invariant geometric algorithms that deal with symbolically encoded, polyphonic music will be introduced. We use the pitch-against-time representation of note-on information, as suggested in [5] (see Fig 2). The musical works in a database are concatenated in a single geometrically represented file, denoted by $T$; $T = t_0, t_1, \ldots, t_{n-1}$, where $t_j \in \mathbb{R}^2$ for $0 \leq j \leq n-1$. In a typical retrieval case the query pattern $P$, $P = p_0, p_1, \ldots, p_{m-1}$; $p_i \in \mathbb{R}^2$ for $0 \leq i \leq m-1$, to be searched for is often monophonic and much shorter than the database $T$ to be searched, that is $m \ll n$. It is assumed that $P$ and $T$ are given in the lexicographic order. If this is not the case, the sets can be sorted in $m \log m$ and $n \log n$ times, respectively.

The problems under consideration are modified versions of two problems originally represented in [1]. The following list gives both the original problems and the modifications under consideration; for the partial matches in P2 and S2, one may either use a threshold $\alpha$ to limit the minimum size of an accepted match, or to search for maximally sized matches only.

(P1) Find translations of $P$ such that each point in $P$ matches with a point in $T$.
(P2) Find translations of $P$ that give a partial match of the points in $P$ with the points in $T$.
(S1) Find time-scaled translations of $P$ such that each point in $P$ matches with a point in $T$.
(S2) Find time-scaled translations of $P$ that give a partial match of the points in $P$ with the points in $T$.

Ukkonen et al introduced online algorithms PI and PII solving the original problems P1 and P2 in $O(mn)$ and $O(mn \log m)$ worst case times, respectively, in $O(m)$ space [1]. Lemström et al [7] showed that the practical performance can be improved at least by an order of magnitude by combining sparse indexing and filtering. P2 is known to belong to a problem family for which $o(mn)$ solutions are conjectured not to exist, there is, however, an online approximation algorithm for it running in time $O(n \log n)$. [8]. Romming and Selfridge-Field [9]

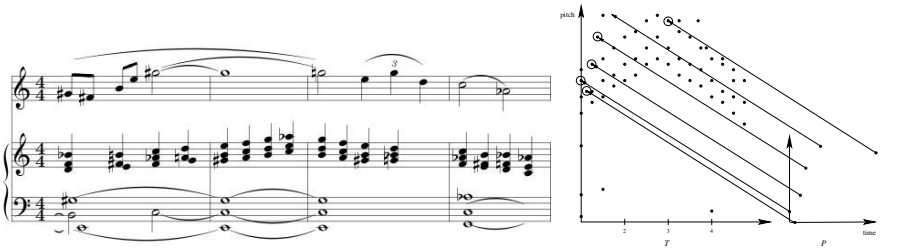



**Fig. 2.** A polyphonic music score, to the left, is represented by a pointset $T$, in the middle, in the geometric representation. Pointset $P$, to the right, corresponds to the first two and a half bars of the melody line (the highest staff of the score) but the fifth point has been delayed somewhat. The depicted trans-set vectors correspond to the translation $f$ that gives the largest partial match of $P$ within $T$.

gave a geometric hashing-based algorithm for S2, which works in $O(n^3)$ space and $O(n^2m^3)$ time.

This paper studies another way to solve S1 and S2. As stated above, in this case, the timing (rhythm) of the music is distorted by a uniform scaling factor; methods ignoring such distortions are called time-scale invariant [6]. The novel time-scale invariant algorithms to be introduced resemble Ukkonen et al's PI and PII algorithms. The new algorithm for S1 runs in time $O(mn^2 \log n)$; the algorithm for S2 in $O(m^2n^2logn)$ time.

## 2 Related Work

Let us denote by $\alpha$ a similarity threshold for P2, and let $p_0, p_1, \ldots, p_{m-1}$ and $t_0, t_1, \ldots, t_{n-1}$ be the pattern and database points, respectively, *lexicographically sorted* according to their co-ordinate values: $p_i < p_{i+1}$ iff $p_i.x < p_{i+1}.x$ or ($p_i.x = p_{i+1}.x$ and $p_i.y < p_{i+1}.y$), and $t_j < t_{j+1}$ iff $t_j.x < t_{j+1}.x$ or ($t_j.x = t_{j+1}.x$ and $t_j.y < t_{j+1}.y$). In our application the elapsing time runs along the horisontal axes, represented by $x$, the perceived height, the *pitch*, is represented by $y$. A translation of $P$ by vector $f$ is denoted as $P + f$: $P + f = p_0 + f, \ldots, p_{m-1} + f$. Using this notation, problem P1 is expressible as the search for a subset $I$ of $T$ and some $f$ such that $P + f = I$. Note that decomposing translation $f$ into horisontal and vertical components $f.x$ and $f.y$, respectively, captures two musically distinct phenomena: $f.x$ corresponds to aligning the pattern time-wise, $f.y$ to *transposing* the musical excerpt to a lower or higher key (see Fig. 2). Note also that a musical time-scaling $\sigma$, $\sigma \in \mathbb{R}^+$, has an effect only on the horisontal translation, the vertical translation stays intact.

*Example 1.* Let $p = (1, 1)$, $f = (2, 2)$ and $\sigma = 3$. Then $p + \sigma f = (7, 3)$.

A straight-forward algorithm solves P1 and P2 in $O(mn \log(mn))$ time. The algorithm first collects exhaustively all the translations mapping a point in $P$ to

another point in $T$. The set of the collected translation vectors are then sorted in lexicographic order. In the case of P1, a translation $f$ that has been used $m$ times corresponds to an occurrence; in the case of P2, any translation $f$ that has been used at least $\alpha$ times would account for an occurrence. Thoughtful implementations of the involved scanning (sorting) of the translation vectors, will yield an $O(mn)$ ($O(mn \log m)$) time algorithm for P1 (P2) [1].

Indeed, the above $O(mn \log m)$ time algorithm is the fastest online algorithm known for P2. Moreover, any significant improvement in the asymptotic running time, exceeding the removal of the logarithmic factor, cannot be seen to exist, for P2 is known to be a `3SUM`-hard problem [8]. It is still possible that P2 is also a `Sorting X+Y` -hard problem, in which case Ukkonen et al's PII algorithm would already be an optimal solution. In [8], Clifford et al introduced an $O(n \log n)$ time approximation algorithm for P2.

To make the queries more efficient, several indexing schemes have been suggested. The first indexing method using geometric music representation was suggested by Clausen et al. [3]. Their sublinear query times were achieved by using inverted files, adopted from textual information retrieval. The performance was achieved with a lossy feature extraction process, which makes the approach non-applicable to problems P1 and P2. Typke et al. [4] proposed the use of metric indexes that works under robust geometric similarity measures. The approach lacks flexibility on features pertinent to the application: it is very difficult to adopt it to support translations and partial matching at the same time. Lemström et al's approach [7] combines sparse indexing and (practically) lossless filtering. Their index is used to speed up a filtering phase that charts all the promising areas in the database where real occurrences could reside. Once a query has been received, the filtering phase works in time $O(g_f(m) \log n + n)$ where function $g_f(m)$ is specific to the applied filter $f$. The last phase involves checking the found $c_f$ ($c_f \leq n$) candidate positions using Ukkonen et al's PI or PII algorithm executable in worst-case time $O(c_f m)$ or $O(c_f m \log m)$, respectively.

A brute-force solution for S2 would work in time $O(m^3 n^3)$ and space $O(mn)$: First all translation vectors are calculated, exhaustively, in lexicographic order. This gives $m$ increasing sequences of vectors (pairs of real values) each of length $n$. Then, each possible time-scaling value is selected by choosing two vectors from two distinct sequences; there are $O(m^2 n^2)$ possibilities in this choice. For each time-scaling value, the maximum co-occurrence between pattern and database needs to be determined. This can be done by checking whether each of the remaining $m-2$ sequences (each containing $n$ vectors) includes a vector that accords with the chosen scaling vector. This is feasible in time $O(mn)$. Candidates thus found are to be verified by checking that the pitch intervals also match. The only non-brute-force method for S2 is by Romming and Selfridge-Field [9]. It is based on geometric hashing and works in $O(n^3)$ space and $O(n^2 m^3)$ time. By applying a window on the database such that $w$ is the maximum number

of events that occur in any window, the above complexities can be restated as $O(w^2n)$ and $O(wnm^3)$, respectively.

## 3 Matching Algorithms

Our matching algorithms for the time-scale invariant problems S1 and S2 resemble somewhat Ukkonen et al's original PI and PII algorithms in that they all use a priority queue as the focal structure. Ukkonen et al's PI and PII work on trans-set translations, or *trans-set vectors*, $f = t - p$ (see Fig. 2), where $p$ and $t$ are points in a given query pattern, of length $m$, and in the underlying database, of length $n$, respectively. Let us assume (without loss of generality) that all the points, both in the pattern and in the database, are unique. The number of trans-set vectors is within the range $[n + m - 1, nm]$. In order to be able to build an index on the database in an offline phase, Lemström et al's method [7] is based on *intra-set vectors*. For instance, translation vector $f$ is an *intra-pattern vector*, if there are two points $p$ and $p'$ $(p, p' \in P)$ such that $p + f = p'$. *Intra-database vectors* are defined accordingly. Naturally, the number of intra-pattern and intra-database vectors are $O(m^2)$ and $O(n^2)$, respectively. Lemström et al study several heuristics, relevant to the application, to keep the index structure of a linear $O(n)$ size.

The set of *positive intra-pattern vectors* include translations $p_{i'} - p_i$ where in the case of S1: $0 \leq i < m$ and $i' = i + 1$, and in the case of S2: $0 \leq i < i' \leq m$. The set of *positive intra-database vectors* include translations $t_{k'} - t_k$ where, independently of the case, $0 \leq k < k' \leq n$. For the convenience of the algorithms, we pretend that there are an extra element $p_m$ in the pattern and another extra element $t_n$ in the database. The matching algorithms take as input intra-set vectors, stored in tables $K[i], 0 \leq i < m$. Table $K[i]$ stores intra-database translations that may match [1] the positive intra-pattern vectors $p_{i'} - p_i$, i.e., translation vectors starting at point $p_i$. See Fig. 3 for an illustration on tables $K[i]$.

The entries in our main data structures will be sorted in a lexicographic order. We will specify the underlying order by an ordered set $\aleph$. $\aleph$ is formed by members of $\{a, b, s\}$, where $a, b$ and $s$ correspond to the accordingly named columns in tables $K[i]$. For instance, lexicographic order $\langle a, s \rangle$ is firstly based on the values on column $a$ (the starting point of the associated intra-database vector), secondly on the values on column $s$ (the associated scaling value). A main loop that goes exhaustively through all the possibilities of positive intra-pattern and positive intra-database vectors to initialise the tables $K[i]$ is needed. To this end, let a positive intra-database vector $g = t_{k'} - t_k$ be such that there is a positive intra-pattern vector $f = p_{i'} - p_i$ for which $g.y = f.y$ (ie. the pitch intervals of the two vectors match). Because $g$ may be part of an occurrence,

[1] Please note the distinction between an occurrence and a match. An *occurrence* involves as many *matching* pairs of intra-database and intra-pattern vectors as is required by the problem specification.

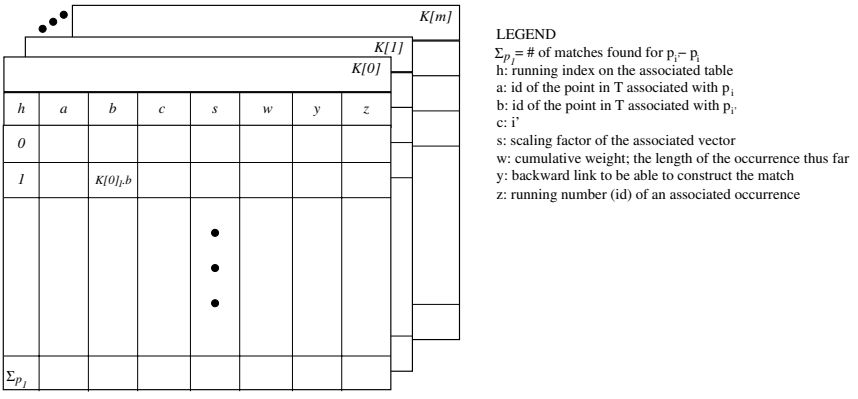



**Fig. 3.** Illustration of the main data structure. Each $K[i]$ stores intra-database vectors $t_{k'} - t_k$, $0 \leq k < k' \leq m-1$ that matches with an intra-patter vector $p_{i'} - p_i$ (where, in the case of S1: $0 \leq i < m-1$ and $i' = i+1$, and in the case of S2 $0 \leq i < i' \leq m-1$) with any positive time-scaling $\sigma \in \mathbb{R}^+$.

a new row, let it be the $h$th, in $K[i]$ is allocated and the following updates are conducted:

$$K[i]_h.a \leftarrow k; \quad K[i]_h.b \leftarrow k'; \tag{1}$$

$$K[i]_h.s \leftarrow \frac{t_{k'}.x - t_k.x}{p_{i'}.x - p_i.x}; \tag{2}$$

$$K[i]_h.y \leftarrow \mathbf{nil}; \quad K[i]_h.w \leftarrow 1; \tag{3}$$

$$K[i]_h.c \leftarrow i'; \quad K[i]_h.z \leftarrow 0. \tag{4}$$

Above, in (1), the associated starting and ending points of the matching intra-database vector are stored in $K[i]_j.a$ and $K[i]_j.b$, respectively. The required time scaling for the intra-vectors to match is stored in $K[i]_j.s$ (2); here extra carefulness is needed in order to avoid dividing by zero: If both the numerator and the denominator equal zero, we set $K[i]_h.s = 1$. If only one of them equals zero, the whole row is deleted from the table altogether (it would represent for an impossible time scaling). The columns $y$ and $w$, initialised in (3), are used for backtracking a found occurrence and storing the length of a candidate occurrence, respectively. The last columns, initialised in (4), will be needed when searching for partial occurrences (in Section 3.2): column $c$ stores the ending point of the associated intra-pattern vector, $z$ is used for identifying an occurrence.

Denoting by $\Sigma_{p_i}$ the number of rows generated above for table $K[i]$, $0 \leq i < m$, for the aforementioned extra elements (for the convenience of the algorithms) we set:

$$K[i]_{\Sigma_{p_i}}.a \leftarrow K[i]_{\Sigma_{p_i}}.b \leftarrow \infty;\ K[i]_{\Sigma_{p_i}}.s \leftarrow K[i]_{\Sigma_{p_i}}.w \leftarrow 0;\ K[i]_{\Sigma_{p_i}}.c \leftarrow i+1$$

As each iteration of the main loop takes a constant time, this exhaustive initialisation process runs in $O(n^2m^2)$ time. Finally, the columns in $K[i]$ are

sorted in lexicographic order $\langle a, s\rangle$. The matching algorithms have an associated priority queue $Q_i$ for each table $K[i]$, $0 < i \leq m$ [2]. For $Q_i$, a lexicographic order $\langle b, s\rangle$ is used. As a reminder, the order is given in the superscript of a priority queue (e.g. $Q_i^{\langle b,s\rangle}$).

### 3.1 S1: Quest for Time Scaled Exact Occurrences

Once the tables $K[i]$ have been initialised and their columns have been sorted in lexicographic order $\langle a, s\rangle$, the transposition-invariant time-scaled exact occurrences can be found using the matching algorithm given in Fig. 4. The algorithm works by observing *piecewise matches* between positive intra-database and intra-pattern vectors

$$t_{k_{i'}} - t_{k_i} = \sigma_i(p_{i+1} - p_i) \tag{5}$$

that are stored in the associated $K[i]$. Above $\sigma_i \in \mathbb{R}^+$ is the time-scaling factor (recall Example 1). The piecewise matches may form a chain $T_{\tau_0 \ldots \tau_{m-1}} = t_{\tau_0}, t_{\tau_1}, \ldots, t_{\tau_{m-1}}$, where $\tau_0, \tau_1, \ldots, \tau_{m-1}$ is an increasing sequence of indices in $T$; $t_{\tau_{i+1}} - t_{\tau_i} = \sigma(p_{i+1} - p_i)$ for $0 \leq i < m-1$ and $\sigma \in \mathbb{R}^+$ is a time-scaling factor common to all the piecewise matches in the chain. Naturally, such a chain would constitute a transposition-invariant, time-scaled exact occurrence. A chain $T_{\tau_0 \ldots \tau_{m'}}$, $m' < m-1$, is called a *prefix occurrence* (of length $m'$); $T_{\tau_{m'-1}, \tau_{m'}}$ is the *final suffix* of the prefix occurrence $T_{\tau_0 \ldots \tau_{m'}}$.

Let $t_{\tau_{i+1}} - t_{\tau_i}$ (that, by definition, equals $\sigma(p_{i+1} - p_i)$) be the final suffix of a prefix occurrence $T_{\tau_1 \ldots \tau_{m'}}$. The prefix occurrence is *extensible* if there is a piecewise match $t_{k'_{i+1}} - t_{k_{i+1}} = \sigma(p_{i+2} - p_{i+1})$ such that

$$t_{\tau_{i+1}} = t_{k_{i+1}} \tag{6}$$

and scaling factor $\sigma$ is the one that was used in forming $T_{\tau_1 \ldots \tau_{m'}}$. The binding in Equation 6 is called the *binding of extension*, $t_{\tau_{i+1}} - t_{\tau_i}$ the *antecedent* and $t_{k'_{i+1}} - t_{k_{i+1}}$ the *postcedent of the binding*.

**Lemma 1.** *If a prefix occurrence is extensible, its final suffix is also extensible.*

*Proof.* Immediate. □

To be more efficient, at point $i+1$, the algorithm actually considers any piecewise match $t_{k'_i} - t_{k_i} = \sigma_i(p_{i+1} - p_i)$ as an antecedent to the binding and tries to extend it. Because in this case the piecewise matches in an occurrence chain have to be consecutive in $P$, the antecedents of the binding are all found in $K[i]$ and their possible extensions, postcedents, in $K[i+1]$. To process all the bindings of extension at point $i+1$, therefore, involves going through all the entries both in $K[i]$ and in $K[i+1]$. To make this process efficient, no entry of either of the tables should be observed more than once for one iteration. In order for this to be possible, both sides of the binding of extension (associated with antecedents

[2] A single priority queue would suffice, but the algorithm would become more complicated.

TimeScaledExactOccurrence($K[i]$)
(1) **for** $j \leftarrow 0, \ldots, \Sigma_{p_0}$ **do**
(2) $\quad Q_1^{\langle b,s\rangle} \leftarrow \mathbf{push}(\&K[0]_j)$
(3) **for** $i \leftarrow 1, \ldots, m-1$ **do**
(4) $\quad q \leftarrow \mathbf{pop}(Q_i^{\langle b,s\rangle})$
(5) $\quad$ **for** $j \leftarrow 0, \ldots, \Sigma_{p_i-1}$ **do**
(6) $\quad\quad$ **while** $[q.b, q.s] < [K[i]_j.a, K[i]_j.s]$ **do**
(7) $\quad\quad\quad q \leftarrow \mathbf{pop}(Q_i^{\langle b,s\rangle})$
(8) $\quad\quad$ **if** $[q.b, q.s] = [K[i]_j.a, K[i]_j.s]$ **then**
(9) $\quad\quad\quad K[i]_j.w \leftarrow q.w + 1$
(10) $\quad\quad\quad K[i]_j.y \leftarrow q$
(11) $\quad\quad\quad Q_{i+1}^{\langle b,s\rangle} \leftarrow \mathbf{push}(\&K[i]_j)$
(12) $\quad\quad\quad q \leftarrow \mathbf{pop}(Q_i^{\langle b,s\rangle})$
(13) $\quad K[i]_{\Sigma_{p_i}}.s \leftarrow \infty$
(14) $\quad Q_{i+1}^{\langle b,s\rangle} \leftarrow \mathbf{push}(\&K[i]_{\Sigma_{p_i}})$
(15) **if** $K[m-1]_j.w = m$ **for some** $0 \le j \le \Sigma_{p_{m-1}}$ **then** `report an occurrence`

**Fig. 4.** Online algorithm for finding transposition-invariant time-scaled exact occurrences

and postcedents) should be enumerated in the same (increasing) order. However, the lefthand side of the binding involves end points of the intra-database vectors in $K[i]$ and the righthand side the start points of the intra-database vectors in $K[i+1]$. Therefore, we use a priority queue $Q_{i+1}^{\langle b,s\rangle}$ whose entries are addresses to rows associated with the antecedents of the binding at $i+1$. In this way, the binding of extension at $i+1$ can be done efficiently by enumerating the entries in $Q_{i+1}$ and $K[i+1]$. Note that the set of piecewise matches extended this way also includes all the final suffixes, and therefore, according to Lemma 1, also all the prefix occurrences.

The binding of extension takes place in line (8) of the algorithm. If a piecewise match is extensible, its length is updated (line 9) and a backtracking link is stored (line 10). The latter becomes useful if any of the extended piecewise matches extends into a proper occurrence, and the whole occurrence is to be revealed (instead of just reporting it).

*Correctness.* Let there be an occurrence $t_{\tau_0}, t_{\tau_1}, \ldots, t_{\tau_{m-1}}$, such that $t_{\tau_{i+1}} - t_{\tau_i} = \sigma(p_{i+1} - p_i)$ for $0 \le i < m-1$ with some $\sigma \in \mathbb{R}^+$. It is obvious from the way the tables are constructed that every element $t_{\tau_{i+1}} - t_{\tau_i}$, associated with a piecewise match, is stored in the corresponding table $K[i]$. Clearly the first antecedent $t_{\tau_1} - t_{\tau_0}$ is treated correctly: It is inserted in the priority queue $Q_1$ in line (2); being part of a proper occurrence, the binding of extension can be done with a correct time-scaling factor and, therefore, the equation condition in line (8) is satisfied. The length of this prefix occurrence is set to 2 (line (9)), and the address of this newly found final suffix is stored in $Q_{i+1}$ (line (11)) to be considered as an antecedent for a binding of extension at iteration 2. Let us now assume that

everything is done correctly up to point $i$, $2 < i < m-1$ and we are dealing with the element $t_{\tau_{i+1}} - t_{\tau_i}$. It is pushed in the priority queue $Q_{i+1}$ in line (12) just before the iteration $i+1$ of the loop in line (3) starts. Then again, being part of a proper occurrence, the antecedent $t_{\tau_{i+1}} - t_{\tau_i}$, found in $Q_{i+1}$, and the postcedent $t_{\tau_{i+2}} - t_{\tau_{i+1}}$, found in $K[i+1]$, can be bound: the extension condition is met and the prefix occurrence is increased to $i+2$, and the address of the new final suffix is passed to further iterations. So, by induction, the algorithm finds the existing occurrences.

That the algorithm does not wrongly increase the length of a piecewise match (causing reports of spurious occurrences or occurrences of lengths exceeding $m$) is down to the binding of extension, i.e., the condition in line (8). Remember that we assume points in $T$ and in $P$ to be unique. Let us make a counter assumption that a piecewise match may be unintentionally extended. To that end, at some iteration $j$ of the inner loop (starting at line (5)) there has to be at least two antecedents in $Q_i$ that can be bound with a postcedent stored in $K[i]_j$ (line (8)). Let $q$ and $q'$ be entries in $Q_i$ that are antecedents for a postcedent in $K[i]_j$, where $K[i]_j.a = t_l$, and let $q$ correspond to a piecewise match $t_l - t_k = \sigma(p_i - p_{i-1})$ and $q'$ to $t_l - t_{k'} = \sigma(p_i - p_{i-1})$. But this is impossible unless $t_k = t_{k'}$ which contradicts our assumption of points being unique in $T$. □

*Analysis.* Let us denote by $|Q_i|$ and $|K[j]|$ the number of entries in $Q_i$ and $K[j]$, respectively. Clearly, in this case, $|Q_i| \leq |K[i-1]|$ for $1 \leq i \leq m$. Moreover, let $\Sigma = max_{i=1}^{m}(|Q_i|, |K[i-1]|)$. The outer loop (line (3)) is iterated $m$ times. Within the inner loop (line (5)), all the entries in $Q_i$ and in $K[i]$ are processed exactly once, resulting in $O(\Sigma)$ entry processing steps. The only operation taking more than a constant time is the updating of the priority queue; it takes at most $O(\log \Sigma)$ time. Thus, the algorithm runs in time $O(m\Sigma \log \Sigma)$. Moreover, the tables $K[i]$ and priority queues $Q_i$ require $O(m\Sigma)$ space.

In this case $\Sigma = O(n^2)$, because each table $K[i]$ contains the piecewise matches for the positive intra-pattern vector $p_{i+1} - p_i$, and there are $O(n^2)$ possibilities to this end. □

### 3.2 S2: Quest for Time Scaled Partial Occurrences

In order to be able to find transposition-invariant time-scaled partial occurrences, we need the two extra columns $c$ and $z$, that were initialised in Equation 4, for tables $K[i]$. Recall that $K[i]_h.c$ stores the ending point $i'$ for an intra-pattern vector $p_{i'} - p_i$ that is found to match an intra-database vector $t_{k'} - t_k$ with some scaling factor $\sigma_i$. Column $z$ is used for storing a running number that is used as an id, for a found partial occurrence. Furthermore, we use an extra table, denoted by $\kappa$, for storing all the found occurrences.

The structure of the algorithm, given in Fig. 5, is similar to the previous algorithm. Again, at point $i$, the antecedents in $Q_i$ are to be extended by postcedents found in $K[i]$. However, as we are looking for partial occurrences this time, we

```
TimeScaledPartialOccurrence(K[i])
(0)  ℓ ← 0
(1)  for j ← 0, ..., Σ_{p_0}
(2)       Q^{⟨b,s⟩}_{K[0]_j.c} ← push(&K[0]_j)
(3)  for i ← 1, ..., m − 1 do
(4)       q ← pop(Q_i^{⟨b,s⟩})
(5)       for j ← 0, ..., Σ_{p_i − 1} do
(6)            while [q.b, q.s] < [K[i]_j.a, K[i]_j.s] do
(7)                 q ← pop(Q_i^{⟨b,s⟩})
(8)            if [q.b, q.s] = [K[i]_j.a, K[i]_j.s] then
(9)                 while min(Q_i^{⟨b,s⟩}) = [q.b, q.s] do
(10)                     r ← pop(Q_i^{⟨b,s⟩})
(11)                     if r.w > q.w then q ← r
(12)                K[i]_j.w ← q.w + 1
(13)                K[i]_j.y ← q
(14)                if K[i]_j.w = α then
(15)                     ℓ ← ℓ + 1
(16)                     K[i]_j.z = ℓ
(17)                     κ[ℓ] ← &K[i]_j
(18)                if K[i]_j.w > α then
(19)                     K[i]_j.z = q.z
(20)                     κ[q.z] ← &K[i]_j
(21)                Q^{⟨b,s⟩}_{K[i]_j.c} ← push(&K[i]_j)
(22)                q ← pop(Q_i^{⟨b,s⟩})
(23)       K[i]_{Σ_{p_i}}.s ← ∞
(24)       Q_{i+1}^{⟨b,s⟩} ← push(&K[i]_{Σ_{p_i}})
(25) ReportOccurrences(κ)
```

**Fig. 5.** Online algorithm for finding transposition-invariant time-scaled partial occurrences

cannot rely on piecewise matches that are consecutive in $P$ but any piecewise match associated with a positive intra-pattern vector

$$t_{k_{i'}} - t_{k_i} = \sigma_i(p_{i'} - p_i) \tag{7}$$

has to be considered. Here $0 \leq k_i < k_{i'} \leq n-1$; $0 \leq i < i' \leq m-1$ and $\sigma_i \in \mathbb{R}^+$. Given a threshold $\alpha$, a chain $T_{\tau_0 \ldots \tau_{\beta-1}}$, such that $t_{\tau_j} - t_{\tau_{j-1}} = \sigma(p_{\pi_j} - p_{\pi_{j-1}})$, for $0 < j \leq \beta$, $\beta \geq \alpha$, where $\tau_0 \ldots \tau_{\beta-1}$ and $\pi_0 \ldots \pi_{\beta-1}$ are increasing sequences of indices in $T$ and $P$, respectively, would constitute for a transposition-invariant time-scaled partial occurrence (of length $\beta$).

That piecewise matches can now be between any two points in the pattern makes the problem somewhat more challenging. This has the effect that, at point $i$, pushing a reference to a priority queue (lines (2) and (21) of the algorithm) may involve any future priority queue, from $Q_{i+1}$ to $Q_m$, not just the successive one as in the previous case; the correct priority queue is the one that is stored in

$K[i]_j.c$ (recall that it stores the end point of the intra-pattern vector associated with the piecewise match). Conversely, the antecedents at point $i$ (stored in $Q_i$) may include references to any past tables, from $K[0]$ to $K[i-1]$, expanding the size of the priority queue $Q_i$.

The two remaining differences to the algorithm above, are in lines (11) and (14-20). In line (11), the algorithm chooses to extend the piecewise match that is associated with the longest prefix occurrence. This is a necessary step, once again, because we are no more dealing with piecewise matches that are consecutive in $P$. In lines (14-20) the algorithm deals with a found occurrence. Lines (14-17) deal with a new occurrence: generate a new running number, $\ell$, for it (that is used as its id) and store a link to the found occurrence to the table of occurrences $\kappa$. Lines (18-20) deal with extending a previously found occurrence.

*Correctness and Analysis.* Denoting by $\Sigma = max_{i=1}^{m}(|Q_i|, |K[i-1]|)$, with an analogous reasoning to that of the previous analysis, we arrive at similar complexities: the algorithm runs in $O(m\Sigma \log \Sigma)$ time and $O(m\Sigma)$ space. Let us now analyse the order of $\Sigma$ in this case. Still it holds that for a positive intra-pattern vector, $p_{i'} - p_i$, there are $O(n^2)$ possible piecewise matches. However, the table $K[i]$ may contain entries associated with piecewise matches with any positive intra-pattern vector ending at point $i'$. Thus, $max_{i=1}^{m}(|K[i-1]|) = O(mn^2)$. As $|Q_i| \leq |K[i-1]|$ for $0 < i \leq m$ and $m = O(n)$, we conclude that the algorithm has an $O(m^2n^2 \log n)$ running time and works in a space $O(m^2n^2)$.

The proof of the correctness of this algorithm is left for an interested reader. □

## 4 Conclusions

In this paper we suggested novel content-based music retrieval algorithms for polyphonic, geometrically represented music. The algorithms are both transposition and time-scale invariant. Given a (polyphonic) query pattern $P = p_0, \ldots, p_{m-1}$ to be searched for in a polyphonic music database $T = t_0, \ldots, t_{n-1}$, the algorithms run in $O(m\Sigma \log \Sigma)$ time and $O(m\Sigma)$ space, where $\Sigma = O(n^2)$ when searching for exact occurrences under such a setting, and $\Sigma = O(n^2m)$ when searching for partial occurrences. Whether this is an improvement in practice over the existing algorithm by Romming and Selfridge-Field [9], working in space $O(n^3)$ and time $O(n^2m^3)$, is left for future experiments on real data.

However, the new approach seems to be very flexible: at the present the author is adopting the approach to a more complex case, where an uneven time deformation is known just to preserve the order of the notes; there are no known solutions for this time-warping invariant problem [6]. Moreover, it seems that with slight modifications to the data structures and ideas presented by Lemström, Mikkilä and Mäkinen in [7], it would be possible to adopt the idea of using a three-phase searching process (indexing, filtering and checking) with a smaller search space and a better running time to those presented here.

## Acknowledgement

The work was supported by the Academy of Finland, grants #108547 and #218156.

## References

1. Ukkonen, E., Lemström, K., Mäkinen, V.: Sweepline the music! In: Klein, R., Six, H.-W., Wegner, L. (eds.) Computer Science in Perspective. LNCS, vol. 2598, pp. 330–342. Springer, Heidelberg (2003)
2. Ghias, A., Logan, J., Chamberlin, D., Smith, B.: Query by humming - musical information retrieval in an audio database. In: ACM Multimedia 1995 Proceedings, San Francisco, CA, pp. 231–236 (1995)
3. Clausen, M., Engelbrecht, R., Meyer, D., Schmitz, J.: Proms: A web-based tool for searching in polyphonic music. In: Proceedings of the International Symposium on Music Information Retrieval (ISMIR 2000), Plymouth, MA (October 2000)
4. Typke, R.: Music Retrieval based on Melodic Similarity. PhD thesis, Utrecht University, Netherlands (2007)
5. Wiggins, G., Lemström, K., Meredith, D.: SIA(M)ESE: An algorithm for transposition invariant, polyphonic content-based music retrieval. In: Proceedings of the International Conference on Music Information Retrieval (ISMIR 2002), Paris, France, October 2002, pp. 283–284 (2002)
6. Lemström, K., Wiggins, G.: Formalizing invariances for content-based music retrieval. In: Proceedings of the 10th International Conference on Music Information Retrieval (ISMIR 2009), Kobe, October 2009, pp. 591–596 (2009)
7. Lemström, K., Mikkilä, N., Mäkinen, V.: Filtering methods for content-based retrieval on indexed symbolic music databases. Journal of Information Retrieval 13(1), pp. 1–21 (2010)
8. Clifford, R., Christodoulakis, M., Crawford, T., Meredith, D., Wiggins, G.: A fast, randomised, maximal subset matching algorithm for document-level music retrieval. In: Proceedings of the 7th International Conference on Music Information Retrieval, Victoria, BC, Canada, October 2006, pp. 150–155 (2006)
9. Romming, C., Selfridge-Field, E.: Algorithms for polyphonic music retrieval: The hausdorff metric and geometric hashing. In: Proceedings of the 8th International Conference on Music Information Retrieval (ISMIR 2007), Vienna, Austria, September 2007, pp. 457–462 (2007)

# Unified View of Backward Backtracking in Short Read Mapping

Veli Mäkinen*, Niko Välimäki**, Antti Laaksonen***, and Riku Katainen

Department of Computer Science, University of Helsinki, Finland
{vmakinen,nvalimak,ahslaaks,rkataine}@cs.helsinki.fi

**Abstract.** Mapping short DNA reads to the reference genome is the core task in the recent high-throughput technologies to study e.g. protein-DNA interactions (ChIP-seq) and alternative splicing (RNA-seq). Several tools for the task (bowtie, bwa, SOAP2, TopHat) have been developed that exploit Burrows-Wheeler transform and the backward backtracking technique on it, to map the reads to their best approximate occurrences in the genome. These tools use different tailored mechanisms for small error-levels to prune the search phase significantly. We propose a new pruning mechanism that can be seen a generalization of the tailored mechanisms used so far. It uses a novel idea of storing all cyclic rotations of fixed length substrings of the reference sequence with a compressed index that is able to exploit the repetitions created to level out the growth of the input set. For RNA-seq we propose a new method that combines dynamic programming with backtracking to map efficiently and correctly all reads that span two exons. Same mechanism can also be used for mapping mate-pair reads.

**Keywords:** Personal genomics, full-text indexing, compressed data structures.

## 1 Introduction

High-throughput *short read sequencing* is revolutionizing the way molecular biology is researched. For example, the routine task of measuring gene expression by microarrays is now being replaced by a technology called *RNA-seq* [21,26]; the transcriptome is shotgun sequenced so that one is left with a set of short reads (typically e.g. of length 36 basepairs) whose sequence is known but it is not known from which parts of the genome they were transcribed. The process is hence reversed by mapping the short reads back to the genome (assuming that the reference genome sequence is known). After the mapping is produced (i.e. after matching each short read sequence to its best exact/approximate occurrence in the genome) one should see clear clusters of occurrences indicating the

* Funded by the Academy of Finland under grant 119815.
** Funded by the Helsinki Graduate School in Computer Science and Engineering.
*** Funded by the Finnish Centre of Excellence for Algorithmic Data Analysis Research.

T. Elomaa et al. (Eds.): Ukkonen Festschrift, LNCS 6060, pp. 182–195, 2010.

expression of the genes residing in the clustered areas. Also alternative splicing can be studied when analyzing the result of the mapping.

Analogous short read mapping tasks are part of protein-DNA interactions studies by *ChIP-seq* [8,9] and population studies by *targeted resequencing* [7].

Measurement errors, genomic variation between individuals, and exon/intron boundaries (in RNA-seq) make these mapping tasks non-trivial. Yet, the mapping task can be formulated as a *multiple approximate string matching* problem [23] by using a proper distance/similarity measure capturing the different error types. Luckily the short reads have very good quality, so not too many measurement errors need to be allowed. Also typical genomic variations are single-nucleotide polymorphisms (SNPs), making a simple *k-mismatches search* [23] a sufficient choice on most reads. For finding more rare *indels* (or allowing such measurement errors) one can apply *k-errors search* [23] instead.

A practical bottleneck in short read mapping is that one experiment produces millions of reads and the mapping should hence be done as fast as possible. Since the genome is static, it makes sense to preprocess it into an *index structure* to speed up queries againts the reads. *Suffix trees* [27,20,25] and *suffix arrays* [19] are typical examples of such *full-text* index structures, and are the basis of the best theoretical results [2] on this indexed approximate matching problem.

Classical full-text indexes take however too much space on genome-scale sequences, so the widely used tools for short read mapping like `MaQ` [13], `SOAP` [15], and `ELAND` (the mapper in *Illumina Solexa* sequencing pipeline) use instead substring hashing techniques as a basis. Recently, many new software packages have come out building on the *Burrows-Wheeler transform (BWT)* [1] and on the *FM-index* [4] concept. The FM-index provides a way to index a sequence within space of compressed sequence exploiting BWT. This index provides so-called *backward search* principle that enables very fast exact string matching on the indexed sequence. Lam et al. [11] extended backward search to simulate backtracking on suffix tree, i.e., to simulate dynamic programming on all relevant paths of suffix tree; their tool `BWT-SW` provides an efficient way to do local alignment without the heuristics used in many common bioinformatics tools. The same idea of bacward backtracking is exploited in the tools tailored for short read mapping: `bowtie` [12], `bwa` [14], `SOAP2` [16].

In this paper, we study the different pruning heuristics used in the BWT-based short read mapping tools. We compare these heuristics againts the best *filtering* algorithms proposed for approximate string matching: A filtering algorithm tries to find fast some candidate positions that are then checked for real occurrences. Many such index-based filters have been proposed in the literature, and the current state-of-the-art is the *suffix filter* [10]. Our experiments indicate a perhaps surprising result that one of the backtracking pruning heuristics – the *case analysis pruning* used in bowtie – is in fact slightly superior to suffix filter.

We then propose a hybrid pruning heuristic – *rotation pruning* – that combines suffix filter and case analysis pruning. Our experiments show that this new pruning method is clearly superior in speed to any of the competitors.

The downsides are that rotation pruning needs a separate index for each pattern length and that each index takes significantly more space (typically 40-times) than the other pruning / filtering mechanisms.

The new index supporting rotation pruning uses a novel idea that may be of independent interest likely to have many other uses outside the current application: We first blow up the data by generating all cyclic rotations of all substrings of the length of the pattern and then use a compressed index [18] that is able to exploit the repetitions created. This has the effect that the index actually represent a virtually large data, but its space requirement remains reasonable. We show an expected case bound of $O(n \log n \log m)$ bits for the index, where $n$ is the length of the sequence indexed and $m$ is the pattern length.

The methods above are limited to simple DNA reads. We also study RNA-seq mapping and mapping with mate-pair reads. The former has the extra difficulty of a possible intron splitting the read and the latter the additional advantage that the reads come by pairs with the approximate in-between distance known. For RNA-seq mapping the typical mechanism used e.g. in `TopHat` [24] is to map first all the reads that map without allowing introns to split, then analyze the occurrence clusters to detect exon boundaries, merge prefixes and suffixes of nearby exons, and match the substrings in merged prefix-suffix pairs to rest of the reads. The way to handle mate-pairs in all the current tools is to map each end separately and pair those occurrences that are within the allowed limits.

We propose a direct way to map RNA-seq and mate-pair reads: We couple the dynamic programming with Cartesian trees [6] to be able to allow one long gap (intron or mate-pair distance) without slowing down the dynamic programming computation. Our experiments show that we are able to obtain the same speed and accuracy as TopHat. Our method is also robust to indels, unlike TopHat.

### 1.1 Background

A *string* $S = S_{1,n} = s_1 s_2 \cdots s_n$ is a *sequence* of *symbols* (a.k.a. characters or letters). Each symbol is an element of an *alphabet* $\Sigma = \{1, 2, \ldots, \sigma\}$. A *substring* of $S$ is written $S_{i,j} = s_i s_{i+1} \ldots s_j$. A *prefix* of $S$ is a substring of the form $S_{1,j}$, and a *suffix* is a substring of the form $S_{i,n}$. If $i > j$ then $S_{i,j} = \varepsilon$, the empty string of length $|\varepsilon| = 0$. A *text* string $T = T_{1,n}$ is a string terminated by the special symbol $t_n = \$ \notin \Sigma$, smaller than any other symbol in $\Sigma$. The *lexicographical order* "$<$" among strings is defined in the obvious way.

**Burrows-Wheeler Transform.** The methods to be studied are derivatives of the *Burrows-Wheeler transform (BWT)* [1]. The transform produces a permutation of $T$, denoted by $T^{bwt}$, as follows: (i) Build the *suffix array* [19] $\mathsf{SA}[1, n]$ of $T$, that is an array of pointers to all the suffixes of $T$ in the lexicographic order; (ii) The transformed text is $T^{bwt} = L$, where $L[i] = T[\mathsf{SA}[i] - 1]$, taking $T[0] = T[n]$. The BWT is reversible, that is, given $T^{bwt} = L$ we can obtain

$T$ as follows [1]: (a) Compute the array $C[1,\sigma]$ storing in $C[c]$ the number of occurrences of characters $\{\$, 1, \ldots, c-1\}$ in the text $T$; (b) Define the *LF mapping* as follows: $LF(i) = C[L[i]] + rank_{L[i]}(L, i)$, where $rank_c(L, i)$ is the number of occurrences of character $c$ in the prefix $L[1, i]$; (c) Reconstruct $T$ backwards as follows: set $s = 1$, for each $i \leftarrow n-1, \ldots, 1$ do $t_i \leftarrow L[s]$ and $s \leftarrow LF[s]$. Finally put the end marker $t_n \leftarrow \$$.

**Backward Search.** The *FM-index* [4] is a self-index based on the BWT. It is able to locate the interval $\mathsf{SA}[sp, ep]$ that contains the occurrences of $P$ without having $\mathsf{SA}$ stored. The FM-index uses the array $C$ and function $rank_c(L, i)$ in the so-called *backward search* algorithm, calling function $rank_c(L, i)$ $O(|P|)$ times. Its pseudocode is given below.

**Algorithm.** Count($P[1 \ldots m]$,$L[1 \ldots n]$)
(1) $i \leftarrow m$;
(2) $sp \leftarrow 1$; $ep \leftarrow n$;
(3) **while** $(sp \leq ep)$ **and** $(i \geq 1)$ **do**
(4) $\quad s \leftarrow P[i]$;
(5) $\quad sp \leftarrow C[s] + rank_s(L, sp-1)+1$;
(6) $\quad ep \leftarrow C[s] + rank_s(L, ep)$;
(7) $\quad i \leftarrow i - 1$;
(8) **if** $(ep < sp)$ **then return** "not found"
**else return** "found $(ep - sp + 1)$ occurrences".

The correctness of the above algorithm is easy to see by induction: At each phase $i$, $[sp, ep]$ gives the maximal interval of suffix array $\mathsf{SA}$ pointing to suffixes prefixed by $P[i \ldots m]$.

To report the occurrence positions $\mathsf{SA}[i]$ for $sp \leq i \leq ep$, a common approach is to sample $\mathsf{SA}$ values and then use the $LF$-mapping to derive the unsampled values from the sampled ones. Many variants of the FM-index have been derived that differ mainly in the way the $rank_c(L, i)$-queries are solved [22]. For example, on small alphabets, it is possible to achieve $nH_k + o(n \log \sigma)$ bits of space, for moderate $k$, with constant time support for $rank_c(L, i)$ [5]. Here $H_k$ is the standard *$k$-th order entropy*, i.e., the minimum number of bits to code a symbol once its $k$-symbol context is seen. There holds $H_k \leq \log \sigma$.

**Backward Backtracking.** The backward search can be extended to *backtracking* to allow the search for approximate occurrences of the pattern [11]. To get an idea of this general approach, let us first concentrate on the *$k$-mismatches* problem, where pattern $P[1 \ldots m]$ approximately matches a substring $X[1 \ldots m]$ of the text $T$, if there are at most $k$ indices $i$ such that $P[i] \neq X[i]$. The following pseudocode finds the $k$-mismatch occurrences, and is analogous to the schemes used in [13,12,14]. The first call to the recursive procedure is `kmismatches(`$P, L, k, m, 1, n$`)`.

**Algorithm.** `kmismatches`$(P, L, k, j, sp, ep)$
(1) **if** $(sp > ep)$ **return** ;
(2) **if** $(j = 0)$ **then**
(3) Report occurrences $\mathsf{SA}[sp], \ldots, \mathsf{SA}[ep]$; **return** ;
(4) **for each** $s \in \Sigma$ **do**
(5) $sp' \leftarrow C[s] + rank_s(L, sp - 1)+1$;
(6) $ep' \leftarrow C[s] + rank_s(L, ep)$;
(7) **if** $(P[j] \neq s)$ **then** $k' \leftarrow k - 1$; **else** $k' \leftarrow k$;
(8) **if** $(k' \geq 0)$ `kmismatches`$(P, L, k', j - 1, sp', ep')$;

The difference to the exact search is that the recursion considers incrementally from right to left all different ways to alter the pattern with at most $k$-substitutions. Simultaneously, the recursion maintains the suffix array interval $\mathsf{SA}[sp \ldots ep]$ where suffixes match the current modified suffix of the pattern. It is easy to extend the algorithm to find all *k-errors* occurrences [14], where also insertions and deletions can be applied to the pattern.

**Pruning Search Space.** The worst case complexity of backward backtracking is $O(|\Sigma|^k m^{k+1})$, assuming the text is long enough to contain nearly all pattern occurrences with $k$-mismatches. There are some recent practical proposals to prune the search space [12,14]; both share the idea of building the FM-index also for the reverse text $T^r = t_n t_{n-1} \cdots t_1$. Let us call *forward FM-index* and *reverse FM-index* the FM-index of $T$ and FM-index of $T^r$, respectively.

In [14] the reverse FM-index is used for precomputing for each prefix $\alpha$ of the pattern, its splitting to minimum number of pieces such that each piece occurs at least once in the text. Let us denote the minimum number of splits $\kappa(\alpha)$. The computation of $\kappa(\alpha)$ for all prefixes $\alpha$ is analogous to the backward search algorithm (applied to reverse pattern on reverse FM-index), and hence works in linear number of steps in the pattern length [14]. The computation is also possible to do with forward FM-index alone by simulating the suffix array binary search (with roughly logarithmic slowdown to the linear preprocessing).

Value $\kappa(\alpha)$ works as a lower bound for the number of errors that must be allowed in any approximate match for the prefix $\alpha$. This estimate can be used to prune the search space of backward backtracking as follows. Each search state knows the number of errors, say $\eta(\beta, sp, ep)$, between a suffix $\beta$ of the pattern and the longest common prefix of suffixes $\mathsf{SA}[sp \ldots ep]$; if $\kappa(\alpha) + \eta(\beta, sp, ep) > k$, the branch can be ignored. Let us call this strategy *prefix pruning*.

In [12] they extend the standard filtering technique of splitting pattern into $k + 1$ pieces [23]: The original idea of pattern splitting is to be able to search each piece exactly, since $k$-errors/-mismatches cannot affect all pieces simultaneosly. Then the surrounding of each exact occurrence of each piece is checked for possible $k$-errors/-mismatches match. This filter is trivial to implement using exact search in FM-index, but for large error levels too many candidate matches need to be checked.

The extension proposed in [12] is to consider separately all cases how $k$ mismatches can be distributed in the $k + 1$ pieces, and perform backtracking for

the whole pattern for each case either from forward or from reverse FM-index, depending on which one is likely to prune better the search space. Let us call this strategy *case analysis pruning*. Too see how it works, let us consider the simplest case $k = 1$ first. Pattern $P$ is split into two pieces $P = \alpha\beta$. One error can be either (a) in $\alpha$ or (b) in $\beta$. In case (a), it is preferable to search for $P = \alpha\beta$ using backward backtracking on the forward FM-index, since $\beta$ must appear exactly and branching is only needed after reading the $|\beta|$ first symbols. In case (b), it is affordable to search for $P^r = \beta^r\alpha^r$ using backward backtracking on the reverse FM-index, since $\alpha^r$ must appear exactly and branching is only needed after reading the $|\alpha|$ first symbols. For obvious reasons $|\alpha| \approx |\beta|$ is a good choice for pruning efficiency. Let us then consider $k = 2$ to see the limitations of the approach. The different ways to distribute two errors into three pieces are (a) 002, (b) 020, (c) 200, (d) 011, (e) 101, and (f) 110. Obviously in cases (a) and (d) it makes sense to use backtracking on the reverse FM-index and in cases (c) and (f) backtracking on the forward FM-index. For cases (b) and (e) either choice is as good or bad. Obviously for any $k$, there is always the bad case where both ends have at least $k/2$ errors. Hence, there is no strategy to start the backtracking with 0 errors, other than in case $k = 1$.

In the sequel, we study *backward backtracking* in more detail and propose a general mechanism to minimize the backtracking effort.

## 2 New Pruning Mechanism

We build on the *suffix filter* of [10] that is another extension of the pattern splitting strategy. Instead of splitting the pattern into pieces to be searched for exactly, the suffixes starting from the start positions of the pieces are considered. More concretely, let pattern $P$ be partitioned into pieces $P = \alpha_1\alpha_2 \cdots \alpha_{k+1}$, then the set of suffixes considered is $\mathcal{S} = \{\alpha_1\alpha_2 \cdots \alpha_{k+1}, \alpha_2\alpha_3 \cdots \alpha_{k+1}, \ldots, \alpha_{k+1}\}$. Then each $S \in \mathcal{S}$ is searched for from the text so that zero errors are allowed before reaching the end if first piece in $S$, one error is allowed before reaching the end of second piece of $S$, and so on. Obviously this search can be done e.g. using backtracking on FM-index (to be precise, backward backtacking on reverse FM-index in order to backtrack on suffixes).

The reason why suffix filter misses no real occurrences can be seen as follows (see [10] for the original proof; we give a new one to introduce some concepts useful in the sequel). Let us limit to $k$-mismatches case. Assume that instead of the suffix set $\mathcal{S}$ the filter would be searching for all *rotations* of the pattern starting from the start positions of the pieces. Namely, set $\mathcal{R} = \{\alpha_1\alpha_2 \cdots \alpha_{k+1}\#, \alpha_2\alpha_3 \cdots \alpha_{k+1}\#\alpha_1, \ldots, \alpha_{k+1}\#\alpha_1\alpha_2 \cdots \alpha_k\}$ would be searched for with the suffix filter stategy. Here the comparison to candidate text substring needs to be done cyclically according to the rotation of the pattern (marked by special symbol $\# \notin \Sigma$ not part of the comparison).

It is then possible to see that all combinations to distribute errors to the pieces are covered by this *rotation pruning*. The following proof is by induction. Let $m(\alpha_i)$ be the number of mismatches in $\alpha_i$ and $\sum_{i=1}^{k+1} m(\alpha_i) = k$. We show that

there is a rotation $P' = \alpha'_1\alpha'_2\cdots\alpha'_{k+1}$ of $P$ such that $\sum_{i=1}^{p} m(\alpha'_i) \leq p-1$ for $1 \leq p \leq k+1$. If $k = 0$, the statement is obviously true. If $k > 0$, we can rotate $P$ into $Q = \alpha''_1\alpha''_2\cdots\alpha''_{k+1}$ such that $m(\alpha''_1) = 0$ and $m(\alpha''_2) > 0$. Then we set $B = \beta_1\cdots\beta_k = \alpha''_2\cdots\alpha''_{k+1}$ and allow $m(\beta_1) = m(\alpha''_2) - 1$ and $m(\beta_i) = m(\alpha''_{i+1})$ for $i > 1$. Now there is, by induction, a rotation $\beta'_1\beta'_2\cdots\beta'_k$ of $B$ such that $\sum_{i=1}^{p} m(\beta'_i) \leq p-1$ when $1 \leq p \leq k$. This gives us the desired rotation of $P$: if the desired rotation of $B$ begins at $\beta_r$, then the desired rotation of $P$ is $P' = Q$, if $r = 1$, otherwise it is $P' = \alpha''_{r+1}\cdots\alpha''_{k+1}\alpha''_1\cdots\alpha''_r$.

Rotation pruning finds hence directly all $k$-mismatch occurrences without needing to resort to verification. Since suffixes are prefixes of rotations, suffix filter finds all the real occurrences as rotation pruning but in addition some that will be filtered out in a verification step.

It is now wortwhile to compare suffix filter with the case analysis pruning mechanism that was considered in the previous section. For $k = 1$, pattern is split into $P = \alpha\beta$ and the case analysis pruning searches $P$ in both directions so that first $|\alpha|$ ($|\beta|$) symbols match exactly before branching. Suffix filter can only exploit one of the directions, since it searches suffix $\beta$ exactly. Therefore it needs to resort to expensive verification step on that part. For larger $k$, the comparison is difficult since although suffix filter can always start the search with zero errors, it may end up doing lot of verification, whereas case analysis pruning always has cases where errors need to be allowed in the beginning but alltogether no verification is required afterwards.

Rotation pruning was introduced here as a tool for showing correctness of suffix filter, but the interesting question is whether it could be applied directly. This is a desirable goal, as rotation pruning shares the good properties of both suffix filter (starting the search with zero errors) and the case analysis pruning (no verification step needed). Applying rotation pruning is indeed possible, with some limitations: Assume we know pattern length $m$ beforehand. Then we can slide a windows of length $m$ over the text $T$ and produce a set of repetitions of the windows

$$\begin{aligned}\mathcal{W} = \{&t_1t_2\cdots t_m\#t_1t_2\cdots t_m,\\ &t_2t_3\cdots t_{m+1}\#t_2t_3\cdots t_{m+1},\\ &\cdots,\\ &t_{n-m+1}t_{n-m+2}\cdots t_n\#t_{n-m+1}t_{n-m+2}\cdots t_n\}\end{aligned}$$

where we have added a special symbol $\# \notin \Sigma$ to mark the rotation positions. Now consider a reverse FM-index for $\mathcal{W}$ or more precicely, a FM-index for a concatenation of strings $\{W^r \mid W \in \mathcal{W}\}$ with another special symbol \$ added between two strings. Backward backtracking on each $S^r$ such that $S \in \mathcal{R}$ with the suffix filter strategy (growing number of errors in the backtracking path) and with forcing symbols # match, implements correctly rotation pruning.

The shortcoming of rotation pruning (in addition to a fixed pattern length) is that the space grows to $mn \log \sigma(1 + o(1))$ bits. However, the repetitions created into $\mathcal{W}$ are amenable for compression: It is shown in [18] that on repetitive collections like $\mathcal{W}$ the Burrows-Wheeler transform contains long runs of

symbols. This can be stated formally on collections created from a random string $T$. The idea is that for a random string $T$ the longest repeating substring is of length $L = O(\log_\sigma n)$. The corollary is that the position of each symbol in Burrows-Wheeler is decided at most by its $L$-length context. If you create a Burrows-Wheeler transform for a collection of substrings of $T$, then even if you modify the end of the substrings, the mapping of symbols to Burrows-Wheeler transform that are further away than $L$ positions from the end of the substring will be unaffected. Namely, a symbol $t_i$ that is copied to many substrings and at distance larger than $L$ from the end of each of the substrings, will be mapped to a run of $t_i$'s in the Burrows-Wheeler transform of the collection. Following the analysis of [18] (almost verbatim, associating rotation positions and end of substrings with mutations) one can conlude that the Burrows-Wheeler transform of $\mathcal{W}$ on random text $T$ contains $R = \min(mn, O(n \log_\sigma n))$ runs on average. Using the RLFM+ index [18] in place of FM-index, one obtains an index cabable for rotation pruning that occupies $(R \log \sigma + 2R \log \frac{mn}{R})(1+o(1)) + O(R \log \log \frac{mn}{R}) + n \log n + O(nm \log(nm)/d)$ bits, where $d$ is a given parameter affecting the running time $O(d(\log \sigma + (\log \log n)^2))$ of computing $\mathsf{SA}[i]$. A simplified upper bound for the space is $O(n \log n \log m)$ by setting $d = m/\log m$. Each backtracking step (LF-mapping) is reasonably fast with the approach: $O(\log \sigma + (\log \log n)^2)$.

## 3 Dynamic Programming and Backtracking

Dynamic programming is a standard technique for searching for approximate occurrences of a pattern in a text. The most common criterion for a match is the Levenshtein distance: the number of substitutions, insertions and deletions needed for matching the pattern. Dynamic programming can be applied to backward backtracking by building one column of the dynamic programming table on each recursive step. The following pseudocode illustrates the idea:

**Algorithm.** `kerrors`$(sp, ep, count, oldcol)$

```
(1)  if (sp > ep) return ;
(2)  if (oldcol[patlen] ≤ klimit) then
(3)      Report occurrences SA[sp], ..., SA[ep]; return ;
(4)  for each s ∈ Σ do
(5)      sp' ← C[s] + rank_s(L, sp − 1)+1;
(6)      ep' ← C[s] + rank_s(L, ep);
(7)      curcol[0] ← count; minval ← count;
(8)      for i = 1 to count do
(9)          if (P[patlen − count + 1] = s) then diag ← 0; else diag ← 1;
(10)         curcol[i] ← min(oldcol[i] + 1, curcol[i − 1] + 1, oldcol[i − 1] + diag);
(11)         if (curcol[i] < minval) then minval ← curcol[i];
(12)     if (minval ≤ klimit) kerrors(sp', ep', count + 1, curcol);
```

This procedure is called `kerrors`$(1, n, 1, firstcol)$ where $firstcol[i] = i$. The arrays *oldcol* and *curcol* contain values of two consecutive columns of the dynamic programming table. The minimum value of the current column (*minval*) is calculated to terminate the search if no further matches are possible.

Dynamic programming can also be used when there can be gaps in the pattern. This is the case when RNA-seq reads have to be located in the DNA. Now, when building the dynamic programming table, one has to consider all previous columns where a gap could have begun and select the minimum value among them. This can be accomplished more efficiently by creating a *Cartesian tree* [6] for each row in the table as proposed in [17]: Cartesian tree is defined on a sequence of numbers so that its root is the minimum element, left child of root is the minimum element from left side of to the root, and right child is the minimum element from right side of the root. Then the whole tree is defined recursively in the same fashion splitting the sequence to left and right parts in each node. It can be constructed incrementally in linear time [6], amortized constant time per step. The construction can easily be modified to maintain minimum at the root when sliding a window through the sequence of numbers, and also in the case when the window slides back and forth in the sequence (although then the amortized running time analysis needs to be revised, see below).

This latter case is exactly the algorithm that we can use in the above dynamic programming through backtracking -approach; Cartesian tree of a sliding window (of length maximum intron length) for each dynamic programming row is maintained during the backtracking search that goes back and forth the virtual suffix tree paths. For this to work, we need to store all the columns from root to the maximal depth (maximum intron length plus read length plus $k$). The sliding window does not need to start from the current position in the matrix, but one can adjust it to disallow too short introns. It is also possible to make it contain only values from plausible exon boundaries (obeying the dinucleotide markers in intron ends).

The worst case running time of the approach is however no better than with the naive algorithm; one can amortize the back and forth steps of maintaining the Cartesian tree of a sliding window only on the total length of paths from root to nodes $v \in V$ such that $V$ is the set of nodes where backtracking ends. This can be worse than the size of the sliding window, which bounds the running time of one step in the naive algorithm and which also gives another bound for the running time of one step in the the Cartesian tree approach. The best case running time of the Cartesian tree approach is still constant per step. In our experiments, the average number of updates in Cartesian tree was 1.91 per step.

Another important feature affecting the practical running time is that backtracking needs to enter all paths from a point where a gap is allowed. To alleviate this bottleneck, we can exploit the forward and reverse FM-indexes. Then we can restrict the long gap to start only after the midpoint so that the search space has been pruned enough to make the full branching feasible.

The same idea can be used for mapping mate-pair reads. Consider the search pattern to be concatenation of the two mate-pair ends. Then the search can be done as with RNA-seq reads except that now the long gap position can be limited to only one place. Again it is possible to do the search in both directions with forward and reverse FM-indexes and continue with gap only in the one having smaller search space left.

## 4 Experiments

We implemented the different pruning techniques described in Sect. 1.1 and 3, and also the suffix filter [10] and the standard pattern partitioning filter [23].

For both filters we implemented a preprocessing step that finds an optimal partitioning by dynamic programming such that the total work is minimized. For pattern partitioning filter this can be computed accurately, since one can compute the exact number of occurrences for each substring of the pattern by backward search. Then it is an easy task to find an optimal partitioning such that total number of occurrences (candidates to be checked) is minimized. For suffix filter we used the exact substring occurrences as a start point to estimate how many pattern suffix occurrences there are in expectation. Optimal partitioning in this case gives the minimum number of candidates to be checked using the exact occurrences of the pieces in partitioning as a prior. This should also reflect to the amount backtracking needed although that is not directly optimized.

For prefix pruning our implementation uses only the forward FM-index; the use of reverse FM-index would speed up its preprocessing step, but does not affect the assymptotic behaviour on our test setup (fixed pattern length).

Our implementation was built on top of succinct data structures from the *libcds* library[1]. We have implemented most of the features explained before under the $k$-mismatches model. We are currently extending the implementation to $k$-errors model by plugging in fast bit-parallel dynamic programming routines. Once these are ready, we will set our implementation publicly available[2].

The following subsections give the results of our experiments on ChIP-seq and RNA-seq data. The results on ChIP-seq show that rotation pruning is superior to other methods in terms of time, and also the only method that is practical on high mismatch-rates. The results on RNA-seq show that our method is on par with TopHat.

The experiments were ran on Intel Xeon E5440 CPU with 32 GB of memory and 64 bit Linux OS.

### 4.1 ChIP-seq

For the first experiments, we used a set of 10 000 ChIP-seq reads of length 36 bp taken from [3]. The reads were aligned against *GRCh37* version of the human genome with $k$-mismatch and varying values of $k$. For the second set of experiments, we used a shorter base sequence and included results also for rotation pruning.

Figure 1 gives the average search time per one read when retrieving all occurrences with $k$-mismatches. The left graph contains results for counting all approximate occurrences. In the right graph, all occurrences are also located. Both filtering methods can locate occurrences without extra cost while checking the candidate matches. Other methods have to look for a sampled suffix position

[1] `http://code.google.com/p/libcds/`

[2] `http://www.cs.helsinki.fi/group/suds/`

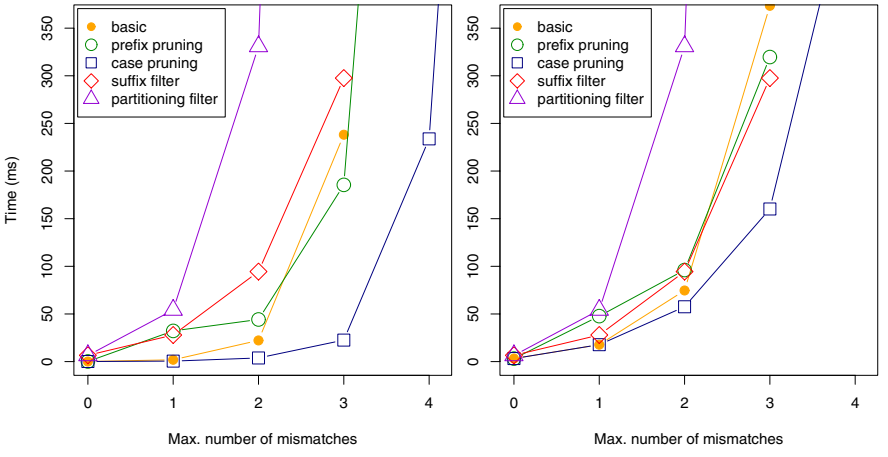



**Fig. 1.** Average search time per read when *counting* (left) or *locating* (right) all occurrences. Averages of 10 000 reads aligned against the human genome.

via LF-mapping, thus, their time complexity depends on the sampling density. In this case, every 32nd suffix was sampled.

Figure 2 gives the average search time per one read when only the *best* occurrences are retrieved. The best occurrence is defined as the first occurrence found with the least number of mismatches. For genome scale data, there are two distinct observations: The partitioning filter does not perform well because the number of candidate occurrences grows substantially fast. The prefix pruning includes a large overhead from the preprocessing making it faster than the basic backtracking only at higher values of $k$. These two effects vanish if reads are matched against a shorter base sequence.

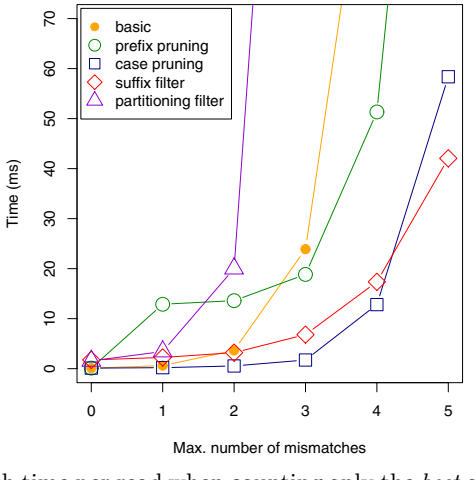



**Fig. 2.** Average search time per read when counting only the *best* occurrences. Averages of 10 000 reads aligned against the human genome.

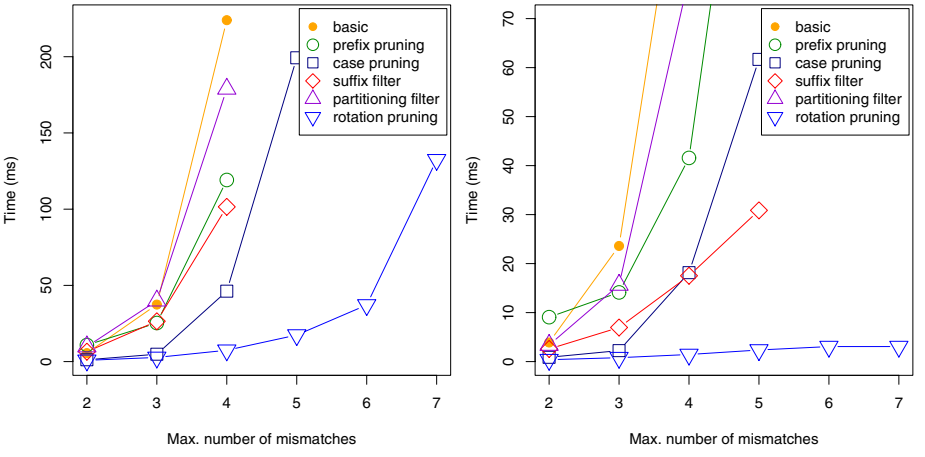



**Fig. 3.** Average search time per read when counting *all* occurrences (left) or only the *best* occurrences (right). Averages of 10 000 reads aligned into a 40MB DNA sequence.

Figure 3 contains average search times against a 40 MB DNA sequence taken from human chromosome 1. For the shorter base sequence, we took a set of 10 000 reads that all align with at most 6 mismatches. The results in the left graph give average times for counting all occurrences with $k$-mismatches. In the right graph, only the best occurrences are counted for. Both graphs also include the rotation pruning which clearly outperforms other methods for $k \geq 4$ mismatches.

Table 1 summarizes memory consumption of different methods for the 40 MB DNA sequence. Case analysis pruning uses twice as much memory compared to other pruning methods and filters. The rotation index was built on run-length encoded indexes, namely RLWT and RLCSA [18], to prevent the index size from doubling when the read length is doubled. Although the results reported in Figure 3 did not use a run-length encoded index, RLCSA have been shown to be competitive in terms of time to the techniques used [18].

As a sanity test, we compared our case analysis pruning against bowtie 0.11.3 [12]. We took a set of million reads from [3] and matched them against the whole human genome using the default settings of bowtie. We also tested bowtie with the option *-v 2* and options *-n 2 -l 36* but their effect was negligible. For bowtie, it takes about a minute to align the given reads on default settings. Our implementation, with similar settings, was about 11 times slower. This shows

**Table 1.** Memory usage of different indexes for a 40 MB DNA sequence. Read length has only a subtle effect on the index size for rotation pruning.

| Read length | 36 bp | 72 bp |
|---|---|---|
| Rotation pruning, RLWT | 1 598 MB | 1 694 MB |
| Rotation pruning, RLCSA | 1 071 MB | 1 143 MB |
| Case analysis pruning | 56 MB | |
| All other methods | 28 MB | |

the effect of other more subtle heuristics and code-level optimizations in bowtie. Because bowtie is optimized for small number of mismatches, say for $k \leq 3$ mismatches, we did not compare it against the rotation pruning. By extrapolating from the results in Figure 3, we can estimate that rotation pruning could align one million high-quality reads per hour with any $k \leq 7$. The rotation pruning index for human genome would easily fit in 64 GB of main memory.

### 4.2 RNA-seq

We ran a test on simulated RNA-seq data that contained no errors at all. We chose one protein coding gene[3] that spans over 8 exons. Its exon and intron lengths vary from 75 to 805, and from 558 to 18 143 base-pairs, respectively. We generated 1 469 reads of length 36 bp from the gene's cDNA sequence, that is, one read from each suffix of the original sequence. We did not induce errors to the reads since it is non-trivial to simulate real-world sequencing errors.

Both our method, described in Sect. 3, and TopHat were able to align the given reads within a few minutes. While our method was about three times faster, doubling the number of reads did not seem to affect TopHat's running time as much as ours. Both methods aligned about 95% of reads correctly. Our method recognized all introns correctly but had problems aligning reads that span over the intron close to the very beginning or end of the read — a situation that is difficult to improve on. TopHat had similar problems, but more notably, TopHat managed to find all but one intron. TopHat did not recognize the second intron nor any of the reads that span over it, not even by doubling the coverage of input reads.

## References

1. Burrows, M., Wheeler, D.: A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation (1994)
2. Cole, R., Gottlieb, L.-A., Lewenstein, M.: Dictionary matching and indexing with errors and don't cares. In: Proceedings of the Thirty Sixth Annual Symposium on the Theory of Computing, pp. 91–100 (2004)
3. Tuupanen, et al.: The common colorectal cancer predisposition snp rs6983267 at chromosome 8q24 confers potential to enhanced wnt signaling. Nature Genetics 41, 885–890 (2009)
4. Ferragina, P., Manzini, G.: Indexing compressed texts. Journal of the ACM 52(4), 552–581 (2005)
5. Ferragina, P., Manzini, G., Mäkinen, V., Navarro, G.: Compressed representations of sequences and full-text indexes. ACM Transactions on Algorithms (TALG) 3(2), article 20 (2007)
6. Gabow, H.N., Bentley, J.L., Tarjan, R.E.: Scaling and related techniques for geometry problems. In: Proc. 16th ACM Symposium on Theory of Computing (STOC 1984), pp. 135–143 (1984)

[3] Gene `CD53-001` and its transcript `ENST00000271324`.

7. Harismendy, O., Ng, P.C., Strausberg, R.L., Wang, X., Stockwell, T.B., Beeson, K.Y., Schork, N.J., Murray, S.S., Topol, E.J., Levy, S., Frazer, K.A.: Evaluation of next generation sequencing platforms for population targeted sequencing studies. Genome Biology 10(R10) (2009)
8. Johnson, D.S., Mortazavi, A., Myers, R.M., Wold, B.: Genome-wide mapping of in vivo protein-dna interactions. Science 316(5830), 1497–1502 (2007)
9. Jothi, R., Cuddapah, S., Barski, A., Cui, K., Zhao, K.: Genome-wide identification of in vivo protein-dna binding sites from chip-seq data. Nucl. Acids Res. 36(16), 5221–5231 (2008)
10. Kärkkäinen, J., Na, J.C.: Faster filters for approximate string matching. In: Proc. 9th Workshop on Algorithm Engineering and Experiments (ALENEX 2007), pp. 84–90. SIAM, Philadelphia (2007)
11. Lam, T.W., Sung, W.K., Tam, S.L., Wong, C.K., Yiu, S.M.: Compressed indexing and local alignment of dna. Bioinformatics 24(6), 791–797 (2008)
12. Langmead, B., Trapnell, C., Pop, M., Salzberg, S.L.: Ultrafast and memory-efficient alignment of short dna sequences to the human genome. Genome Biology 10(3), R25 (2009)
13. Li, H., Ruan, J., Durbin, R.: Mapping short DNA sequencing reads and calling variants using mapping quality scores. Genome Research 18, 1851–1858 (2008)
14. Li, H., Durbin, R.: Fast and accurate short read alignment with burrows-wheeler transform. Bioinformatics (2009) (Advance access)
15. Li, R., Li, Y., Kristiansen, K., Wang, J.: Soap: short oligonucleotide alignment program. Bioinformatics 24(5), 713–714 (2008)
16. Li, R., Yu, C., Li, Y., Lam, T.-W., Yiu, S.-M., Kristiansen, K., Wang, J.: Soap2. Bioinformatics 25(15), 1966–1967 (2009)
17. Mäkinen, V.: Parameterized Approximate String Matching and Local-Similarity-Based Point-Pattern Matching. PhD thesis, University of Helsinki (2003)
18. Mäkinen, V., Navarro, G., Sirén, J., Välimäki, N.: Storage and retrieval of individual genomes. In: Batzoglou, S. (ed.) RECOMB 2009. LNCS, vol. 5541, pp. 121–137. Springer, Heidelberg (2009)
19. Manber, U., Myers, G.: Suffix arrays: a new method for on-line string searches. SIAM Journal on Computing 22(5), 935–948 (1993)
20. McCreight, E.: A space-economical suffix tree construction algorithm. Journal of the ACM 23(2), 262–272 (1976)
21. Morin, R.D., Bainbridge, M., Fejes, A., Hirst, M., Krzywinski, M., Pugh, T.J., McDonald, H., Varhol, R., Jones, S.J.M., Marra, M.A.: Profiling the hela s3 transcriptome using randomly primed cdna and massively parallel short-read sequencing. BioTechniques 45, 81–94 (2008)
22. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Computing Surveys 39(1), article 2 (2007)
23. Navarro, G.: A guided tour to approximate string matching. ACM Comput. Surveys 33(1), 31–88 (2001)
24. Trapnell, C., Pachter, L., Salzberg, S.L.: Tophat: discovering splice junctions with Rna-seq. Bioinformatics 25(9), 1105–1111 (2009)
25. Ukkonen, E.: On-line construction of suffix trees. Algorithmica 14(3), 249–260 (1995)
26. Wang, Z., Gerstein, M., Snyder, M.: Rna-seq: a revolutionary tool for transcriptomics. Nature Reviews Genetics 10(1), 57–63 (2009)
27. Weiner, P.: Linear pattern matching algorithm. In: Proc. 14th Annual IEEE Symposium on Switching and Automata Theory, pp. 1–11 (1973)

# Some Applications of String Algorithms in Human-Computer Interaction

Kari-Jouko Räihä

Unit for Computer-Human Interaction (TAUCHI),
Department of Computer Sciences
FIN-33014 University of Tampere, Finland
kari-jouko.raiha@cs.uta.fi

**Abstract.** Two applications of string algorithms in human-computer interaction are reviewed: one for comparing error rates of text entry techniques, another for abstracting collections of scan paths (paths of eye movements). For both applications, the classic string edit distance algorithm proves useful. For the latter application shortest common supersequences provide one option for further development. Applying them as such could be misleading, but a suitable approximation could provide a useful representation of a set of scan paths.

**Keywords:** text entry, edit distance, Levenshtein distance, minimum string distance, gaze input, usability evaluation, scan path comparison, shortest common supersequence.

## 1 Introduction

String algorithms have a huge number of applications in probably all fields of science. In human-computer interaction, there are applications (such as efficient information retrieval) that critically depend on efficient string algorithms. Here I focus on the use of string algorithms in analyzing interaction techniques and interaction activities. Two areas are considered: text entry techniques and interaction with graphical user interfaces, analyzed with the help of eye tracking. Well-known string edit distance algorithms prove useful for both, and shortest common supersequences offer some promise for the latter, which still calls for definite, broadly accepted solutions. I review both established approaches and discuss some possibilities for further research.

## 2 Text Entry and Minimum String Distance

Written communication has a long history, going back thousands of years to the Sumerians. Technological innovations like the printing press, typewriters, and computers have increased the ease and speed of producing written text. The basic principle has, however, remained much the same: the resulting text is composed a character at a time, be it using the movable type of a printing press or keypresses on the typewriter or computer keyboard. Silfverberg [31] provides an excellent overview of the historical development of text entry technologies.

T. Elomaa et al. (Eds.): Ukkonen Festschrift, LNCS 6060, pp. 196–209, 2010.

The proliferation of mobile devices (mobile phones and palmtop computers) in the last two decades has opened a Pandora's box in text entry research and development. The challenge, of course, is the small size of the device: it can no more incorporate a full-size keyboard, causing the need for radically new ways of entering text. The multi-tap technique used in mobile phones, whereby each key of the keypad can be used to select a character among several alternatives by pressing it repeatedly, is widespread. But there are a number of other techniques, including predictive techniques such as T9 [9], marks that resemble handwritten characters such as Graffiti [7], continuous techniques like Dasher [38] (a radical departure from traditional discrete, hunt and peck techniques), and even a technique that requires just one key [16]. As stated by MacKenzie and Tanaka-Ishii in their recent compendium on text entry systems, "Text entry has never been as important as it is today, because of the huge success of mobile computing and text messaging on mobile phones and other small devices like the PDA and hand-held PC." [17].

### 2.1 Metrics for Evaluating Text Entry Techniques

The increase in the variety of techniques available for text entry has brought along a need for their systematic evaluation. In early days, the main comparisons were between the various layouts of the keys on the keyboard. The design of the standard layout today, the so-called QWERTY keyboard, was driven by constraints of the technology: keys were placed in locations that would minimize the chance of two consecutive key presses jamming the corresponding levers. Technology advanced and this design criterion lost its importance, and later studies found that some other layouts, notably the so-called Dvorak layout [3], would be better for human motor performance and, consequently, text entry speed. Nevertheless, the QWERTY layout had by then reached a status where switching to another layout was no more feasible, given the fairly small improvements (10–20 %) in entry speed that could be expected after a long training period [31, p. 10].

Today the situation with new mobile devices and text entry techniques is different. New techniques are invented frequently, and they can be evaluated, analyzed, and refined before it is too late in the sense that a de-facto standard would prevent better techniques from getting adopted.

What, then, are the appropriate metrics that should be used in the evaluation and comparison? Obviously, text entry rate, or the speed of producing the written text, is a key metric. This can be evaluated using established methods by asking test participants to produce text, either at will ("composition") or by copying existing text presented to them in a suitable way ("transcription"). The latter, of course, allows more objective comparisons. Expert typists can achieve a rate of more than 100 *words per minute* (WPM), while the world record according to the Guinness Book of World Records [23] is more than 200 WPM (obtained using the Dvorak layout). Here the "word" in the "words per minute" has been normalized to equal five characters, to accommodate the variation in word lengths. Thus, "Esko Ukkonen" would count as 2.4 words.

This, however, is not the only important metric. Wobbrock [41] and MacKenzie [14] provide surveys on the various metrics and evaluation methods that can and have been used with text entry systems. In addition to entry rate, another important metric is the expected error rate of a technique: the usefulness of a fast technique is greatly

undermined if it is highly error prone, that is, if users frequently produce incorrect characters in the text.

Here we come across the first classical string algorithm: the one used for computing the Levenshtein distance, or edit distance, or (as it is customarily called in text entry research), minimum string distance. Let *S* be a source string over an alphabet and *T* the target string over the same alphabet (that is, *S* is the model to be copied, and *T* is the text produced by the typist). Then MSD(*S*, *T*) is the minimum number of edit operations needed to transform *S* into *T*. Here the edit operations are deleting a character, inserting a character, or changing a character into another character. In text entry research, it is customarily assumed that all operations are equally costly, i.e., the minimum string distance is computed using the unit cost model. As noted by Ukkonen [37], the algorithm for computing the minimum string distance has been reinvented several times in various contexts, and text entry research adds to the long list.

The minimum string distance can then be used to define a metric for the error rate: the "*MSD error rate*" for strings *S* and *T*, $\text{MSD}_{\text{error}}(S, T)$, is defined as MSD(*S*, *T*) divided by the maximum of the lengths of *S* and *T* [32]. In other words, we obtain a normalized metric based on MSD by taking into account the lengths of the strings *S* and *T*. This will allow its robust application over text entry tasks for strings of varying length.

The MSD error rate works well for evaluating the correctness of the output of the text entry task, but it still does not tell the whole story. The result may be perfectly correct if it is obtained by meticulously correcting all errors produced during the text entry task. This would obviously compromise text entry speed at the expense of correctness of output. For this reason the use of a third metric is customary to support a comprehensive view of the performance of the techniques. *Keystrokes per character* (KSPC) is defined as the ratio of number of key presses and the number of characters in the resulting text. This is a useful metric for evaluating the performance during a single text entry task. The same principle can also be applied more generally using the so-called KSPC characteristic measure [15]: it provides a metric for comparing the best-case performance of text entry techniques. For a typewriter and for text that only consists of lower case letters, the KSPC characteristic measure equals 1. Introducing capital letters in the text, and thereby making the use of the shift key necessary, increases KSPC. With the multi-tap method of mobile phones, where several key presses are needed for many characters, the KSPC characteristic measure is more than 2. MacKenzie [15] and Wobbrock [41] give more details.

This combination of metrics: WPM, $\text{MSD}_{\text{error}}$, and KSPC, allows a balanced comparison of the merits of text entry techniques. KSPC, in particular, allows the evaluation of a technique both before it has even been implemented (using the KSPC characteristic measure) and in actual use (using the KSPC performance measure).

### 2.2 Text Entry by Eye Gaze

The development of devices is not the only change that is taking place in the world of text entry. New modalities are brought to use as well. In particular, text entry by eye gaze has attracted increased attention [22, 19], because it may be the only available form of communication for a select user group, such as those suffering from a stroke or advanced stages of ALS. Here, too, techniques based on keyboards have been dominant, but a number of new techniques have emerged.

When eye gaze is used to enter text with a keyboard, the keyboard is displayed on the computer screen. Instead of a physical key press, a key is activated by eye gaze by looking at it for long enough, longer than a threshold set in the system. This threshold is called the *dwell time* needed for key activation.

Such a technique has been used with success for some decades now, although the reported text entry rates have remained rather low, between 7 and 12 WPM [4, 21]. This is natural, as the dwell time sets an upper limit for the maximum obtainable speed, and the single input channel does not allow parallel activities, like the use of the ten fingers on the traditional keyboard.

It is no wonder, then, that the introduction of Dasher in the turn of the century [38, 39] attracted a lot of attention both in the research community and in the press. The basic mode of operation of Dasher is shown in Figure 1. It makes use of text prediction, allocating space to the characters in proportion to their likelihood based on the prefix of the word entered thus far. Moreover, it is a dynamic system: the characters grow and move to the left, towards the user's cursor. This rate can be controlled by the user by moving the cursor further to the right (to increase the speed) or to the left (to slow down).

Dasher is a technique that learns as it is being used. It updates the language model that is used for prediction by adding in the corpus the text entered by the user. In addition, it also adapts its basic rate of moving the characters towards the cursor: as the user becomes more skilled and pushes forward with faster rate, Dasher also increases the baseline rate, thereby allowing the user to achieve a speed that is personalized to the skill and preferences of each individual user.

Another remarkable property of Dasher is that it can be controlled by a variety of input devices, including a mouse, trackball, stylus, and eye tracker. The technique looks bewildering when looked from the side for the first time, but it feels very different to the

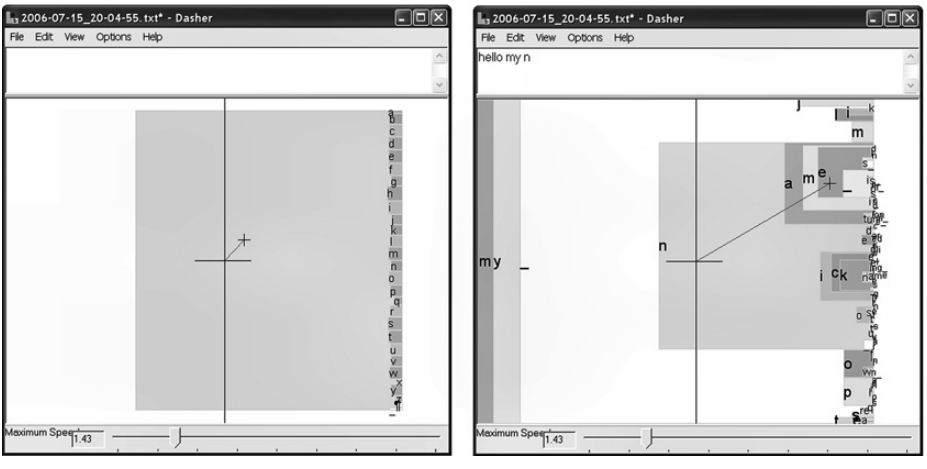



**Fig. 1.** Dasher in its initial state (on the left) and in the middle of entering the word "name". Note how the letters that can follow "n" (already entered by virtue of having moved to the left beyond the center line) occupy varying amounts of space, with "a" being the most likely letter to follow "n", and "name" being the most likely word to start with "n". [22]

person using the system: the experience is similar to steering a car through traffic, which, too, may look scarier to someone not personally in control of the vehicle.

Dasher does not have the intrinsic limitation of text entry rate caused by the dwell time. In 2002 Ward and MacKay reported [39] an entry rate of 25 WPM after just an hour of practice, and a top rate of 34 WPM for an expert user. These were striking numbers compared to the rates achieved previously, and Dasher has since been considered as the fastest way of entering text by eye gaze.

The rates reported by Ward and MacKay were obtained with just a few participants, and with the experts being the developers themselves. In particular, it can be expected that Dasher takes some time to learn, given that it is radically different from keyboard-based techniques. We therefore carried out a longitudinal study to investigate the learning curve and to verify the previous results with a somewhat larger group of users [35, 19].

The study involved 11 users, of which one was a clear outlier and left out of the following numbers. All users typed text for 15 minutes a day in 10 consecutive days. For details of the setup, see [35, 19]. Table 1 summarizes the results. BS is the number of backspace operations, which were entered by moving the cursor left, beyond the center line, causing the letters to detract. This metric is included in lieu of KSPC, which is not applicable in the context of Dasher, since it does not have distinct keystrokes.

**Table 1.** Performance metrics for Dasher controlled by eye gaze. Data for 10 users from 10 trials of 15 minutes each. BS is the number of backspace operations.

| Trial | Average WPM | Average $MSD_{error}$ | Average BS |
|---|---|---|---|
| First | 2.49 | 10.72 % | 0.26 |
| Last | 17.26 | 0.57 % | 0.13 |

Table 1 shows that considerable learning has taken place. In fact, the learning curve [35, 19] is quite exceptional; it does not show signs of leveling off even after 2.5 hours of practice. Thus it can be expected that the participants could have reached similar performance as reported previously [39], albeit with some more practice. The best entry rate achieved by any participant in this experiment was 23.11 WPM. The language model, which was based on a much bigger corpus in the original study done in English, surely had an effect on the lower rates achieved in the longitudinal study carried out with the Finnish language model.

In addition to being a fundamentally different technique for entering text, Dasher has another difference to the way the dwell-based methods have been used in previous studies. Dasher not only adjusts its speed by observing the user, but also lets the user set the speed manually through a selection box in the interface. The latter is not common in dwell-based techniques, but there is no reason why it could not be done. The question then naturally arises: how much were the previous poor results on text entry rate using dwell-based keyboards due to a dwell time threshold that prevented the users from achieving the best rate they would have been capable of?

To study this question, a similar longitudinal study as with Dasher was carried out using a dwell-time adjustable soft keyboard [20, 19]. A simple lever was added to the standard soft keyboard, and two buttons allowed the user to control the dwell time threshold. Table 2 summarizes the results.

**Table 2.** Performance metrics for a soft keyboard with adjustable dwell time controlled by eye gaze. Data for 9 users from 10 trials of 15 minutes each.

| Trial | Average WPM | Average $MSD_{error}$ | KSPC |
|---|---|---|---|
| First | 6.90 | 1.28 % | 1.09 |
| Last | 19.89 | 0.36 % | 1.18 |

The results are in striking contrast with what had been the common belief before the experiment. Text entry by dwell time is *not* inherently slow, provided that users are given tools to adjust the interface to their liking. The only aspect where the figures are not favorable to the technique is the increase in KSPC, but this is not surprising. The speed-accuracy tradeoff is a familiar phenomenon, and the best text entry rates in Table 2 were achieved with really low dwell times between 200 and 400 ms. Thus an increase in the need of corrective actions is understandable.

In the case of the dwell-based technique the learning curve had started to level off, as opposed to the situation with Dasher. An even longer longitudinal study would be needed to find out the real expert performance with the techniques. Most importantly, the metrics show that both techniques are usable, and can perform with satisfactory efficiency after a reasonable practice time.

The minimum string distance does a good job in indicating the errors remaining in the resulting text, and it has become a standard tool in text entry research. Nevertheless, there might be room for some variations. In particular, it can be questioned whether the unit cost model is the best in the comparisons. For instance, if the source text has a capital letter and the user enters the correct letter but in lower case, should this be counted as equally costly as entering an entirely different letter? Or is entering a wrong punctuation mark as bad as omitting the punctuation mark completely? Exploring such alternatives could have marginal interest, but most likely they would not change the big picture in any significant way. They could be more useful in finding out the kind of errors that are typically made using each technique, an area that is still largely unexplored.

## 3 Scan Paths and Shortest Common Supersequences

In the previous section I discussed examples of how eye gaze can be used for controlling computers. The recent boom in the use of eye trackers has, however, been in collecting information about how users view the computer screen, without it having an effect on the interaction. This has proven a good source of information on, for instance, the effectiveness of web page layout, and on the learnability of graphical user interfaces.

In a typical web usability study, users are asked to study a web page, either freely or to find an answer to a specific question or solution to a problem. While they are carrying out this task, their eye movements are recorded. The gaze points collected by the eye tracker are automatically categorized into fixations, gaze points that are close together and used to perceive information, and saccades, quick movements between fixations. For analysis, the result can be overlaid on top of the screen image to show how the user's gaze has moved.

Figure 2a shows the data for one participant studying a web page produced by Google for the search on “shortest common supersequence”. The user was checking the results to see whether there were any links that would be relevant for approximating the supersequence. The customary coding of eye movement data is that fixations are depicted as circles, with the diameter of the circle indicating the duration of the fixation, i.e., how long the eye gaze stayed fixated in the same spot. Saccades are depicted as lines connecting the fixations. The fixations can be numbered to indicate the order in which they took place. The resulting directed graph is called a scan path.

The visualization in Figure 2a is reasonably readable, although there is some clutter that prevents the full path from being seen clearly in a single picture. However, usability studies are not carried out with a single user, but with several users, usually in the tens or even hundreds. It becomes tedious and uninformative to study each resulting scan path separately. There is a need for a good visualization showing several scan paths at the same time.

Figure 2b uses the same technique as Figure 2a, but now for four scan paths (for four test participants, each visualized in different color) in the same picture. Already with such a small number of paths the visualization becomes almost useless. Better techniques are clearly needed. An obvious solution is to reduce the amount of information and find suitable abstractions.

The most popular approach today is to ignore the order of the fixations completely and to simply base the analysis on how long the gaze has been fixated on different areas of the screen. By suitable smoothing algorithms, the level of attention attracted by the different areas can be visualized in an easily perceived form using different levels of shading or colors, in the style familiar from maps. Such visualizations have been called “attentional landscapes” [25] and “fixation maps” [42]. Nowadays “heatmaps” is the most common term used for this concept.

Heatmaps serve well many purposes, especially those where the order of visiting the different areas of the screen is not important. With some stretching of imagination,

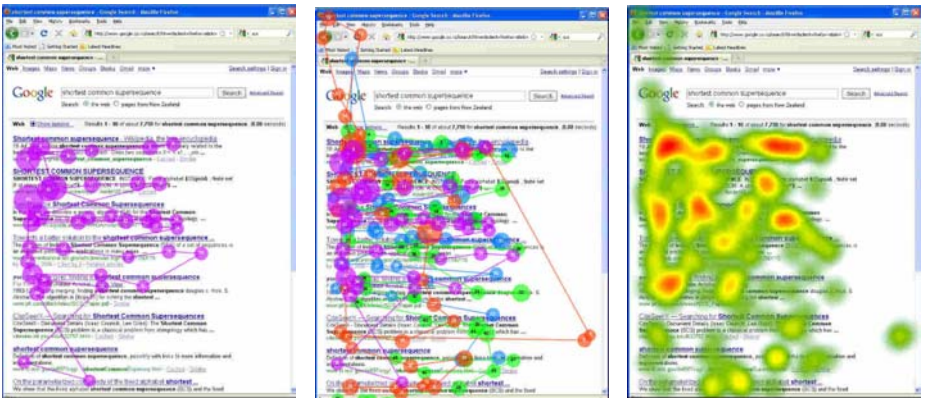


**Fig. 2.** Scan paths of users scanning hits produced by Google for a search on “shortest common supersequence”. Data for (a) one user on the left and for (b) four users in the middle. On the right (c), the same data as in the middle but visualized as a heatmap.

one can see in Figure 2c an "F-shape", which has been claimed to be common browsing and reading behavior for a variety of web pages [24]. Heatmaps can show if some areas have been viewed rarely and for a short time, and also which are the areas attracting the most attention. If these do not match the intention of the page designers, the heatmap is convincing evidence that redesign is in order.

For other tasks, however, heatmaps abstract away too much of the information. Suppose the researcher is interested in how the page of hits produced by Google is used. Do users typically read it in its entirety, or do they optimistically browse further once a first potentially useful link is found? The heatmaps are of limited use for such a research question. Another abstraction that turned out useful for this particular question is to abstract away some of the information concerning the locations of the fixations: here the main interest is in whether an item has been browsed at all, not in how the entry has been read. Omitting the $x$-coordinate from the visualization frees this dimension for visualizing time, so that the $y$-coordinate shows the vertical position of a fixation, but the $x$-coordinate shows the order of fixations. Such a time plot [26] was useful in detecting essentially two different types of browsing behavior of the result page [1].

Result pages of search engines are, however, a special case. Researchers have for some time been in the hunt for a more general visualization that would allow the comparison of scan paths in a general case, without losing the order information. The first step commonly applied in the proposals is omitting the duration information of the fixations. Each scan path then becomes an ordered sequence of nodes—already hinting at the possible usefulness of string algorithms in their comparison.

A critical question is what the nodes in such strings are. We cannot really take the center points of the fixations as nodes, because two users (or one user on several trials) rarely look at exactly the same point on the screen even when perceiving the same information. The common solution is to divide the screen into so-called areas of interest (AOIs) and lump together all fixations in the same area. Then the scan path with the AOIs as nodes shows the order of visitation of the AOIs. Typical AOIs could be the page heading, navigation menu, login field, images, and text blocks. If we label the AOIs with, say, letters, then the similarity of two scan paths can be (again!) computed using the minimum string distance between such strings.

There is a caveat for such an approach, though. Eye trackers are not accurate; the spatial accuracy of modern devices is in the order of 0.5 degrees. On top of this, the human eye is able to perceive accurately an area of 1–2 degrees of visual angle. Taken together, these two factors have the effect that the eye tracker may think that the user is perceiving, say, the right edge of one area, when in fact the content really perceived is in the left edge of the neighboring area on the right. In other words, the mapping of gaze points collected by an eye tracker to the AOIs may be incorrect.

The situation can be somewhat improved by using clustering algorithms, i.e, not using predefined areas of interest, but letting the mapping algorithm find the areas automatically, based on where the gaze points are clustered. Several such algorithms exist [29, 33] and they work well when the information presented is mainly pictorial. For textual information, however, they are of less use, since reading consists of sequences of short fixations, not fixations clustered in the same spot. A hybrid method that combines predefined AOIs and dynamic clustering would probably do a better job than either technique alone.

If the minimum string distance is used to compare two scan paths, the use of the unit cost algorithm is even less likely to be appropriate than in the case of comparing text entry techniques. All areas on the screen are not of equal importance, but the importance cannot be determined universally; it depends on the task. The effects of the task on eye movement data have been known for decades [43]. In the context of usability research, for someone studying the effect of advertisements these areas are the most important, whereas for someone else studying the interaction with the active interface content the advertisements can perhaps be completely ignored. Moreover, the order of visiting the areas can sometimes be important, sometimes meaningless, as long as certain areas are visited in *some* order. Such differences can best be accommodated by assigning different costs to the string editing operations based on the task.

In specific cases, the importance of visiting the various areas in certain order can be determined objectively. A case in point is a study done on the usability of an interactive tool for exploring parallel coordinate visualizations. Parallel coordinates [11] are an unusual way for visualizing multidimensional spaces. Instead of trying to present the points in a traditional coordinate system, the coordinates are laid out beside each other. In effect, a point in the traditional 2- or 3-dimensional space becomes a line in the parallel coordinate visualization. There is no limit on how many dimensions can be handled.

Because the representation differs so much from the traditional way of thinking, it has been questioned whether parallel coordinates are usable, or whether they require a long learning curve. We used eye tracking to study these questions [30]. The participants performed several tasks on a car database visualized using parallel coordinates (Figure 3).

Solving a query using parallel coordinates requires viewing areas in the visualization in certain order. For instance, to solve one of the questions in the experimental task

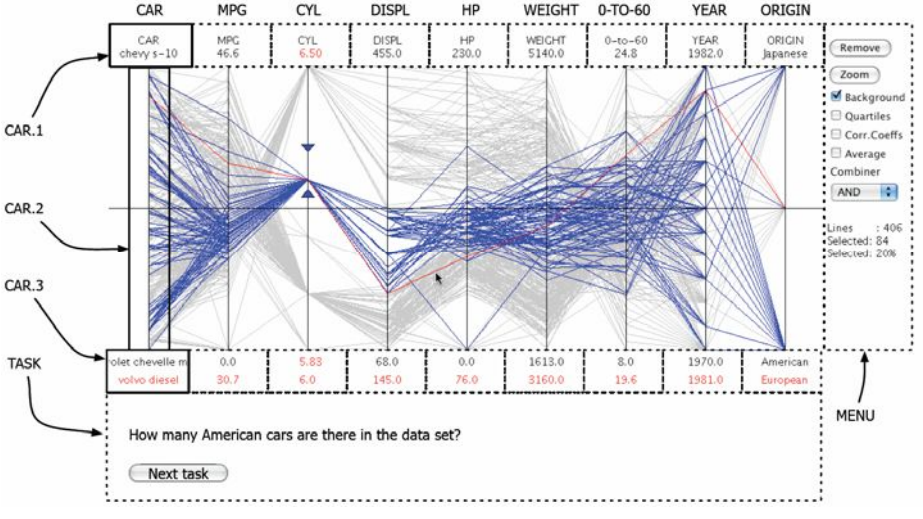



**Fig. 3.** Parallel coordinate visualization of a car database, with areas of interest indicated for eye movement analysis. The interactive query has selected cars with 6 cylinders for inspection. [30].

("What is the most common number of cylinders for cars manufactured in 1973?"), the user has to select the year 1973 from the YEAR.2 area, brush the cylinder number from the CYL.2 area, and read the result from the MENU area. The number of fixations spent in other areas of interest can be taken as a confusion metric: the more fixations focused on other areas, the less clear it was to the participant how to solve the task or where to find the information needed for solving it. Essentially we have a model scan path for optimal performance that can be easily compared to those recorded for the participants.

Returning to the general case: suppose we have a set of scan paths, represented as strings over an alphabet. How can we find a representation for them all? Here, finally, the shortest common supersequences (SCSs) enter the picture. Given two sequences $S_1$ and $S_2$, a sequence $T$ is a common supersequence of $S_1$ and $S_2$ if both $S_1$ and $S_2$ can be obtained from $T$ by deleting zero or more characters. The shortest common supersequence of a set of sequences $\{S_1, S_2, \ldots, S_n\}$ is a sequence $T$ of minimal length such that $T$ is a supersequence of each $S_i$.

From purely a technical point of view, an SCS would be the optimal representation for the entire set of scan paths. Computing an SCS is NP-hard, but the sequences representing scan paths are typically not prohibitively long, and the problem is solvable in practice.

Unfortunately, this straightforward technique does not necessarily produce natural results. The SCS might not be a good approximation of any of the sequences that it represents. For instance, if AB and BA are scan paths for two users, an SCS for AB and BA would be ABA, indicating that the user's gaze would jump back and forth between the two areas, which is not the case for either user. In general, for a set of otherwise identical paths which differ in only one position, such as {ABCA, ABDA, ABEA, ABFA}, one of many similar shortest common supersequences would be ABCDEFA. Intuitively, such a long representation does not convey the essence of the component scan paths.

To remedy such potential problems with SCSs, it has been suggested that an "average" scan path should be used to represent a set of paths [10]. Unfortunately, no details on how such an average representation could be computed were given in [10]; some algorithms have later been suggested by Torstling [34]. In terms of SCSs, an approximate shortest common supersequence [13] could be a step in the same direction. However, it would suffer from the same problems as MSDs for comparing two paths: the sequences should not be treated uniformly. Some changes and omissions are more important than others, and this should be taken into account in building a representation of the whole.

Finally, it has also been suggested that similar scan paths could be clustered together, to find subsets of paths with similarities [40]. The problem is similar to that of clustering partial rankings [36], since each scan path is in essence a ranking of a subset of the AOIs. In this respect, scan path analysis is akin to clickstream analysis. There is evidence that such techniques work adequately, but the final step still calls for a solution: how to represent each cluster, that is, what is it that makes the paths in the cluster similar? Having a good representation for similar paths would be a big step forward in communicating typical scan paths to designers and also in allowing researchers to explore various scanning strategies. The aggregation, comparison, and representation of scan paths remains an active area of research [2, 5, 6, 8, 12].

## 4 Personal Reminescenses

String algorithms and human-computer interaction are both vast research fields, and so is their intersection. The few examples discussed in this essay have been chosen because they remind me of joint history with Esko Ukkonen, whom this Festschrift honors.

String edit distance is an example of a classic concept that can be computed by a dynamic programming algorithm. When we studied together at the University of Helsinki in the 1970ies, computer science was a young discipline and there was not a large number of courses available. While studying for a Master's degree we at some point shared an office and took some courses together. I particularly remember the course on Sorting and Searching, which often led to continued discussions in our office room after getting back from a lab session. It was one of the motivators for continued study of algorithms in a small study group that met weekly and went through some classic works in algorithms and formal languages, without formal supervision. The algorithm for computing the string edit distance was undoubtedly among those studied in the group. Many members of that group have since gone on to become professors of computer science in a number of Finnish universities.

Some years later we had started to pursue our own research interests, with Esko focusing on data structures, parsing theory, and numerical algorithms, and my interest being at the time in compiler-compilers in general and attribute grammars in particular. We had our own offices next to each other, and daily conversations were still frequent. Esko had defended his PhD thesis in computer science as one of the first in Finland, while I was still struggling with mine.

They were days of limited computing power, and much effort went into developing techniques that could be implemented to run efficiently. The particular technique I was working on was the evaluation of attribute grammars in passes. Attribute grammars are a declarative language, and to populate one parse tree with attribute values according to the declarative specification, the tree can be traversed several times consecutively either from left to right or right to left, with a set of attributes being evaluated on each pass. The optimization problem (somewhat academic even by the standards of that time) was to find the best possible ordering for the passes, i.e., decide which passes should go in which direction, and which attributes to evaluate during each pass, in order to minimize the number of passes needed.

After tedious work with trying to solve the problem I had managed to reduce it to a problem that was NP-complete, the shortest common supersequence problem. So it seemed I was done—almost; the catch was that the problem was known to be NP-complete for alphabets of size 5 or more [18]. In my case, the alphabet consisted of just two characters, corresponding to the two possible evaluation directions. After being stuck at this point with no progress, I explained the problem to Esko. I believe he did his usual disappearance act—grabbed a large bottle of Coca-Cola and a box of raisins, and hid in one of the empty classrooms of the department not to be disturbed. After some hours he would emerge with a solution. I would like to think that in this case it was at least some days instead of some hours, since it took me a good amount of time just to verify his construction. It worked, of course, and was eventually published [28], as was the application to the problem of attribute evaluation [27].

I keep these joint experiences in high regard and consider them important contributors to my education in computer science.

**Acknowledgments.** I thank Päivi Majaranta and Saila Ovaska for their comments on the manuscript. This work was supported by the Academy of Finland grant 1130044. The paper was written while the author was visiting the University of Canterbury in Christchurch, New Zealand.

## References

1. Aula, A., Majaranta, P., Räihä, K.-J.: Eye-Tracking Reveals the Personal Styles for Search Result Evaluation. In: Costabile, M.F., Paternò, F. (eds.) INTERACT 2005. LNCS, vol. 3585, pp. 1058–1061. Springer, Heidelberg (2005)
2. Duchowski, A.T., Driver, J., Robbins, A., Ramey, B.N.: Scanpath Comparison Revisited. In: Proc. Symposium on Eye Tracking Research & Applications (ETRA). ACM Press, New York (in press, 2010)
3. Dvorak, A., Merrick, N., Dealey, W., Ford, G.: Typewriting Behavior: Psychology Applied to Teaching and Learning Typewriting. American Book Company, New York (1936)
4. Frey, L.A., White Jr., K.P., Hutchinson, T.E.: Eye-Gaze Word Processing. IEEE Transactions on Systems, Man, and Cybernetics 20, 944–950 (1990)
5. Goldberg, J.H., Helfman, J.I.: Scanpath Clustering and Aggregation. In: Proc. Symposium on Eye Tracking Research & Applications (ETRA). ACM Press, New York (in press, 2010)
6. Goldberg, J.H., Helfman, J.I.: Visual Scanpath Representation. In: Proc. Symposium on Eye Tracking Research & Applications (ETRA). ACM Press, New York (in press, 2010)
7. Graffiti, `http://en.wikipedia.org/wiki/Graffiti_Palm_OS`
8. Grindinger, T., Duchowski, A.T., Sawyer, M.: Group-Wise Similarity and Classification of Aggregate Scanpaths. In: Proc. Symposium on Eye Tracking Research & Applications (ETRA). ACM Press, New York (in press, 2010)
9. Grover, D.L., King, M.T., Kushler, C.A.: Reduced Keyboard Disambiguating Computer. U.S. Patent 5818437 (1998)
10. Hembrooke, H., Feusner, M., Gay, G.: Averaging Scan Patterns and What They Can Tell Us. In: Proc. Symposium on Eye Tracking Research & Applications (ETRA), p. 41. ACM Press, New York (2006)
11. Inselberg, A.: The Plane with Parallel Coordinates. The Visual Computer 1, 69–91 (1985)
12. Jarodzka, H., Holmqvist, K., Nyström, M.: A Vector-based, Multidimensional Scanpath Similarity Measure. In: Proc. Symposium on Eye Tracking Research & Applications (ETRA). ACM Press, New York (in press, 2010)
13. Jiang, T., Li, M.: On the Approximation of Shortest Common Supersequences and Longest Common Subsequences. In: Shamir, E., Abiteboul, S. (eds.) ICALP 1994. LNCS, vol. 820, pp. 191–202. Springer, Heidelberg (1994)
14. MacKenzie, I.S.: Evaluation of Text Entry Techniques. In: [17], ch. 4, pp. 75–101 (2007)
15. MacKenzie, I.S.: KSPC (Keystrokes Per Character) as a Characteristic of Text Entry Techniques. In: Paternó, F. (ed.) Mobile HCI 2002. LNCS, vol. 2411, pp. 195–210. Springer, Heidelberg (2002)
16. MacKenzie, I.S.: The One-Key Challenge: Searching for a Fast One-Key Text Entry Method. In: Proc. ACM Conference on Computers and Accessibility (ASSETS), pp. 91–98. ACM Press, New York (2009)
17. MacKenzie, I.S., Tanaka-Ishii, K. (eds.): Text Entry Systems: Mobility, Accessibility, Universality. Morgan Kaufmann, San Francisco (2007)

18. Maier, D.: The Complexity of Some Problems on Subsequences and Supersequences. Journal of the ACM 25, 322–336 (1978)
19. Majaranta, P.: Text Entry by Eye Gaze. PhD Thesis, Dissertations in Interactive Technology, Number 11, Department of Computer Sciences. University of Tampere (2009)
20. Majaranta, P., Ahola, U.-K., Špakov, O.: Fast Gaze Typing with an Adjustable Dwell Time. In: Proc. 27th International Conference on Human Factors in Computing Systems (CHI), pp. 357–360. ACM Press, New York (2009)
21. Majaranta, P., MacKenzie, I.S., Aula, A., Räihä, K.-J.: Effects of Feedback and Dwell Time on Eye Typing Speed and Accuracy. Universal Access in the Information Society 5, 199–208 (2006)
22. Majaranta, P., Räihä, K.-J.: Text Entry by Gaze: Utilizing Eye Tracking. In: [17], ch. 9, pp. 175–187 (2007)
23. McWhirter, N. (ed.): The Guinness Book of World Records, 23rd US edn. Sterling, New York (1985)
24. Nielsen, J.: F-Shaped Pattern for Reading Web Content. Alertbox (2006), http://www.useit.com/alertbox/reading_pattern.html
25. Pomplun, M., Ritter, H., Velichkovsky, B.M.: Disambiguating Complex Visual Information: Towards Communication of Personal Views of a Scene. Perception 25, 931–948 (1996)
26. Räihä, K.-J., Aula, A., Majaranta, P., Rantala, H., Koivunen, K.: Static Visualization of Temporal Eye-Tracking Data. In: Costabile, M.F., Paternò, F. (eds.) INTERACT 2005. LNCS, vol. 3585, pp. 946–949. Springer, Heidelberg (2005)
27. Räihä, K.-J., Ukkonen, E.: Minimizing the Number of Evaluation Passes for Attribute Grammars. SIAM Journal on Computing 10, 772–786 (1981)
28. Räihä, K.-J., Ukkonen, E.: The Shortest Common Supersequence Problem over Binary Alphabet is NP-complete. Theoretical Computer Science 16, 187–198 (1981)
29. Santella, A., DeCarlo, D.: Robust Clustering of Eye Movement Recordings for Quantification of Visual Interest. In: Proc. Symposium on Eye Tracking Research & Applications (ETRA), pp. 27–34. ACM Press, New York (2004)
30. Siirtola, H., Laivo, T., Heimonen, T., Räihä, K.-J.: Visual Perception of Parallel Coordinate Visualizations. In: Proc. 13th International Conference on Information Visualisation (IV), pp. 3–9. IEEE Press, New York (2009)
31. Silfverberg, M.: Historical Overview of Consumer Text Entry Technologies. In: [17], ch. 1, pp. 3–25 (2007)
32. Soukoreff, R.W., MacKenzie, I.S.: Measuring Errors in Text Entry Tasks: An Application of the Levenshtein String Distance Statistic. In: Extended Abstracts of the ACM Conference on Human Factors in Computing System (CHI), pp. 319–320. ACM Press, New York (2001)
33. Špakov, O.: iComponent – Device-Independent Platform for Analyzing Eye Movement Data and Developing Eye-Based Applications. PhD Thesis, Dissertations in Interactive Technology, Number 9, Department of Computer Sciences, University of Tampere (2008)
34. Torstling, A.: The Mean Gaze Path: Information Reduction and Non-Intrusive Attention Detection for Eye Tracking. M.Sc. thesis, Report XR-EE-SB 2007:008, The Royal Institute of Technology, Stockholm (2007)
35. Tuisku, O., Majaranta, P., Isokoski, P., Räihä, K.-J.: Now Dasher! Dash Away! Longitudinal Study of Fast Text Entry by Eye Gaze. In: Proc. Symposium on Eye Tracking Research & Applications (ETRA), pp. 19–26. ACM Press, New York (2008)

36. Ukkonen, A.: Visualizing Sets of Partial Rankings. In: Berthold, M.R., Shawe-Taylor, J., Lavrač, N. (eds.) IDA 2007. LNCS, vol. 4723, pp. 240–251. Springer, Heidelberg (2007)
37. Ukkonen, E.: Algorithms for Approximate String Matching. Information and Control 64, 100–118 (1985)
38. Ward, D.J., Blackwell, A.F., MacKay, D.J.C.: Dasher: A Data Entry Interface Using Continuous Gestures and Language Models. In: Proc. ACM Symposium on User Interface Software and Technology (UIST), pp. 129–137. ACM Press, New York (2000)
39. Ward, D.J., MacKay, D.J.C.: Fast Hands-Free Writing by Gaze Direction. Nature 418, 838 (2002)
40. West, J.M., Haake, A.R., Rozanski, E.P., Karn, K.S.: eyePatterns: Software for Identifying Patterns and Similarities Across Fixation Sequences. In: Proc. Symposium on Eye Tracking Research & Applications (ETRA), pp. 149–154. ACM Press, New York (2006)
41. Wobbrock, J.O.: Measures of Text Entry Performance. In: [17], ch. 3, pp. 47–74 (2007)
42. Wooding, D.S.: Fixation Maps: Quantifying Eye-Movement Traces. In: Proc. Symposium on Eye Tracking Research & Applications (ETRA), pp. 31–36. ACM Press, New York (2002)
43. Yarbus, A.L.: Eye Movements and Vision. Plenum Press, New York (1967)

# Approximate String Matching with Reduced Alphabet*

Leena Salmela[1] and Jorma Tarhio[2]

[1] University of Helsinki, Department of Computer Science
leena.salmela@cs.helsinki.fi
[2] Aalto University
Deptartment of Computer Science and Engineering
jorma.tarhio@tkk.fi

**Abstract.** We present a method to speed up approximate string matching by mapping the factual alphabet to a smaller alphabet. We apply the alphabet reduction scheme to a tuned version of the approximate Boyer–Moore algorithm utilizing the Four-Russians technique. Our experiments show that the alphabet reduction makes the algorithm faster. Especially in the $k$-mismatch case, the new variation is faster than earlier algorithms for English data with small values of $k$.

## 1 Introduction

The approximate string matching problem is defined as follows. We have a pattern $P[1...m]$ of $m$ characters drawn from an alphabet $\Sigma$ of size $\sigma$, a text $T[1...n]$ of $n$ characters over the same alphabet, and an integer $k$. We need to find all such positions $i$ of the text that the distance between the pattern and a substring of the text ending at that position is at most $k$. In the $k$-difference problem the distance between two strings is the standard edit distance where substitutions, deletions, and insertions are allowed. The $k$-mismatch problem is a more restricted one using the Hamming distance where only substitutions are allowed.

Among the most cited papers on approximate string matching are the classical articles [1,2] by Esko Ukkonen. Besides them he has studied this topic extensively [3,4,5,6,7,8,9,10,11]. In this paper we present a practical improvement for Esko Ukkonen's approximate Boyer–Moore algorithm (ABM) [7] developed together with J. Tarhio. The ABM algorithm is based on Horspool's variation [12] of the Boyer–Moore algorithm [13]. ABM has variations both for the $k$-difference problem and for the $k$-mismatch problem. The $k$-difference variation is a filtration method. Recently Salmela et al. [14] introduced a tuned version of ABM for small alphabets. Here we consider an alphabet reduction technique which makes the tuned ABM more practical in the case of large alphabets. Our approach reduces the preprocessing time and the space usage of the algorithm. Our experiments show that the new variation is faster than the original ABM. Moreover, the new

* Supported by Academy of Finland grants 118653 (ALGODAN) and 134287.

T. Elomaa et al. (Eds.): Ukkonen Festschrift, LNCS 6060, pp. 210–220, 2010.

variation is faster than earlier mismatch algorithms for English data with small values of $k$.

The rest of the paper is organized as follows. We start with a short review on alphabet transformations in string matching. After that we review earlier versions of ABM. Then we explain our alphabet transformations in detail. Before conclusions we review results of our experiments.

## 2 Alphabet Transformations in String Matching

Alphabet transformation is a widely used method to increase the efficiency of string matching. There are several types of alphabet transformations. One of the most common transformations is hashing [15]. One can check in a hash table, whether a window of the text possibly equals the pattern. By selecting a suitable hash function, one can control the probability of false matches [15]. In case of multiple patterns, one can apply binary search [16] or two-level hashing [17] for checking candidate matches.

In algorithms of Boyer–Moore type it is common to use $q$-grams, i.e. substrings of $q$ characters, instead of single characters in shift calculation. This technique was already mentioned in the classical paper by Boyer and Moore [13]. The aim is to increase the size of effective alphabet, which leads to longer shifts especially in the case of small alphabets. The approach extends to multidimensional [18,19] and parameterized matching [20]. In many algorithms [21,22], $q$-grams and hashing occur together in shift calculation.

It is common to use $q$-grams instead of single characters also for other purposes than shifting. The aim is to increase practical scanning speed [23,24,25] or to improve selectivity [3,4,26]. Grams are not always continuous, but they may be gapped [27] or equidistant [28].

Still another type is relaxed preprocessing with a reduced alphabet [29,30]. In approximate string matching this extends the applicability of the Four-Russians technique [31,32], which is used to precompute edit distances between arbitrary $q$-grams and the $q$-grams of a pattern. With a reduced alphabet one can apply a larger $q$ without extra space and preprocessing time. In Section 4, we will consider an application of transformations of this type in detail.

## 3 Tuned Version of ABM

In this section we will describe a tuned version of the ABM algorithm [7]. In the next section we will then use this algorithm to illustrate how to apply an alphabet reduction technique to speed up an approximate string matching algorithm that uses $q$-grams.

As preprocessing the ABM algorithm computes the shifts for each character of the alphabet as in the Boyer–Moore–Horspool algorithm [12]. During searching the shift is then computed by considering last $k + 1$ characters of the current window. The shift is the minimum of the precomputed shifts for each of those $k + 1$ characters. After shifting, at least one of these characters will be aligned

**Algorithm 1.** Search for $P[1...m]$ in $T[1...n]$ with at most $k$ errors

```
1: Preprocessing:
2: for all G ∈ Σ^q do
3:     for i ← 1 to m do
4:         D[i] ← the minimum number of errors for aligning G with P[1...i] when
                  deletions in the beginning of either G or P[1...i] are free
5:     M[G] ← D[m]
6:     D_s[G] ← m − max{i | i < m and D[i] ≤ k}
7: Searching:
8: j ← m − k {j ← m for the mismatch version of the algorithm}
9: while j ≤ n do
10:    G ← T[j − q + 1...j]
11:    if M[G] ≤ k then
12:        verify the potential match
13:    j ← j + D_s[G]
```

correctly with the pattern or the pattern is shifted past the first one of these characters.

Liu et al. [33] tuned the $k$-mismatch version of ABM for smaller alphabets. Their algorithm, called FAAST, uses a stronger shift function based on a variation of the Four-Russians technique [31,32] to speed up the search. Instead of minimizing $k+1$ shifts during search, it uses a precomputed shift table for a $q$-gram aligned with the end of the pattern, where $q \geq k+1$ is a parameter of the algorithm. (The original paper used the notation $(k+x)$-gram.) The shift table is calculated so that after the shift at least $q-k$ characters are aligned correctly or the window is shifted past the last $q$-gram of the previous window.

Salmela et al. [14] further refined the FAAST algorithm and adapted it also to the $k$-difference problem. Their algorithm stores the number of substitutions or differences for aligning each $q$-gram with the end of the pattern and uses this precomputed value in the searching phase instead of recomputing it. We will use Algorithms 1 and 2 from [14] as a basis for our algorithm with alphabet reduction. The first one is an algorithm for the $k$-mismatch problem and the second one solves the $k$-difference problem. The pseudo code of these algorithms is shown as Algorithm 1. The preprocessing of these algorithms takes $O(m\sigma^q)$ time[1] and the average complexity of searching is $O(n(\log_\sigma m+k)/m)$ when $q = \Theta(\log_\sigma m+k)$. In the average complexity of searching and when computing the value of $q$, $\sigma$ should be replaced by $1/p$, where $p$ is the probability of two random characters matching, if the alphabet is not uniform. The space complexities of the mismatch version and the difference version of the algorithm are $O(m\sigma^q + mq) = O(m\sigma^q)$ and $O(m\sigma^q + mq + m^2) = O(m\sigma^q)$, respectively. We see now that the naive approach of using an alphabet of size 256 for English text is not feasible as the space (and preprocessing) requirement of the algorithm grows exponentially when $q$ is increased. Even if we map each character to a unique integer, the alphabet size is too large to be practical for larger values of $q$.

[1] See [14] for details on how to implement the preprocessing phase to reach this bound.

## 4 Algorithm with Alphabet Reduction

We are now ready to present an alphabet reduction scheme for approximate string matching. The scheme can be applied to any algorithm using $q$-grams. As an example, we apply it to the tuned version of the ABM algorithm presented in the previous section. A similar alphabet reduction scheme has been earlier presented by Fredriksson and Navarro [30], but their alphabet mapping is different from ours.

We will first present the algorithm with alphabet reduction assuming we have a mapping function $g : \Sigma \mapsto \hat{\Sigma}$ which maps each character of the alphabet to a character in the reduced alphabet $\hat{\Sigma}$ of size $\hat{\sigma}$. We first note that if a pattern has an (approximate) occurrence in a text, then the pattern that is mapped to the reduced alphabet has the same (approximate) occurrence in a text that is also mapped to the reduced alphabet. However, the mapped pattern might also have additional (approximate) occurrences in the mapped text.

Instead of mapping the whole text to the reduced alphabet, we will use the following method which only maps the needed $q$-grams of the text to the reduced alphabet. The preprocessing phase will now operate with the reduced alphabet. That is, we map each character of the pattern to the reduced alphabet and compute the arrays $M$ and $D_s$ for all $q$-grams in the reduced alphabet. The searching phase uses the same mapping of the $q$-grams of the text when accessing the arrays $M$ and $D_s$ but the verification of a potential match is performed with the original alphabet. The time complexity of the preprocessing phase is reduced to $O(m\hat{\sigma}^q)$ and the average complexity of searching becomes $O(n(\log_{\hat{\sigma}} m+k)/m)$ when $q = \Theta(\log_{\hat{\sigma}} m + k)$. The space complexities of the mismatch and difference versions of the algorithm are also reduced to $O(m\hat{\sigma}^q + mq) = O(m\hat{\sigma}^q)$ and $O(m\hat{\sigma}^q + mq + m^2) = O(m\hat{\sigma}^q)$, respectively. We now see that if we map the English alphabet for example to a reduced alphabet of size 8, using much larger values of $q$ becomes feasible.

We notice that the average complexity of searching in the reduced alphabet scheme is theoretically larger than in the plain algorithm. However, our experiments show that in practise searching is faster in the reduced alphabet scheme. First we note that if the alphabet is nonuniform, some characters may be very rare and without alphabet reduction these characters increase the $q$-gram space unnecessarily as they are rarely accessed and thus do not improve filtering noticeably. Another issue is that $q$ must be an integer and therefore we might have to make compromises when choosing the value of $q$ as the optimal $q$ is $c(\log_\sigma m+k)$ for some constant $c$ and this optimal $q$ might not be an integer. This problem is emphasized when the alphabet is large as there are fewer feasible choices for the value of $q$. Furthermore if $k$ is large, even choosing $q = k+1$, which is the minimum possible value for $q$ in the algorithm, might be infeasible with the original alphabet and then reducing the alphabet size can make it feasible to use a large enough $q$.

The remaining problem is to find the mapping function $g$ given the size of the reduced alphabet. The mapping function should minimize the probability that two random characters match. This probability is minimized by a mapping that

produces the most uniform reduced alphabet [30]. Fredriksson and Navarro [30][2] have earlier used the following method to find this mapping. They first sort the characters in ascending order of frequency. The $i$:th character in this order is then mapped to the ($i$ mod $\hat{\sigma}$):th character in the reduced alphabet.

The problem of finding the mapping is defined formally as follows. We are given the frequency $f_c$ of each character $c \in \Sigma$ and an integer $\hat{\sigma}$ that defines the size of the reduced alphabet. Our task is now to partition the characters in $\Sigma$ into $\hat{\sigma}$ subsets $S_i$ such that the following objective function is minimized:

$$\max_{i \in [1,\hat{\sigma}]} \left\{ \sum_{c \in S_i} f_c \right\} - \min_{i \in [1,\hat{\sigma}]} \left\{ \sum_{c \in S_i} f_c \right\}.$$

Each of the subsets $S_i$ is then mapped to a unique character in $\hat{\Sigma}$. This formulation is equivalent to the $\hat{\sigma}$-way number partitioning problem.

The number partitioning problem has been shown to be NP-complete [34] and thus we resort to the following well known greedy algorithm to find the mapping function. We first sort the characters in decreasing order of frequency. Starting from the most frequent character we map that character to the least frequent character of the reduced alphabet.

We considered the following schemes for reducing the alphabet:

**Pattern Alphabet.** The reduced alphabet consists of all characters that occur in the pattern and one extra character [29]. All characters that do not occur in the pattern are mapped to this extra character.

**Reduced Alphabet.** We compute the reduced alphabet using the greedy algorithm outlined above.

**Reduced Pattern Alphabet.** We combine the two above methods. We first form an alphabet consisting of the characters occurring in the pattern and an extra character as in the pattern alphabet method. Then we use the reduced alphabet method to reduce this alphabet.

We also tried first classifying the characters into separators and letters or separators, vowels, and consonants, and then applying alphabet reduction to these groups separately but this approach was not competitive.

## 5 Experimental Results

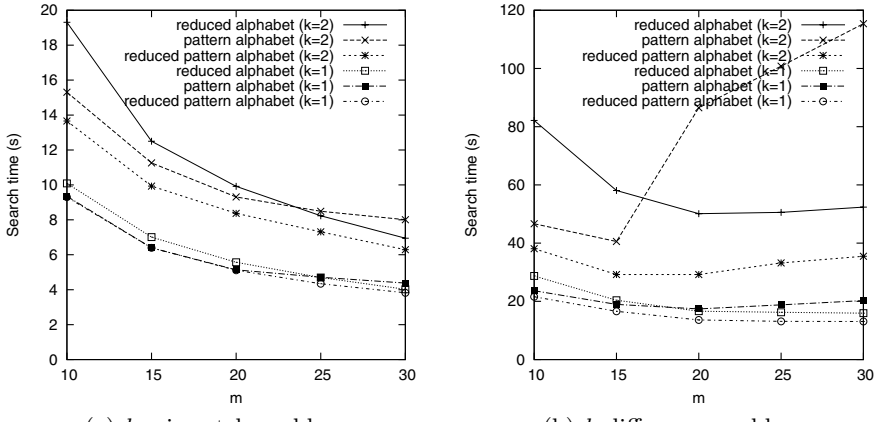
Experiments were run on an Intel 3.16 GHz dual core CPU with 3.7 GB of memory and 32 kB L1 cache and 6144 kB L2 cache. The computer was running Linux 2.6.27. The algorithms were written in C and compiled with the gcc 4.3.2 compiler producing 32-bit code. We used the 50 MB English text from the *PizzaChili* site, `http://pizzachili.dcc.uchile.cl`. Each pattern set consists of 200 different patterns of the same length. The patterns are randomly chosen

[2] This method is not outlined in the paper but can be found in the corresponding code.

from the text and a substitution, insertion, or deletion is introduced at every position with probability 0.05. As an example, the 200 patterns of length 20 have 261 total matches when searching allowing one substitution and a total of 680 matches when allowing one substitution, deletion, or insertion.

### 5.1 Comparison of Alphabet Reduction Techniques

The value of $q$ was varied from 2 to 7, and we tried reduced alphabet sizes of 4, 8, 16, and 32. We show the results for the best observed values of these parameters. Figure 1 shows the searching times excluding the preprocessing time for the $k$-mismatch and $k$-difference problems for $k = 1$ and $k = 2$. Table 1 shows the best parameter values for each of the methods. As can be seen, the best method in all cases is the reduced pattern alphabet method.




(a) $k$-mismatch problem (b) $k$-difference problem

**Fig. 1.** Search times

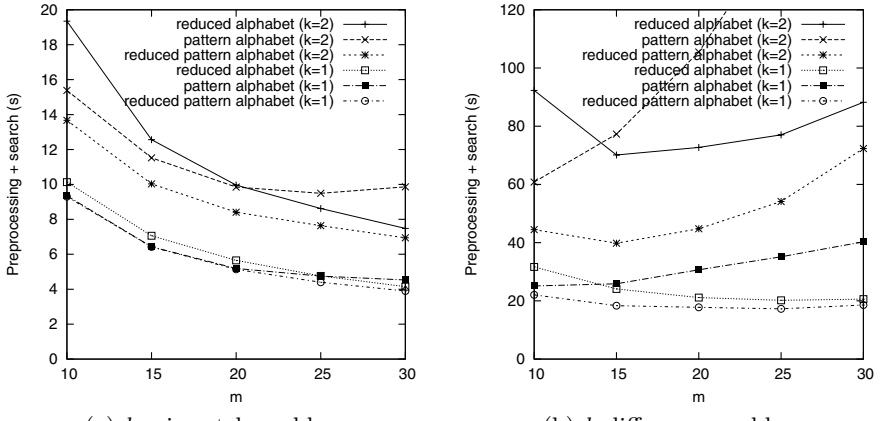
Figure 2 shows the combined preprocessing and search times. Again we see that the reduced pattern alphabet method is the best. Furthermore, comparing Figures 1(a) and 2(a) we note that in the $k$-mismatch problem the preprocessing time is much smaller than the searching time. In the $k$-difference problem, especially the pattern alphabet method has a high preprocessing cost but also the other methods have a moderately high preprocessing cost when $k = 2$.

The increase in the preprocessing time is due to using a larger value of $q$ to speed up searching. Figure 3 shows how the preprocessing time grows when $q$ is increased in the $k$-difference algorithms when $m = 10$ and $k = 1$.

We also ran similar experiments with the 50 MB protein text from the *PizzaChili* site, `http://pizzachili.dcc.uchile.cl`. The results with protein data were very similar to our results with English text.

**Table 1.** The parameters yielding the best search time for each of the methods and the corresponding number of different $q$-grams in the reduced alphabet. If different values were best for different pattern lengths, the alternative values are shown on subsequent rows. To compute the number of different $q$-grams in the pattern alphabet method, we use the average reduced alphabet size for pattern length 30 which was 17.21. Note that the reduced pattern alphabet method is equivalent to the pattern alphabet method for the 1-mismatch problem because the size of the reduced alphabet is larger than the length of the pattern.

| Algorithm | 1-mismatch | | 2-mismatch | | 1-difference | | 2-difference | |
|---|---|---|---|---|---|---|---|---|
| | Param. | $\hat{\sigma}^q$ | Param. | $\hat{\sigma}^q$ | Param. | $\hat{\sigma}^q$ | Param. | $\hat{\sigma}^q$ |
| Pattern Alphabet | $q = 3$ | $< 2^{13}$ | $q = 4$ | $< 2^{17}$ | $q = 5$ | $< 2^{21}$ | $q = 6$ | $< 2^{25}$ |
| | | | | | | | $q = 5$ | $< 2^{21}$ |
| Reduced Alphabet | $\hat{\sigma} = 16, q = 3$ | $2^{12}$ | $\hat{\sigma} = 4, q = 6$ | $2^{12}$ | $\hat{\sigma} = 8, q = 6$ | $2^{18}$ | $\hat{\sigma} = 8, q = 7$ | $2^{21}$ |
| | $\hat{\sigma} = 8, q = 4$ | $2^{12}$ | $\hat{\sigma} = 8, q = 5$ | $2^{15}$ | $\hat{\sigma} = 4, q = 8$ | $2^{16}$ | | |
| Reduced Pattern Alphabet | $\hat{\sigma} = 32, q = 3$ | $2^{15}$ | $\hat{\sigma} = 8, q = 4$ | $2^{12}$ | $\hat{\sigma} = 8, q = 6$ | $2^{18}$ | $\hat{\sigma} = 8, q = 7$ | $2^{21}$ |
| | | | $\hat{\sigma} = 8, q = 5$ | $2^{15}$ | | | | |




(a) $k$-mismatch problem (b) $k$-difference problem

**Fig. 2.** Combined preprocessing and search times for a text of length 50 MB

### 5.2 Comparison with Other Algorithms

We compared the performance of the best alphabet reduction scheme, reduced pattern alphabet, with the following algorithms:

- ABM: The original ABM algorithm.
- BYP: The algorithm by Baeza-Yates and Perleberg [35] divides the pattern into smaller pieces so that if the pattern occurs at some position, at least one of the pieces must have an exact occurrence at that position. The algorithm then searches for exact matches of the pieces and verifies the occurrences found by the exact search.

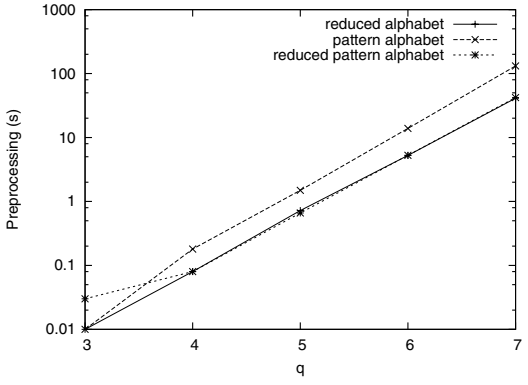
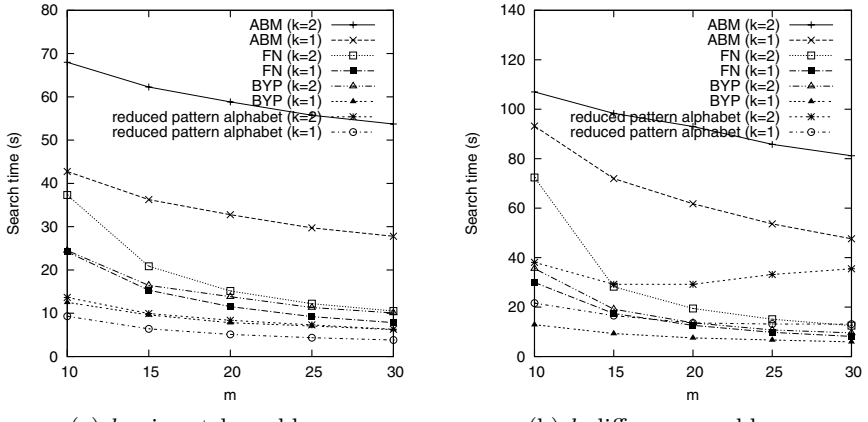



**Fig. 3.** Preprocessing time of the $k$-difference algorithm for varying values of $q$ when $m = 10$ and $k = 1$. Reduced alphabet and reduced pattern alphabet methods use reduced alphabet size 8.




(a) $k$-mismatch problem (b) $k$-difference problem

**Fig. 4.** Comparison of the best alphabet reduction scheme, reduced pattern alphabet, ABM, BYP, and FN

– FN: The algorithm by Fredriksson and Navarro [30] reads non-overlapping $q$-grams ($\ell$-grams in the original paper) in an alignment and with the help of preprocessed tables determines the minimum number of substitutions or differences for aligning the $q$-grams with the pattern in some way. When the minimum number of substitutions or differences exceeds $k$, the pattern is shifted so that the first of these $q$-grams is no longer aligned with the pattern. The potential matches must be verified. To make the comparison

fair, we modified their algorithm so that it uses the reduced pattern alphabet method, which improved its performance although the improvement was not as clear as in our algorithm.

Figure 4 shows the searching times of the algorithms. As can be seen, the alphabet reduction technique combined with the use of $q$-grams makes the new algorithm significantly faster than the plain ABM algorithm. In the $k$-mismatch case, our new algorithm is the fastest, BYP being the second fastest, FN third, and ABM clearly the slowest. In the $k$-difference case BYP takes the lead which was also the case in the experiments by Fredriksson and Navarro [30]. Overall FN is the second best although our algorithm is faster for short patterns which are important in practise. Again ABM is clearly the slowest.

## 6 Conclusions

We have presented an alphabet reduction technique to speed up algorithms for approximate string matching. We applied the technique to a Boyer–Moore style algorithm which uses the Four-Russians technique to compute shifts with small alphabets. When improved with alphabet reduction, the algorithm performs surprisingly well on large alphabets too. The space usage of the Four-Russians approach used in the algorithm is not feasible if the whole large alphabet is used but becomes practical with alphabet reduction. Our experiments on English data show that the algorithm with alphabet reduction is the fastest algorithm in the $k$-mismatch problem for small values of $k$.

**Acknowledgements.** We thank the referee, who helped us to improve this paper.

## References

1. Ukkonen, E.: Algorithms for approximate string matching. Information and Control 64(1-3), 100–118 (1985)
2. Ukkonen, E.: Finding approximate patterns in strings. J. Algorithms 6(1), 132–137 (1985)
3. Jokinen, P., Ukkonen, E.: Two algorithms for approximate string matching in static texts. In: Tarlecki, A. (ed.) MFCS 1991. LNCS, vol. 520, pp. 240–248. Springer, Heidelberg (1991)
4. Ukkonen, E.: Approximate string matching with q-grams and maximal matches. Theor. Comput. Sci. 92(1), 191–211 (1992)
5. Ukkonen, E.: Approximate string-matching over suffix trees. In: Apostolico, A., Crochemore, M., Galil, Z., Manber, U. (eds.) CPM 1993. LNCS, vol. 684, pp. 228–242. Springer, Heidelberg (1993)
6. Ukkonen, E., Wood, D.: Approximate string matching with suffix automata. Algorithmica 10(5), 353–364 (1993)
7. Tarhio, J., Ukkonen, E.: Approximate Boyer–Moore string matching. SIAM J. Comput. 22(2), 243–260 (1993)

8. Jokinen, P., Tarhio, J., Ukkonen, E.: A comparison of approximate string matching algorithms. Software–Pract. Exp. 26(12), 1439–1458 (1996)
9. Fredriksson, K., Navarro, G., Ukkonen, E.: Optimal exact and fast approximate two dimensional pattern matching allowing rotations. In: Apostolico, A., Takeda, M. (eds.) CPM 2002. LNCS, vol. 2373, pp. 235–248. Springer, Heidelberg (2002)
10. Kärkkäinen, J., Navarro, G., Ukkonen, E.: Approximate string matching on Ziv–Lempel compressed text. J. Discrete Algorithms 1(3-4), 313–338 (2003)
11. Mäkinen, V., Ukkonen, E., Navarro, G.: Approximate matching of run-length compressed strings. Algorithmica 35(4), 347–369 (2003)
12. Horspool, R.N.: Practical fast searching in strings. Software–Pract. Exp. 10(6), 501–506 (1980)
13. Boyer, R.S., Moore, J.S.: A fast string searching algorithm. Commun. ACM 20(10), 762–772 (1977)
14. Salmela, L., Tarhio, J., Kalsi, P.: Approximate Boyer–Moore string matching for small alphabets. Algorithmica (in press)
15. Karp, R.M., Rabin, M.O.: Efficient randomized pattern-matching algorithms. IBM J. Research and Development 31(2), 249–260 (1987)
16. Zhu, R., Takaoka, T.: A technique for two-dimensional pattern matching. Commun. ACM 32(9), 1110–1120 (1989)
17. Muth, R., Manber, U.: Approximate multiple strings search. In: Hirschberg, D.S., Meyers, G. (eds.) CPM 1996. LNCS, vol. 1075, pp. 75–86. Springer, Heidelberg (1996)
18. Kärkkäinen, J., Ukkonen, E.: Two- and higher-dimensional pattern matching in optimal expected time. SIAM J. Comput. 29(2), 571–589 (1999)
19. Tarhio, J.: A sublinear algorithm for two-dimensional string matching. Pattern Recogn. Lett. 17(8), 833–838 (1996)
20. Salmela, L., Tarhio, J.: Fast parameterized matching with q-grams. J. Discrete Algorithms 6(3), 408–419 (2008)
21. Lecroq, T.: Fast exact string matching algorithms. Inf. Process. Lett. 102(6), 229–235 (2007)
22. Wu, S., Manber, U.: A fast algorithm for multi-pattern searching. Technical report, Dept. of Computer Science, U. of Arizona (1994)
23. Kim, S.: A new string-pattern matching algorithm using partitioning and hashing efficiently. J. Exp. Algorithmics 4(2) (1999)
24. Fredriksson, K.: Shift-or string matching with super-alphabets. Inf. Process. Lett. 87(4), 201–204 (2003)
25. Ďurian, B., Holub, J., Peltola, H., Tarhio, J.: Tuning BNDM with q-grams. In: Proc. ALENEX 2009, pp. 29–37. SIAM, Philadelphia (2009)
26. Salmela, L., Tarhio, J., Kytöjoki, J.: Multipattern string matching with q-grams. J. Exp. Algorithmics 11 (2006)
27. Fontaine, M., Burkhardt, S., Kärkkäinen, J.: BDD-based analysis of gapped q-gram filters. Int. J. Found. Comput. Sci. 16(6), 1121–1134 (2005)
28. Fredriksson, K., Grabowski, S.: Practical and optimal string matching. In: Consens, M.P., Navarro, G. (eds.) SPIRE 2005. LNCS, vol. 3772, pp. 376–387. Springer, Heidelberg (2005)
29. Berry, T., Ravindran, S.: Tuning the Zhu–Takaoka string matching algorithm and experimental results. Kybernetika 38(1), 67–80 (2002)
30. Fredriksson, K., Navarro, G.: Average-optimal single and multiple approximate string matching. J. Exp. Algorithmics 9 (2004)

31. Arlazarov, V.L., Dinic, E.A., Kronrod, M.A., Faradzev, I.A.: On economic construction of the transitive closure of a directed graph. Doklady Academi Nauk SSSR 194, 487–488 (in Russian); English translation in Soviet Mathematics Doklady 11, 1209–1210 (1970)
32. Masek, W.J., Paterson, M.S.: A faster algorithm for computing string edit distances. J. Comput. Syst. Sci. 20(1), 18–31 (1980)
33. Liu, Z., Chen, X., Borneman, J., Jiang, T.: A fast algorithm for approximate string matching on gene sequences. In: Apostolico, A., Crochemore, M., Park, K. (eds.) CPM 2005. LNCS, vol. 3537, pp. 79–90. Springer, Heidelberg (2005)
34. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. Freeman, New York (1979)
35. Baeza-Yates, R., Perleberg, C.: Fast and practical approximate string matching. Inf. Process. Lett. 59(1), 21–27 (1996)

# ICT4D: A Computer Science Perspective

Erkki Sutinen and Matti Tedre

University of Eastern Finland
P.O. Box 111, 80101 Joensuu
firstname.lastname@uef.fi

**Abstract.** The term ICT4D refers to the opportunities of Information and Communication Technology (ICT) as an agent of development. Research in that field is often focused on evaluating the feasibility of existing technologies, mostly of Western or Far East Asian origin, in the context of developing regions. A computer science perspective is complementary to that agenda. The computer science perspective focuses on exploring the resources, or inputs, of a particular context and on basing the design of a technical intervention on the available resources, so that the output makes a difference in the development context. The modus operandi of computer science, construction, interacts with evaluation and exploration practices. An analysis of a contextualized information technology curriculum of Tumaini University in southern Tanzania shows the potential of the computer science perspective for designing meaningful information and communication technology for a developing region.

**Keywords:** ICT4D, ICT, Computer Science, Information Technology, Developing Countries.

## 1 Introduction

Computer science as a discipline emphasizes *construction* as a major research approach. For example, in algorithm research, even though a formal analysis of a computational method is important for understanding the behavior and competitiveness of the method, the key invention lies in the *design* of the method. This, naturally, requires a meaningful representation of the problem. In addition to theoretical computer science, the constructive approach is common in other branches of the computing discipline, too, including programming languages, data management, and parallel computing. Also in the human-oriented research areas of computing, such as human-centered design, the constructive tradition calls for designing novel gadgets for intuitive human-computer interaction.

The emphasis of construction in computer science is not surprising, taken into account the disciplinary forefathers of computer science: mathematics, engineering, and natural (or empirical) sciences [19]. From the viewpoint of mathematics, George Pólya in his classic book "How to Solve It?" [13] emphasized the experimental approach in a mathematical theory building process. From the engineering viewpoint, the constructive interest is the cornerstone of engineering

T. Elomaa et al. (Eds.): Ukkonen Festschrift, LNCS 6060, pp. 221–231, 2010.

disciplines [23], [4]. From the science viewpoint, the constructive nature of natural sciences is evident in the fact that more often than not, advances in modern natural science are tied to advances in instruments. For instance, development of particle physics is tied to development of particle detectors [12] and the development of astronomy relies heavily on development of telescopes and other instruments. For any human construction—be it abstract logico-mathematical construction, tangible construction of tools, or co-construction of experiment and theory—experimentation and improvisation are natural parts of the process.

In the construction process of many of its key terms and concepts, computer science has successfully exploited cultural patterns of the surrounding society. Elementary data structures use vocabulary and ideas from society, life, and nature. Take, for instance, concepts such as parents, relatives, agents, viruses; worms and bugs; as well as sleeping, killing, living, and dying (processes). Numerous algorithms and approaches are named after common human strategies. Take, for instance, divide and conquer, greedy algorithm, Trojan horse, and backdoor. Even the core ideas of several classic computational problems are easily accessible to most people living in a Western society—take, for instance, the dining philosophers' problem and the traveling salesman's problem. In fact, the history of the whole discipline can be understood from a socio-cultural perspective [14]. In his book "African Fractals" Ron Eglash has analyzed the artistic and architectural patterns of West African cultures and shown the relation of those patterns to self-iterative or recursive computation [8].

ICT4D, short for information and communication technology for development, is an established term that denotes a novel interdisciplinary research field, which explores the impact of ICT in various development challenges. The term *development* has been used in a wide variety of meanings, but often, as in this chapter, the term *development* is used as a normative term that refers loosely to improvement of people's situations in developing countries or regions. Similarly, the term *context* is a broad and ambiguous term, but we use the term *context* to refer to a particular situation in its geographical and socio-economic surroundings, like a Tanzanian village, a remote eco-tourism farm in North-Eastern Finland, or a South African township school. Any developing context has a set of challenges; a challenge refers to a possibly complex and hard-to-define combination of problems, needs, and resources. For example, a management challenge of a primary school in Soweto might consist of problems (such as too large classes and too few teachers), unfulfilled needs that students and teachers perceive (such as a need for textbooks and electricity), and resources (such as knowledge-hungry students). It is easy to get convinced that conventional European school management software is not an effective tool to address this particular management challenge.

ICT4D has not attracted the interest of the mainstream of computer science research. In that regard it shares a marginal disciplinary position with several other application areas of computer science, such as educational technology. The reasons might be twofold. On one hand, ICT4D research has been considered to belong to the area of social sciences, and included under, for instance, human

geography. This has led ICT4D researchers to evaluate existing technologies rather than design new ones. On the other hand, from the first sight the problems of using ICT in a developing context do not seem computationally challenging to attract a constructive computer scientist. Pictures from a South African township with masses of young children chatting with each other on their modest mobile phones do not easily raise the imagination or curiosity of a serious computer scientist. However, the research institutes of ICT industry have been more active; examples include, for instance, Nokia Research Africa.

## 2 Traditional ICT4D: Social Sciences Perspective for Evaluating ICT in Developing Contexts

ICT4D is a rich and diverse, multidisciplinary undertaking with multiple research aims (e.g.,[5], [21]), but for the purposes of our analysis we divide ICT4D research into four categories, according to whether the development challenge at hand is well-defined or well known, and according to whether technical solutions for the problem exist readily or not. We acknowledge that this rough division ignores various types of ICT4D research, such as diffusion research (see, e.g., [7]). We also acknowledge the fact that for many development challenges the most effective solutions are not technical solutions, and that we are often better off without new technological interventions. But for demonstrating the need for a computer science perspective in ICT4D research, our division is well suited. The two divisions above dissect ICT4D to four categories as follows (see also Table 1):

1. There is a body of ICT4D research that focuses on pinpointing and understanding social or economic needs or issues, and that attempts to *match* an existing technical solution—preferably an affordable one—with a specific need or issue. The inverse variant of this research type, though rarely explicitly acknowledged, is when researchers have a tool or innovation and they focus on finding a problem that their tool can solve. The matching-type of research, which is a sort of puzzle-solving activity, is largely aimed at seeking problems that current mainstream technical solutions can solve.
2. There is also a body of ICT4D research that focuses on *evaluating* to what extent an existing technical solution meets some certain, well-known needs, or how well an existing technical solution solves some specific socio-economic challenge. On a more general level, evaluative research explores and measures the changes that introduction of new tools bring forth. That type of evaluative research is usually aimed at understanding the dynamics, restrictions, and ramifications of existing technical tools in context of one specific issue (e.g., [11]). Often the outcomes of this type of research are qualitative and quantitative reports of uses of technology.
3. There is some *exploratory* research on possibilities of technology in developing countries, where the problem space is not well known and where one of the goals is to explore the problem space in order to delimit the boundaries of problems and open up new areas for investigation. This type of research often works as groundwork for further research of the two previous types.

4. Finally, there is some research on ICT4D topics where the need, problem, or even potential resources are well known but there does not yet exist a technology to meet the need, solve the problem, or release the potential. Often it is also unknown whether technology can contribute to the situation at all. In this type of research, the issue may be well known and even well understood, but there are no suitable technical tools available for the job, so the task of the researcher is to *construct* (define, design, implement, and test) a tool for tackling the issue at hand. This is the home ground for computer science oriented ICT4D research.

**Table 1.** Division of research approaches in ICT4D

| | ***ICT: Existing*** | ***ICT: Not Existing*** |
|---|---|---|
| ***Development challenge: Not known*** | 1) Matching a tool and a problem | 3) Exploratory research |
| ***Development challenge: Known*** | 2) Evaluation research | 4) Constructive research |

Many larger research studies involve several categories of Table 1; for instance, finding an existing tool to solve an issue is often followed by an evaluation of the solution. The categories of Table 1 are not specific to any academic discipline, although certain disciplines may emphasize some types of research over others. For instance, exploratory research is not only a part of social studies and humanities, but also of computer science in general. Peter Fletcher argued that much of research in computing is not of the sort "[seek] *the best solution to a previously specified problem*" [9]. He wrote that often computer scientists work with problems that are poorly understood, and with which one major goal is to understand the problem and delimit it more precisely. Fletcher argued that, "*Computer science could not consist entirely of* [solving precisely specified problems], *since someone has to think up the precisely specified problems in the first place and convince us that they are worth pursuing.*"

## 3 ICT4D Reborn: Computer Science Perspective for Designing Meaningful ICT

In spite of opposite impressions, evaluation-oriented research in ICT4D (e.g., [11]) can be considered to be technology-driven, for that type of research is driven by the technical tools at hand. Evaluative research is often technology-dependent in a sense that in many cases those research studies take technical equipment as constant, unchangeable givens. Insofar as some ICT4D researchers, whose backgrounds are in social sciences or humanities, might not be deeply aware of the opportunities of technology, those researchers can only use existing tools for tackling development issues. Largely, this dynamics leads to a "one size

fits all" kind of technology-dependence in many contemporary ICT4D research and development projects.

However, research and design of technology that supports development in a particular context does not need to follow the methodological agenda or methodological dogmas of social sciences and humanities. The constructive interest of computer science provides the ICT4D research area with a complementary contribution: design of *new* technical innovations. This leads away from technology-dependence towards technology independence, or at least technological agnosticism. Technology independence refers to an atelier type of approach where a computer scientist, like an artist in a working room, works as a master of a variety of tools for the best possible artifact.

Thus, a technical or constructive orientation towards ICT4D does not need to follow a technocratic orientation where the design team or project team follows a predefined technical plan with an equally pre-given set of tools to implement a specific task. Quite the contrary: a computer science orientation is needed for curbing too much technoenthusiasm. If anyone, it is computer scientists who know the limits of automatic computing and ICT. In the same vein, computer scientists also know the possibilities of modern information and communication technology.

### 3.1 The Art of Assembling Artifacts Using Contextual Inputs

Computer scientists are expected to have the ability to understand, analyze and conceptualize a technical challenge; to identify the inputs that need to be used to address the challenge; to design a method that gives a computational solution for given inputs; and to evaluate the overall solution based on some context-specific criteria. In the area of ICT4D, this might be conceptualized as an *inputs-driven* approach. For this approach, we have identified four stages that are involved in the process of constructing ICT4D:

1. analyze the given context and pay attention to its potential resources that call for their realization or empowerment through technology (*contextualization*);
2. represent the resources as inputs for a computational artifact (*conceptualization*);
3. design and build an artifact that contributes to a holistic solution to the development challenge (*concretization*); and
4. evaluate the difference that the artifact makes, in terms of the challenge, for the local context (*consideration* and *contemplation*).

To give an example of the driven-by-inputs process, let us consider the need for health information systems—a vastly complex system if there ever was one. Those systems have multiple users of various kinds, they involve exceedingly complicated data structures (some of which are necessary in some cases but optional in others), they require various kinds of interfaces for different uses, they entail a range of computational requirements as well as types of output data,

they involve intricate communication and synchronization systems, and they are irrevocably intertwined with larger, very complex socioeconomic systems. Some of the challenges above are reducible to classical computational problems, whereas others are highly contextual and require new unorthodox approaches. Some challenges are puzzle-like well-constrained intellectual exercises, whereas others require a holistic understanding and analysis of the multidimensional sociotechnical system. That is, the system construction is driven by inputs and resources.

### 3.2 Towards a Functional ICT4D Design

The main aim of a computer science oriented ICT4D initiative, whether of an explorative type or a constructive type, is to come up with functional technology that contributes to changing conditions within a given context. The attribute *functional* refers to a situation where technology in a self-evident or intuitive and aesthetically appropriate way helps its user to realize a task of personal importance at hand, and gives him or her a functional user experience. Thus, functional is more than usable or useful; it is easy to come up with tools that are useable or useful but not functional. A functional artifact needs to promote both access and ownership. Access refers to values such as usefulness and usability; whereas ownership is based on contextual, cultural, and individual priorities. Thus, access can be measured by objective criteria, while ownership is always a more subjective issue.

Figure 1 illustrates two alternative design processes that start from an existing example, model, or theory, and that aim at a functional ICT artifact. The lower curve demonstrates a generalize-first process, which is motivated from an access perspective and, hence, generalizes an existing solution to meet the needs of the masses. Only after generalization the solution will be personalized, adapted, or tuned to match an identified user group. The problem of this process is that the generalization stage might result in an artifact that is too stripped-off to accommodate the particularities of a given context.

The upper curve in Figure 1 describes a concretize-first design process, which gives priority to the inputs of the particular context. Therefore, the first steps enrich rather than prune out features of the solution. It is also important to observe that the initial design phase aims at concretization rather than abstraction that is usually the result of any generalization step. Interestingly, with regard to abstract models of computability, due to his fascination with machinery Alan Turing came up with a nearly tangible, machine-like model of computability [20] whereas Church's theorization of computability was a more traditional, mathematical one [6] (see also Turing's biography [10]).

Both the generalize-first and concretize-first processes feature the explorative and constructive interests of computer science-based ICT4D research. However, explorative aspects dominate in the concretize-first approach, whereas the generalize-first agenda, due to its emphasis on access at the beginning of the process, favors construction as a solution to an at least partly known, generic development challenge.

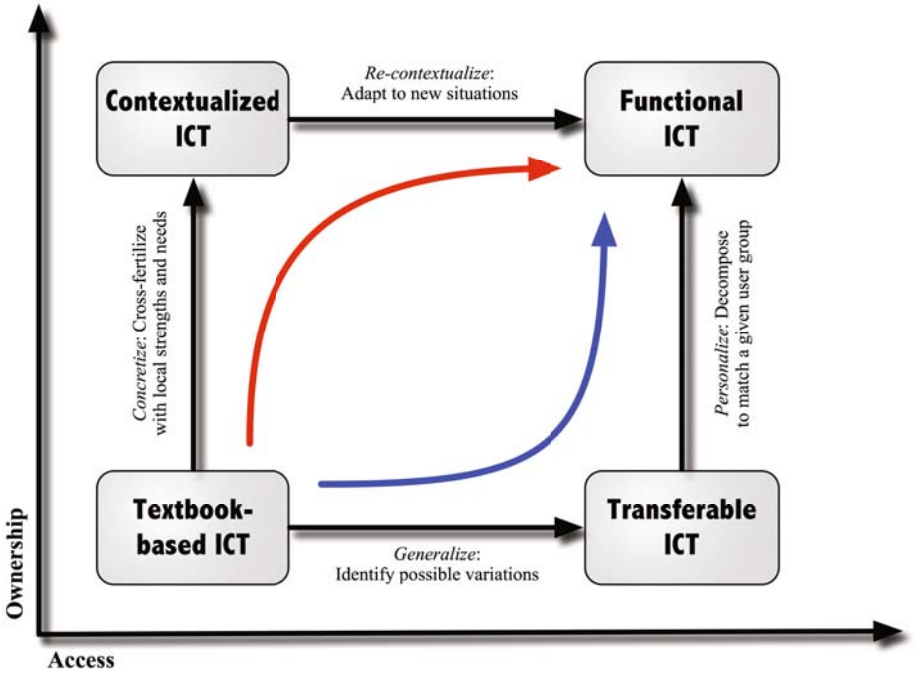



**Fig. 1.** Two alternative paths towards a functional ICT artifact

To characterize the two processes in above-mentioned terms Contextualize, Conceptualize, Concretize, and Consider, the processes differ from each other by the order in which the four C:s are applied. The generalize-first process starts from Conceptualize to guarantee the accessibility of the design for an unspecified, but broad target group. This requires also the Consider aspect as the basis for a feedback mechanism. The Concretize and Contextualize aspects follow in, largely, this order.

The concretize-first process proceeds through the Contextualize, Conceptualize and Concretize stages, in this order. The re-contextualization stage requires the Consider aspect.

In Figure 1, steps taken along the positive *Access* axis emphasize the product-oriented, needs/theory-based, deductive, or top-down aspects of the process. Steps along the positive *Ownership* axis represent creative, strength-based, empirical, inductive, and bottom-up perspectives.

## 4 Interplay of Computer Science and ICT4D

Development challenges offer a new thread to the formation process of computer science as a discipline. Since most of the contexts of development challenges are

outside the social, cultural, and economic boundaries of the traditional realm of ICT applications, the challenges call computer science to a cross-fertilization process. The process, if successful, can transform and renew computer science.

The contextual specifics of developing regions call for exciting new application areas of computer science. For instance, the rich legacy of storytelling can be a platform for novel ways to represent text. The importance of the social structures and belongingness in many parts of Africa can serve as a foundation for new types of social media. Conflicts in extremely heterogeneous environments call for technical assets to their resolution; they require not only understanding of a text but various connotations thereof. Indigenous knowledge systems can provide new ideas for knowledge management, or even data structures [1]. Ancient games still played offer new ideas for digital entertainment [8]. Various resource allocation or logistical systems are exceedingly complex due to exceptions, improvisations, varying weather and other nature conditions, and differences of communication habits among the multiple ethnic groups, or of literacy among the users of one system.

Table 2 summarizes the aspects of a computer science perspective to ICT4D when compared to that of other forms of scientific inquiry in the field of ICT4D. The aspects in the right column strongly indicate that the computer science approach promotes change in a given context. For example, *effectiveness* as a key quality focus requires that a designed ICT artifact not only improves current practices—by making them more efficient than before—but qualitatively changes them to extend beyond the earlier expectations, best practices, or quality assurance metrics. For example, educational challenges in Africa are so dire that streamlining primary schools with whatever connectivity for improved Internet access will change only little, because of the low ICT literacy of teachers. Instead, a flexible and mobile notebook with an adaptive reading interface—supporting a large variety of knowledge representations—might be helpful, if designed together with intended users.

Besides making clear the contribution of computer science to ICT4D, Table 2 illustrates the opportunities of ICT4D challenges to change computer science as a discipline. For example, the expectations that developing contexts set for ICT are an important reminder of the ethical task of computer science.

**Table 2.** ICT4D challenges to computer science

| | ***ICT4D*** | ***Computer Science*** |
|---|---|---|
| ***Research interest*** | Matching and evaluation | Exploration and construction |
| ***Role of technology*** | Technology-dependent | Technologically agnostic |
| ***Expectations from ICT*** | Conservation, replication, and increase of efficiency | Agent of change |
| ***Quality focus*** | Efficiency | Effectiveness |
| ***Driving motivations*** | Outputs, unsatisfied needs | Inputs, available resources |
| ***User experience*** | Useful, usable | Functional |

### 4.1 ICT4D as an Agent of Change in Computer Science Education

For example, at Tumaini University–Iringa University College, our Tanzanian colleagues and we have worked together to develop an IT curriculum that responds to the severe and manifold development issues in Iringa region in Tanzania (e.g., [3]). We started from the standard ACM/IEEE curricula and have over the years developed Tumaini's IT curriculum to resonate with the local needs, problems, and possibilities, as well as with local materials, resources, and competences [15]. The development process of the contextualized IT curriculum can be explained through the artifact creation model explained in Figure 1.

The environment—natural, social, economic, and technical environment—of developing countries brings about issues that require special attention in curriculum design. We have had to accommodate the generic ACM/IEEE "prototype" curricula of computing heavily to the Iringan context, where inadequate electrical and communications infrastructure cause hardware to malfunction, wear out, and break; a hostile natural environment causes problems with equipment and eventually destroys it; quirks of local manufacturing and procurement complicate acquisitions; counterfeit products, inexistent customer care, and lack of warranty make purchases risky; excessively complex customs and shipping procedures make foreign acquisitions painful; widespread problems with corruption make accounting tricky; and lack of qualified staff is a systemic problem [15]. None of the above issues can be found in computer science textbooks or IT textbooks, but they still pose a very real challenge for IT professionals in Tanzania. Without the ability to cope with issues such as the above, IT professionals' professionalism is compromised.

The IT profession in industrialized countries, such as in Finland, relies on scores of specialists who can tackle narrowly bounded topics well, and who in their work focus only on that topic. However, specialization-oriented systems rely on the availability of an army of specialists: there must be someone to fill each specialization area. In developing countries narrow specialization is not usually feasible, for a common situation is that there is not a single IT professional available within a fifty-kilometer radius.

In developing Tumaini University's B.Sc. Program in IT, our process has followed the concretize-first approach, where we first have taken general curricula and contextualized it so that it responds to the actual challenges of the environment. From there we have proceeded to conceptualize both the curriculum as well as the process that led to it (e.g., [15], [22]). For several years now, our joint work has been concretized in an accredited contextualized IT program in rural Tanzania. The program's further development is based on constant consideration of the program's successes and challenges (e.g., [15], [16], [17], [18]).

Constructive interest has driven the process from the very beginning. The contextualized IT program at Iringa, Tanzania, has not only offered a meaningful IT program for the needs of Iringa region, but by pushing beyond the limits of standard IT curricula, the program also challenges some orthodox views on computer science education, too [15].

## 5 Conclusions

In this chapter, we have—to our knowledge, for the first time in literature—outlined the computer science perspective to ICT4D research. A computer science approach is necessary for exploring and constructing functional information and communication technology for making a positive change in a given development context. Probably due to a lack of awareness of ICT4D challenges, of which many can be understood or abstracted as computational, very few computer science research groups have addressed ICT4D issues. However, the interest of computer science community is waking up, hand in hand with the increasing markets, for ICT in emerging economies. For instance, cross-cultural design has gained a stable foothold during the past decade (e.g., [2]).

The computer science approaches to ICT4D research can be classified based on how well a particular development challenge is understood or known in a given context. At the other end of the spectrum, a research team with a computer science background needs to devote time to exploring the inputs of the context. At the opposite end, the team can start their design process on a given set of inputs, which specify the situation. In these cases, a somewhat generalized solution can be later adapted to various contexts.

The contribution of computer science is direly needed in the state-of-the-art ICT4D research. In fact, computer science researchers are able to get back to the days of the emerging discipline, but in another societal context. While the current models of computation originated in Europe between the two World Wars and during WW2, and while research in relational databases and the theory of data management are indebted to half-empty commercial planes, the new problems in the world's developing areas call for computer scientists for new exploration.

**Acknowledgments.** Our research was funded by the Academy of Finland grants #128577 (Sutinen) and #132572 (Tedre).

## References

1. Ascher, M., Ascher, R.: Mathematics of the Incas: Code of the Quipu. Dover Publications, New York (1981)
2. Aykin, N. (ed.): Usability and Internationalization of Information Technology. Lawrence Erlbaum, New York (2005)
3. Bangu, N., Haapakorpi, R., Lund, H.H., Myller, N., Ngumbuke, F., Sutinen, E., Vesisenaho, M.: Information Technology Degree Curriculum in Tanzanian Context. In: Proceedings of the IST-Africa 2007 Conference, Maputo, Mozambique, May 9-11, CDROM (2007)
4. Brooks Jr., F.P.: The Computer Scientist as Toolsmith II. Comm. ACM 39(3), 61–68 (1996)
5. Burrell, J., Toyama, K.: What Constitutes Good ICTD Research? Information Technologies and International Development 5(3), 82–94 (2009)
6. Church, A.: An Unsolvable Problem of Elementary Number Theory. Am. J. of Mathematics 58, 354–363 (1936)
7. Donner, J.: Research Approaches to Mobile Use in the Developing World: A Review of the Literature. The Information Society 24, 140–159 (2008)

8. Eglash, R.: African Fractals: Modern Computing and Indigenous Design. Rutgers University Press, New Jersey (1999)
9. Fletcher, P.: The Role of Experiments in Computer Science. Journal of Systems and Software 30(1-2), 161–163 (1995)
10. Hodges, A.: Alan Turing: the Enigma. Vintage, London (1992)
11. Jensen, R.: The Digital Provide: Information (Technology), Market Performance, and Welfare in the South Indian Fisheries sector. The Quarterly Journal of Economics 122(3), 879–924 (2007)
12. Pickering, A.: The Mangle of Practice: Time, Agency, and Science. The University of Chicago Press, Chicago (1995)
13. Pólya, G.: How to Solve It, 2nd edn. Penguin Books Ltd., London (1957)
14. Tedre, M.: The Development of Computer Science: A Sociocultural Perspective. University of Joensuu Press, Joensuu (2006)
15. Tedre, M., Bangu, N., Nyagava, S.I.: Contextualized IT Education in Tanzania: Beyond Standard IT Curricula. Journal of Information Technology Education 8(1), 101–124 (2009)
16. Tedre, M., Kamppuri, M.: Students' Perspectives on Challenges of IT Education in Rural Tanzania. In: Proceedings of IST-Africa 2009 conference, Kampala, Uganda, May 6-8, CD-ROM (2009)
17. Tedre, M., Ngumbuke, F.D., Bangu, N., Sutinen, E.: Implementing a Contextualized IT Curriculum: Ambitions and Ambiguities. In: Proceedings of the 8th Koli Calling International Conference on Computing Education Research, Lieksa, Finland, November 13-16, pp. 51–61 (2009)
18. Tedre, M., Chachage, B., Faida, J.: Integrating Environmental Issues in IT Education in Tanzania. In: Proceedings of the 39th ASEE/IEEE Frontiers in Education Conference, San Antonio, TX, USA, October 18-21 (2009)
19. Tedre, M., Sutinen, E.: Crossing the Newton-Maxwell Gap: Convergences and Contingencies. Spontaneous Generations: A Journal for the History and Philosophy of Science 3(1) (2009)
20. Turing, A.M.: On Computable Numbers, With an Application to the Entscheidungsproblem. Proc. of London Math. Soc., Series 2, 42, 230–265 (1936)
21. Unwin, T. (ed.): ICT4D. Cambridge University Press, Cambridge (2009)
22. Vesisenaho, M., Kemppainen, J., Islas Sedano, C., Tedre, M., Sutinen, E.: Contextualizing ICT in Africa: The Development of the CATI Model in Tanzanian Higher Education. African Journal of Information and Communication Technology 2(2), 88–109 (2006)
23. Wegner, P.: Research Paradigms in Computer Science. In: Proceedings of the 2nd International Conference on Software Engineering, San Francisco, California, USA, October 13-15, pp. 322–330 (1976)

# Searching for Linear Dependencies between Heart Magnetic Resonance Images and Lipid Profiles

Marko Sysi-Aho[1,⋆], Juha Koikkalainen[2], Jyrki Lötjönen[2], Tuulikki Seppänen-Laakso[1], Hans Söderlund[1], Tiina Heliö[3], and Matej Orešič[1]

[1] VTT Technical Research Centre of Finland, Tietotie 2, 02044 Espoo, Finland
Tel.: +358 40 5316949
marko.sysi-aho@vtt.fi
[2] VTT Technical Research Centre of Finland, Tampere, Finland
[3] HYKS Helsinki District Hospital, Helsinki, Finland

**Abstract.** Information derived from "omics" data in life science research are frequently limited by specific spatial or temporal scales these data describe. As a case study of integrating physiological and molecular data in human, here we study associations between the heart magnetic resonance images and serum lipidomic profiles. In the best case, such associations could help infer the physiologic state of the heart from a blood serum sample without need to use expensive imaging techniques. Strong marginal and partial correlations are found between the lipid profiles and parameters derived from the heart images. Regression analyses are applied to study these dependencies in more detail. This study demonstrates the feasibility of mapping lipid profiles to heart images, and thus combining information from two very different scales, small molecules and macroscopic physiologic features. Such mappings could be generalized to other "omics" data as well to complete our picture of the holistic function of a living organism.

## 1 Introduction

During the last decade the maturation of "omics" technologies such as genomics, transcriptomics, proteomics and metabolomics has resulted in overwhelming amounts of data [1,2,3]. There is an ever increasing need to develop methods for extracting information not only from datasets reflecting a specific organismal level such as metabolome, but also on how the parts of the organism interact with each other. The area of systems biology has emerged as a scientific discipline which aims to study such interactions at multiple spatial and temporal scales in living systems [4,5]. One of the first steps in building knowledge onto the "omics" data is to establish a connection between measurements of these data and the more traditional phenotype characteristics such as disease status,

⋆ Corresponding author.

T. Elomaa et al. (Eds.): Ukkonen Festschrift, LNCS 6060, pp. 232–243, 2010.

blood count or outlook of body organs. Such connections provide the basic information on which to build more comprehensive and detailed understanding of the whole system function. Here, we search for dependencies between lipidomics data [6] and magnetic resonance image data (MRI) [7] which are obtained from the same human subjects [8].

Analyses of lipidomics and MRI data pose several computational challenges. The first one is due to high dimensionality of feature space as compared to the number of samples: MR images contain tens of thousands of pixels and hundreds of lipids are identified from blood serum samples analyzed by the ultra performance liquid chromatograph mass spectrometer (LC/MS) [9]. Second main challenge concerns the extraction of physiologically relevant features from the 3D MR images, which make the images easier to interpret and considerably reduce their dimension. A method applied for the feature detection should be computationally efficient, reliably align the images across different subjects and be insensitive to changes in orientation and sizes of the images [10]. Third challenge is related to the profiling of the lipids, which is complicated due to the complex nature of the biologic matrix and global nature of the profiling method. In global profiling the lipids are not *a priori* restricted to known species which makes their quantification and identification more difficult.

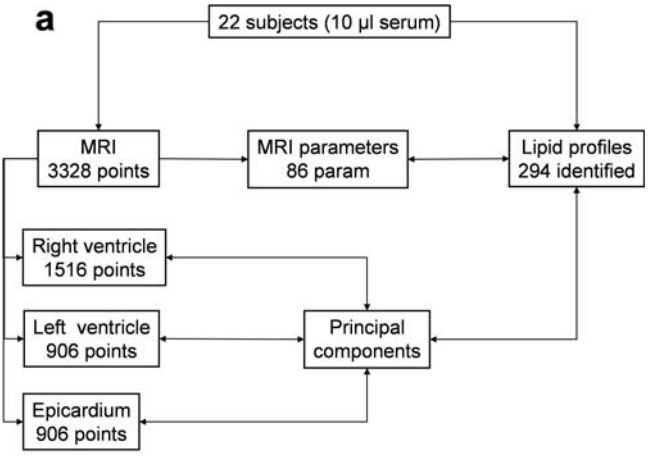
## 2 Methods

### 2.1 Study Population

The total number of subjects in the study was 22. All subjects were of Finnish origin, of which $N = 14$ were female. The age range was $34 \pm 11$ years. As here we are not focusing on associations with health outcomes, no clinical phenotype data are discussed in this paper. An outline of the study and summary of the clinical characteristics of the subjects are shown in Figure 1.

### 2.2 Imaging

The imaging was done as described previously [8]. In short, patients were evaluated by personal and family history, physical examination, 12-lead ECG, and transthoracic echocardiography (M-mode, two-dimensional and Doppler, Vivid 7, GE Medical). The echocardiographic examinations were carried out by experienced cardiologists. Cine MRI was performed with a 1.5 T system (Sonata, Siemens Medical Solution) and a body array coil. A retrospectively ECG-gated segmented Steady State Free Precession imaging was used with following parameters: echo time 1.6 ms, repetition time 3.0 ms, matrix 256 256, field of view 240 340 mm, flip angle 52 degree. Short-axis cine stack and a long-axis cine slice of both ventricles were obtained with a section thickness 6 mm, intersection gap 20%, and temporal resolution 42-49 ms. The images were mapped to a common coordinate system using a surface model for the heart.




**b**

| | |
|---|---|
| N (women/men) | 22 (14/8) |
| Age (years) | 34 (11) |
| Height (m) | 1,73 (0,10) |
| Weight (kg) | 73 (15) |
| Body mass index (kg/m$^2$) | 24,2 (3,4) |

**Fig. 1.** a) Study outline. Blood serum samples and heart MR images were obtained from 22 subjects. The MR images were mapped to a heart surface model (3328 points) consisting of sub-models for the epicardium (906 points) and for the left (906 points) and right (1516 points) ventricles using the method previously described in Ref. [8]. 86 physiologically relevant parameters were extracted from the mapped heart MR images. Blood serum samples were analyzed using UPLC/MS instrument and altogether 294 lipid species were identified. Dependencies between the lipid profiles and the MR image parameters were searched for by analysing correlations and by applying regression models. b) Summary of clinical characteristics of the subjects involved into the study. The reported values are averages and standard deviations (in parenthesis) over the 22 subjects.

### 2.3 Image Analysis

The left and right ventricles and epicardium were semi-automatically segmented by a technician together with a radiologist from each time frame of the cine MRI series with a software tool developed for this purpose [11]. The automatic segmentation took 1-2 minutes. To reach the optimal segmentation accuracy the time used for the manual refinement was not limited and it took approximately 30 minutes. In the tool, an *a priori* heart model, consisting of triangulated surfaces of the ventricles and epicardium, was deformed to fit both short- and long-axis MRI data. Because the same *a priori* model was used for each subject, the number of the surface points was identical in each case (906 points for

left ventricle, 1516 for right ventricle, and 906 for epicardium), and the point-correspondence existed between all the subjects and time frames. The surfaces were rigidly aligned in the same coordinate system to remove the position and orientation variations from the data. This enabled point-wise comparison of the cardiac motion in the study population.

The shape and motion of the heart was modelled using point distribution models (PDM) [12]. A 4D extension of PDMs, a statistical motion model (SMM) [13], was constructed from the triangulated surfaces with the point correspondence. In a 3D PDM the x-, y-, and z- coordinates of the nodes of a surface are catenated as a single shape vector, whereas in a 4D SMM the x-, y-, z-, coordinates from each time frame are catenated in a single vector, for which the PCA is applied to. In this study, only end-diastolic and end-systolic frames were used to make the analysis simpler.Only end-diastolic and end-systolic frames were used to make the analysis simpler.

In SMM, the principal component analysis is applied to the variations in the data, which gives a set of modes of variation describing the typical shape and motion patterns in the data. In practice, the point coordinates of a triangulated surface are concatenated into one vector $x_i$, and the covariance matrix of the data is computed as:

$$\sum = \frac{1}{N-1} \sum_{i=1}^{N} (x_i - \bar{x})(x_i - \bar{x})^T, \tag{1}$$

where $\bar{x}$ is the mean model of the heart, and $N$ is the total number of subjects. The eigenvectors of the covariance matrix are the modes of variation, and the corresponding eigenvalues give the amount of variance the eigenvectors explains. In other words, the larger the eigenvalue the more important is the corresponding eigenvector in describing the variability of the shape and motion of the heart. The number of eigenvectors is limited by the total number of study population subjects minus one. However, the eigenvectors with small eigenvalues are supposed to model only noise and therefore they are discarded from the analysis. In this study, the first $n$ eigenvectors that explained 95% of the total variance were used and rest of the rest were considered representing noise.

The obtained eigenvectors and -values can be used to approximate the study population shape using a linear model

$$x = \bar{x} + \Phi b, \tag{2}$$

where $\Phi$ is a matrix consisting of the eigenvectors and $b$ is a weight vector. Because the eigenvectors are orthogonal, the weights of $i$th study population subject can be computed from

$$b_i = \Phi^T (x_i - \bar{x}). \tag{3}$$

In this study, these weights $b_i$ were used to parameterize the shape and motion of the heart. The SSM was performed for each structure separately and also simultaneously for all the structures. SSM was selected as the parameterization

as it provides a compact and easily understandable parameterization of the shape and motion of the heart.

### 2.4 Lipid Profiling

The lipidome was analyzed as described previously [14]. In brief, serum samples (10 $\mu$l) diluted with 0.15 M NaCl (10 $\mu$l) and spiked with a standard mixture containing 10 lipid species were extracted with a mixture of chloroform and methanol 2:1 (100 $\mu$l). The extraction time was 0.5 h and the lower organic phase was separated by centrifuging at 10 000 r.p.m. for 3 min. Another standard mixture containing three labeled lipid species was added to the extracts and the lipids were analyzed on a Waters Q-Tof Premier mass spectrometer combined with an Acquity Ultra Performance LC$^{TM}$ (UPLC). The column, kept at 50°C, was an Acquity UPLC$^{TM}$ BEH C18 1 × 50 mm with 1.7 $\mu$m particles. The solvent system included water (1% 1 M ammonium acetate, 0.1% HCOOH) and a mixture of acetonitrile and 2-propanol (5:2, 1% 1 M $NH_4Ac$, 0.1% HCOOH). The flow rate was 0.2 ml/min and the total run time including column re-equilibration was 18 min. Data were processed using MZmine2 software (http://mzmine.sourceforge.net). Lipids were identified using the internal spectral library and lipid notation follows the conventions recommended by the LIPID MAPS consortium [15].

Calibration was performed as follows: all monoacyl lipids except cholesterol esters, such as monoacylglycerols and monoacyl-glycerophospholipids, were calibrated with lysophosphatidylcholine GPCho(17:0/0:0) (Avanti Polar Lipids, Alabaster, AL) as an internal standard. All diacyl lipids except phosphatidylethanolamines were calibrated with phosphatidylcholine GPCho(17:0/17:0) (Avanti Polar Lipids), the phosphatidylethanolamines with GPEtn(17:0/17:0) (Avanti Polar Lipids) and the triacylglycerols and cholesterol esters with triacylglycerol TG(17:0/17:0/17:0) (Larodan Fine Chemicals, Malmö, Sweden).

### 2.5 Summary of Data Sets

The two main data sets of this study that are obtained by processing the MR images and LC/MS spectra as described above are:

- The MRI parameter data, a 22×86 matrix of continuous valued variables derived from the MRI surface mapped data.
- The lipid data, a 22×294 matrix of continuous valued variables describing the concentration estimates of the identified lipids.

## 3 Results

### 3.1 Correlations between Lipid Data and MRI Parameter Data

For simplicity and in particular for the number of variables being much larger than the number of samples, $p \gg n$, both in the MR image parameter data

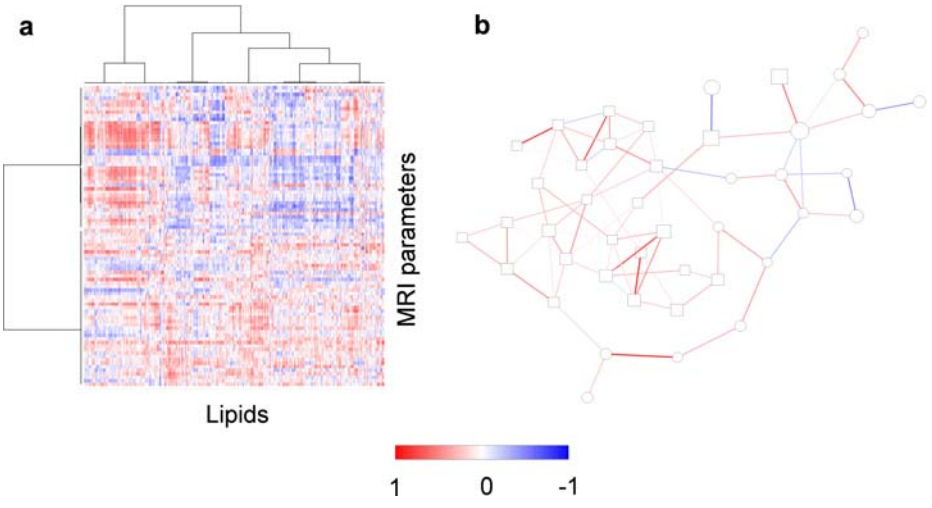



**Fig. 2.** Correlations between lipid profiles and MR image parameters. a) Heatmap of correlations between the lipid profiles and the MR image parameters shows that both positive and negative correlations between certain lipid classes and image parameters exist. b) Partial correlation graph. Squares correspond to lipid variables and circles to MRI parameter variables. An edge denotes partial correlation between the nodes it connects and colour coding is the same as that used in the heatmap. Width of an edge is proportional to the inverse of the non-rejection rate [16], which indicates trust on the existence of the edge. A threshold of 0.55 was used for the non-rejection rate and all edges with higher values were omitted from the graph.

($p = 86$) and in the lipid data ($p = 294$) we confine ourselves to linear dependencies between the two data sets. The marginal correlations between image parameters and lipid profiles were assessed by calculating the Pearson's correlation coefficients and they were visualized using the heatmap function of the R statistical software (R) (www.r-project.org). In order to take the interactions among the lipid profiles and MRI parameters into account, partial correlations were estimated using the R package *qpgraph* [16]. The estimated partial correlations were visualized as a network using Cytoscape (www.cytoscape.org) and yEd (www.yworks.com) graphical editors.

Both marginal and partial correlations reveal that there are linear dependencies between the serum lipid profiles and the MR images. These dependencies are illustrated in Figure 2. The correlation values are colour coded such that positive/negative correlations correspond to red/blue colours. White colour denotes no correlation. In Figure 2 a) lipids are on the horizontal axis and MRI parameters on the vertical axis. One can see areas dominated by the red and blue colours indicating that specific lipid classes are consistently correlated with the same heart image features. Figure 2 b) is a reduced presentation of the partial correlations between the lipid profiles (squares) and the MRI parameters

(circles). Colour of an edge denotes the partial correlation between the nodes it connects and the coding is the same as that used in the heatmap of Figure 2 a). Width of an edge is proportional to the inverse of the non-rejection rate [16], which indicates trust on the existence of the edge. A threshold of 0.55 was used for the non-rejection rate and all edges with higher values were omitted from the graph. It is noteworthy that lipids tend to show stronger correlations between themselves than the MRI parameters do. This is expected as the MRI parameters are derived to give moderate number of physiologically relevant variables that describe the heart, whereas the lipids are linked through their reaction pathways. It is also interesting that only a small subset of the lipids are linked to the MRI parameters and thus their role in explaining changes in the physiology of the heart may be crucial.

### 3.2 Regression of Lipid Profiles on MRI Parameters

In addition to correlation based analysis, regression methods were applied to elucidate dependencies between the MR image parameters and the lipid profiles in more detail. Due to the $p \gg n$ the regression must be regularized and we chose to apply the *elasticnet* regression method [17] as implemented in the elasticnet package of R. Elasticnet is convenient for our current application because the $L_1$ and $L_2$ penalties on the regression coefficients can be flexibly tuned by changing $u_1$ and $u_2$ respectively in Eq. 4, which allows exploration of solutions ranging from simple models with no lipids at all to more complex ones that are either sparse, containing non-correlated lipids, or non-sparse with several correlated lipids included. Elasticnet solution is obtained by minimizing

$$L(u_1, u_2, b) = \|y - Xb\|^2 + u_1\|b\|_1^2 + u_2\|b\|_2^2, \tag{4}$$

where $u_1$ and $u_2$ are tuning parameters, $y$ is the response variable and $X$ a matrix containing the explanatory variables. In this study, each of the image parameters were separately explained by the whole lipid profile data and the tuning parameters were selected by minimizing the mean cross-validation error (see Figure 3 a):

$$e = 1/n \sum_{i=1}^{n} e_i,$$

$$e_i = e(p(i), u_1, u_2, y, X) = \|y_{p(i)} - X_{-p(i)} b_{-p(i)}\|_2^2, i = 1, \ldots, n. \tag{5}$$

Above $p = (p(1), \ldots, p(n))$ is partition of the samples into n blocks and $-p(i)$ refers to all samples but those that belong to the $i$th block (which belong to $p(i)$). Here we randomly assigned the samples into $n = 5$ blocks for 100 times and calculated the average errors.

The cross-validation errors were also used to assess whether the lipid profiles in general explained an image parameter. The criterion was defined as follows:

$$e_{min} + sd(e_{i,min}) < e_{cons} \tag{6}$$

$\Rightarrow$ The image parameter in question can be explained with the lipid profiles.

Above $e_{min}$ is the minimum average cross-validation error, $e_{i,min}$ is the standard deviation of the cross-validation errors over the $n$ blocks for the model that gives the minimum average error and $e_{cons}$ is the average cross-validation error of the constant model. In other words, if the minimum mean cross-validation error plus one standard deviation of the error at minimum was lower than the mean cross-validation error of the constant model, *i.e.*, a model that contained no lipids, then the image parameter was interpreted to be dependent on the lipid profiles. The final model for each image parameter was estimated using all samples and the pair of tuning parameters that corresponded to the minimum average cross validation error. Due to the small number of samples ($n = 22$) statements on the dependencies are qualitative rather than quantitative. For example, lipids that correspond to non-zero coefficients in the regression model are merely considered being influential and answers to more detailed questions, for example, on the strength of the influence are open to various interpretations.

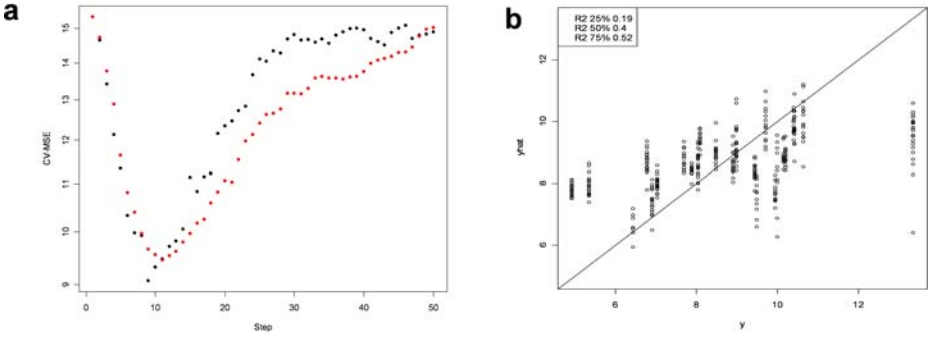



**Fig. 3.** Associations between the MRI parameters and lipid profiles were studied using the elasticnet regression model [17]. Elasticnet has two tuning parameters that control the $L_1$ and $L_2$ penalties on the regression coefficients. a) An average cross-validation error for a selected MRI parameter as explained by the lipid profiles. The horizontal axis (step) corresponds to the $L_1$ tuning parameter with the leftmost point (step = 0) equalling the error of the constant model, that is, the model containing no lipids. Optimal tuning parameters were searched for by exploring a grid of pre-selected $L_1 \times L_2$ values. The pair that gave smallest average cross-validation error (vertical axis) was selected for the final regression. Here, for demonstration purposes, curves for only two different $L_2$ parameters are shown. The black curve has smaller minimum error (at step 9) than the red curve (at step 11), which corresponds to larger $L_2$ penalty. b) In order to estimate the degree by which an MRI parameter can be explained with the lipid profiles a cross-validated $R^2$ statistics was calculated between the cross-validated predictions and the actual measurements for the selected regression model.

### 3.3 Regression of Lipid Profiles on the Heart Surface Model

The sensitivity of heart images to changes in selected lipid profiles was also studied. The heart MR images were decomposed into their principal components, which in turn were explained by the lipid profiles using ridge regression [18]. The regression model was fitted using all the mapped heart MR images and lipid profiles. The averaged lipid profiles were then entered into the fitted regression model in order to obtain a mean heart model. Then the concentration of a selected lipid was changed by plus two times the standard deviation of the lipid value in the study population and a new heart model was computed using the regression model. The change in heart model was visually evaluated for the end-diastolic shape and the deformation from end-diastole to end-systole using colour maps overlaid on the heart model. The changes and deformations were projected on the surface normal of the mean heart model to regularize the results.

Sensitivity of the heart model to lipid concentration changes is demonstrated in Figure 4 for one particular lipid. Figure 4 a) shows in red the surface model of the heart as obtained from the average lipid profiles. The blue colour shows the surface after changing the concentration of the selected lipid by two standard deviations. Figure 4 b) shows the shape change along the surface normal when the lipid value was changed by two standard deviations with reddish colours corresponding to outward deviations and bluish colours to inward deformations. Figure 4 c) shows the end-diastole to end-systole motion along surface normal for the mean heart model (red = inward deformation, blue = outward deformation), and Figure 4 d) the end-diastole to end-systole motion along surface normal when the lipid value was changed by two standard deviations.

### 3.4 Discussion

In this study we demonstrated the feasibility of mapping lipidomic profiles to various physiologic features reflecting the function of the heart. Due to the high dimensionality of both the MRI parameter and lipid profile data sets only linear dependencies were considered. Positive and negative marginal and partial correlations existed between specific lipid species and the heart MRI parameters. Regression analyses gave further information on these dependencies, shedding the light on which lipids most accurately describe specific features of the heart.

Over-fitting of the statistical models is always a risk when the data dimensions are high and the number of subjects is small, as in our study. For this reason we prefer qualitative rather than quantitative conclusions. For example, associations determined from cross-validation errors of the elasticnet regression models are robust in the sense that variability from the data and separate model fits are controlled. Thus, making a statement that a specific MR image parameter is linked to the lipid profiles is quite safe but more quantitative statements, for example, on the strength of the contribution of each lipid are still vague. The same applies to the heart image changes when the lipid concentrations are varied. Here we demonstrated the feasibility of constructing such mappings, rather than using them to interpret the exact effects of the lipids. Reconstructing the heart

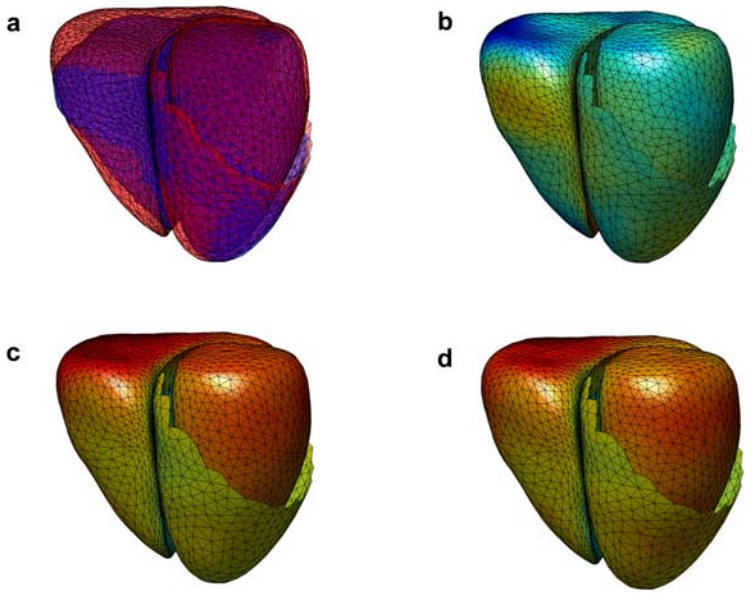



**Fig. 4.** Sensitivity of the heart model on lipid concentration changes. The MRI was decomposed to principal components which were explained by the lipid profiles (see Figure 1). a) Red: original heart, blue: surface after changing the lipid value by two standard deviations. b) The shape change along the surface normal when the lipid value was changed by two standard deviations (red = outward, blue = inward deformation). c) End-diastole to end-systole motion along surface normal for mean model (red = inward deformation, blue = outward deformation). d) End-diastole to end-systole motion along surface normal when the lipid value was changed by two standard deviations.

image from the principal components explained by the lipid profiles via the regression models does not reveal local associations. For revealing these, the heart images should first be partitioned into small enough regional units and mappings between these units and the lipid profiles should be explored. The methodology presented here can be straight applied to such studies.

In order to assess the ultimate power of the approach presented in this paper, a large scale study containing thousands of samples would be needed. However, already with the current small sample set the concept has proven feasible. We thus believe that research efforts on integrating data which describe various scales of a living organism, such as "omics" data and images, will be emphasized in the future and importantly complement our picture of the holistic function of the organism.

### 3.5 Conflict of Interest

The authors declare no conflict of interest.

## References


1. Fan, J.-B., Chee, M.S., Gunderson, K.L.: Highly parallel genomic assays. Nature Reviews Genetics 7(8), 632–644 (2006)
2. Tan, K., Ipcho, S.V.S., Trengove, R.D., Oliver, R.P., Solomon, P.S.: Assessing the impact of transcriptomics, proteomics and metabolomics on fungal phytopathology. Molecular Plant Pathology 10(5), 703–715 (2009)
3. Sreekumar, A., Poisson, L.M., Rajendiran, T.M., Khan, A.P., Cao, Q., Yu, J., Laxman, B., Mehra, R., Lonigro, R.J., Li, Y., Nyati, M.K., Ahsan, A., Kalyana-Sundaram, S., Han, B., Cao, X., Byun, J., Omenn, G.S., Ghosh, D., Pennathur, S., Alexander, D.C.: Metabolomic profiles delineate potential role for sarcosine in prostate cancer progression. Nature 457(7231), 910–914 (2009)
4. Newman, J.R., Weissman, J.S.: Systems biology: many things from one. Nature 444(7119), 561–562 (2006)
5. Nicholson, J.K., Lindon, J.C.: Systems biology: Metabonomics. Nature 455(7216), 1054–1056 (2008)
6. Watson, A.D.: Thematic review series: systems biology approaches to metabolic and cardiovascular disorders. Lipidomics: a global approach to lipid analysis in biological systems. J. Lipid Res. 47(10), 2101–2111 (2006)
7. Karamitsos, T.D., Francis, J.M., Myerson, S., Selvanayagam, J.B., Neubauer, S.: The role of cardiovascular magnetic resonance imaging in heart failure. J. Am. Coll. Cardiol. 54(15), 1407–1424 (2009)
8. Koikkalainen, J.R., Antila, M., Lotjonen, J.M., Helio, T., Lauerma, K., Kivisto, S.M., Sipola, P., Kaartinen, M.A., Karkkainen, S.T., Reissell, E., Kuusisto, J., Laakso, M., Oresic, M., Nieminen, M.S., Peuhkurinen, K.J.: Early familial dilated cardiomyopathy: identification with determination of disease state parameter from cine MR image data. Radiology 249(1), 88–96 (2008)
9. Sun, J., Schnackenberg, L.K., Holland, R.D., Schmitt, T.C., Cantor, G.H., Dragan, Y.P., Beger, R.D.: Metabonomics evaluation of urine from rats given acute and chronic doses of acetaminophen using NMR and UPLC/MS. J. Chromatogr. B Analyt. Technol. Biomed. Life Sci. 871(2), 328–340 (2008)
10. Lotjonen, J.M., Wolz, R., Koikkalainen, J.R., Thurfjell, L., Waldemar, G., Soininen, H., Rueckert, D.: The Alzheimer's Disease Neuroimaging Initiative: Fast and robust multi-atlas segmentation of brain magnetic resonance images. Neuroimage (2009)
11. Lotjonen, J., Kivisto, S., Koikkalainen, J., Smutek, D., Lauerma, K.: Statistical shape model of atria, ventricles and epicardium from short- and long-axis MR images. Med. Image Anal. 8(3), 371–386 (2004)
12. Cootes, T.F., Taylor, C.J., Cooper, D.H., Graham, J.: Active shape models-Their training and Application. Comput. Vis. Image Underst. 61, 38–59 (1995)
13. Perperidis, D., Mohiaddin, R., Rueckert, D.: Construction of a 4D statistical atlas of the cardiac anatomy and its use in classification. In: Int. Conf. Med. Image Comput. Comput. Assist. Interv., vol. 8(Pt 2), pp. 402–410 (2005)
14. Laaksonen, R., Katajamaa, M., Paiva, H., Sysi-Aho, M., Saarinen, L., Junni, P., Lutjohann, D., Smet, J., Van Coster, R., Seppanen-Laakso, T., Lehtimaki, T., Soini, J., Oresic, M.: A systems biology strategy reveals biological pathways and plasma biomarker candidates for potentially toxic statin-induced changes in muscle. PLoS One 1, e97 (2006)
15. Fahy, E., Subramaniam, S., Murphy, R.C., Nishijima, M., Raetz, C.R., Shimizu, T., Spener, F., van Meer, G., Wakelam, M.J., Dennis, E.A.: Update of the LIPID MAPS comprehensive classification system for lipids. J. Lipid Res. 50(Suppl. S9-14) (2009)

16. Castelo, R., Roverato, A.: Reverse engineering molecular regulatory networks from microarray data with qp-graphs. J. Comput. Biol. 16(2), 213–227 (2009)
17. Zou, H., Hastie, T.: Regularization and variable selection via the elastic net. J. R Statist. Soc. B 67(2), 901–920 (2005)
18. Jain, R.K.: Ridge regression and its application to medical data. Comput. Biomed. Res. 18(4), 363–368 (1985)

# The Support Vector Tree

Antti Ukkonen

HIIT, Helsinki University of Technology*
antti.ukkonen@hiit.fi

**Abstract.** Kernel based methods, such as nonlinear support vector machines, have a high classification accuracy in many applications. But classification using these methods can be slow if the kernel function is complex and if it has to be evaluated many times. Existing solutions to this problem try to find a representation of the decision surface in terms of only a few basis vectors, so that only a small number of kernel evaluations is needed. However, in all of these methods the set of basis vectors used is independent of the example to be classified. In this paper we propose to adaptively select a small number of basis vectors given an unseen example. The set of basis vectors is thus not fixed, but it depends on the input to the classifier. Our approach is to first learn a non-sparse kernel machine using some existing techique, and then using training data to find a function that maps unseen examples to subsets of the basis vectors used by theis kernel machine. We propose to represent this function as a binary tree, called a support vector tree, and devise a greedy algorithm for finding good trees. In the experiments we observe that the proposed approach outperforms existing techniques in a number of cases.

## 1 Introduction

Classification is a fundamental problem in machine learning. Typical research on classification methods concentrates on improving either the scalability of the learning algorithm, or accuracy of the resulting classifier, or both. These properties are important problems in most, if not all applications. However, in some cases the running time of the classifier itself can be of importance. Speech recognition and packet classification in IP networks are classical examples of applications where it is crucial that classification can be carried out in "real time". Another example are information retrieval systems that use machine learning algorithms to classify documents to relevant and non-relevant ones. If the classifier is applied to every document that match a given query, a single evaluation of the classifier has to be very fast, as there can be tens of thousands of such documents. In such cases it is thus necessary to resort to relatively simple classifiers.

However, we might obtain a better classification accuracy with kernel or ensemble based methods that are computationally slower. In this paper we study

* The work was conducted while the author was visiting Universitat Pompeu Fabra / Yahoo Research Barcelona.

T. Elomaa et al. (Eds.): Ukkonen Festschrift, LNCS 6060, pp. 244–259, 2010.

the problem of speeding up classification using kernel machines. We use the non-linear support vector machine (SVM) [23] as an example, but our approach is applicable to any classifier that uses a similar decision rule. In particular, we can consider any algorithm where the decision is based on the following sum:

$$\text{class}(\mathbf{x}) = \text{sign}\Big( \sum_{(\mathbf{s}_j, a_j) \in S} a_j g(\mathbf{s}_j, \mathbf{x}) + \theta \Big). \tag{1}$$

The set $S$ together with the function $g$ represent the classifier. Evaluating this sum can be slow if the set $S$ is large, and if the function $g$ is computationally complex. One solution is to explicitly restrict the size of $S$ when learning the classifier. This is a common approach, as it can also result in faster learning algorithms. The idea we propose in this paper is rather different. *Instead of computing the sum in Equation 1 over the entire set $S$, we compute it over the set $f(\mathbf{x}) \subset S$ that depends on the example $\mathbf{x}$ we are classifying.*

We consider the following approach: First we find the set $S$ representing a kernel based classifier, such as an SVM. This can be done using any standard algorithm. Given $S$ we learn a function $f$ that maps any given example $\mathbf{x}$ to a subset $f(\mathbf{x}) \subset S$. This function can be learned either with the same training data that were used to find the set $S$, or using a different set of examples. When classifying $\mathbf{x}$, we compute the sum in Equation 1 only over the basis vectors in $f(\mathbf{x})$. To represent the function $f$, we propose a binary tree that induces a disjoint partition of the feature space. Examples belonging to the same region are mapped to the same subset of $S$. We call this tree the *support vector tree*.

The rest of this paper is structured as follows: This section concludes with a discussion of related work and the detailed contributions of this article. In Section 2 we give a general definition of the problem, while the support vector trees are described in Section 3. Finding an optimal tree is likely to be hard, and in Section 4 we propose a greedy heuristic with polynomial running time that finds good trees in practice. In the experiments (Section 5) we compare our method with recent related work, and Section 6 is a short conclusion.

### 1.1 Related Work

Kernel machines and in particular SVMs [23] have been studied considerably for the past fifteen years. A complete review of this work is obviously beyond the scope of this paper. However, we try to mention most articles that are relevant considering the main objective of this paper: *speeding up classification with SVMs*. Also, for the basics of SVM learning we recommend the excellent tutorial by Burges [4], as we do not discuss these here.

Finding sparse SVMs is an old and well studied problem. For interesting initial work on the topic we refer the reader to [5,3]. Burges and Schölkopf [5] devise a post-processing algorithm for finding a reduced set of basis vector given the $S$ of support vectors. Most of the approaches that followed differ from [5] and our paper by formulating a version of the SVM learning problem that attempts to *directly* find a sparse solution. Examples of older work on this kind of techniques

include [20], [11], [22], and [19]. More recently, in [24] Wu et. al. discuss another direct method for building sparse kernel machines. They report experimental results where the accuracy of the full SVM is in some cases achieved using only a very small fraction (5%) of the original support vectors. Unlike other related work [8] proposes an ensemble-like method.

A recent paper that studies the problem of sparse SVM learning is [16]. While the main motivation for [16] seems to be making SVM training more scalable, the proposed algorithm also has the property of giving solutions that can have a considerably smaller number of support vectors without a significant decrease in accuracy. In the experiments of [16] it is shown that the number of basis vectors can be reduced by two orders of magnitude without affecting the accuracy of the resulting classifier. This is similar to the results in [24]. Another interesting property of [16] is that the set $S$ may contain vectors outside the training data. We compare our method against the algorithm proposed in [16].

Most of the methods for sparse SVM learning let the learning algorithm automatically determine the number of support vectors. However, in some applications it is useful to be able to set the desired number of support vectors in advance. An algorithm that admits this is proposed in [10]. Also the method we describe here allows the "budget" to be specified in advance, as do the methods of [24] and [16].

### 1.2 Contributions of This Paper

- We propose a method to speed up classification using kernel machines by using only a subset of support vectors. This subset is a function of the example to be classified.
- We propose a method called the support vector tree to efficiently select the support vectors given an unseen example. The method resembles a decision tree but differs on a number of important aspects.
- We propose a greedy algorithm for learning a support vector tree given training data. An analysis based on the Master theorem [9] shows that the running time of this algorithm is at least of order $O(n^3)$ where $n$ is the size of the training data.
- We describe experiments where the support vector tree is compared with a classical nonlinear SVM and a state-of-the art algorithm for learning sparse SVMs on a number of benchmark data sets.

## 2 Problem Definition

We continue with some formal definitions. Let $\Omega$ be a universe of objects. Usually we let $\Omega = \mathbb{R}^n$, but the proposed method is to a large extent oblivious to the type of input examples. Let $S \subset \Omega \times \mathbb{R}$ be a set of objects from $\Omega$ together with a *weight* associated with each object. That is, we have $S = \{(\mathbf{s}_j, a_j)\}_{i=1}^m$. Moreover, denote by $g : \Omega \times \Omega \to \mathbb{R}$ a function mapping pairs of objects from $\Omega$ to the set of reals. We can think that the set $S$ represents for example a

nonlinear SVM. The function $g$ is the kernel function, and the $(\mathbf{s}_j, a_j)$ pairs are the support vectors and their weights. We consider classifiers where an example $\mathbf{x} \in \Omega$ is assigned to the class $+1$ or $-1$ depending on the sign of the sum in Equation 1.

Using this sum to classify a new example $\mathbf{x}$ requires $m$ evaluations of the function $g$. This may be a problem in some applications if the set $S$ is large and computing $g$ is slow. This can happen if $g$ is e.g. a string kernel [18,21]. As a remedy, most previous approaches to speed up classification with kernel machines look for sparse solutions to the learning problem. Roughly put, the idea is to find a set $S'$, so that $|S'| \ll |S|$, and

$$\sum_{(\mathbf{s}'_j, b_j) \in S'} b_j g(\mathbf{s}'_j, \mathbf{x}) \approx \sum_{(\mathbf{s}_j, a_j) \in S} a_j g(\mathbf{s}_j, \mathbf{x}).$$

A common property of the previous approaches is thus that all input instances are classified using the same set $S'$. However, it is easy to imagine that if $S'$ is selected separately for each unseen example $\mathbf{x}$, we may obtain a better approximation, and possibly need a smaller number of evaluations of the function $g$. More precisely, instead of computing the sum over a fixed set $S'$ when classifying $\mathbf{x}$, we compute it over a set $S'$ that depends on the input $\mathbf{x}$. We express this idea more formally in the rest of this section.

Let $S$ and $g$ be as defined above, and let $D \subset \Omega$ be a set of input examples. The examples in $D$ can be labeled or unlabeled. Furthermore, let $\Phi$ be a family of functions that map objects from the set $\Omega$ to subsets of the set $S$. The general formulation of our problem is as follows:

*Problem 1.* Given the data $D = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$, the set $S = \{(\mathbf{s}_1, a_1), \ldots, (\mathbf{s}_m, a_m)\}$, the function $g$, and the family of functions $\Phi$, find the function $f \in \Phi$ that minimizes the cost

$$\sum_{\mathbf{x}_i \in D} \underbrace{\left( \sum_{(\mathbf{s}_j, a_j) \in f(\mathbf{x}_i)} a_j g(\mathbf{s}_j, \mathbf{x}_i) \; - \sum_{(\mathbf{s}_j, a_j) \in S} a_j g(\mathbf{s}_j, \mathbf{x}_i) \right)^2}_{\mathrm{c}(\mathbf{x}_i)}. \tag{2}$$

Note that we are approximating the sum instead of only its' sign. There are two reasons for this. First, we expect this to better retain the generalization ability of the resulting classifier. Second, in some applications we are not only interested in the sign, but also the exact value of the sum. This is the case for instance if the kernel machine is to be used for ranking [13].

Clearly Problem 1 is under-constrained in the sense that if the family $\Phi$ is not chosen carefully, we may end up with a trivial solution that simply maps every $\mathbf{x} \in \Omega$ to the set $S$. This is not meaningful considering that we want to reduce the number of evaluations of the function $g$. Therefore, the problem is interesting only if we restrict the kinds of functions $\Phi$ may contain.

From a practical standpoint we are interested in functions $f$ such that $|f(\mathbf{x})| \leq k$ for all $\mathbf{x}$ for some fixed $k$. To solve Problem 1, we can consider each $\mathbf{x}_i \in D$

separately, and for each find a subset $S'$ of $S$, $|S'| \leq k$, that minimizes $\mathrm{c}(\mathbf{x}_i)$. This can be seen as a variant of the NP-complete SUBSET-SUM problem, where the question is to find a subset of a given set $A$ of numbers that sum up to a given number $B$ [12]. In our case we have $A = \{a_j g(\mathbf{s}_j) \mid (a_j, x_j) \in S\}$ and $B = \sum_{a \in A} a$. (SUBSET-SUM is usually defined for integers. We can scale and subsequently round the values in $\mathbf{X}^{D,S}$ so that the input is integer valued.) Finding an optimal $f(\mathbf{x}_i)$ is thus unlikely to be easy. And even if we could find the optimal subset for each instance in the training data, we still need to be able to use the function $f$ with unseen examples. One solution would be to store all of $D$ together with the $f(\mathbf{x}_i)$s, and for an example $\mathbf{x} \notin D$ let $f(\mathbf{x}) = f(\mathbf{x}^*)$ where $\mathbf{x}^* = \arg\min_{\mathbf{x}_i \in D} \mathrm{dist}(\mathbf{x}_i, \mathbf{x})$. This means, however, that we have to solve a nearest neighbor query when evaluating $f(\mathbf{x})$.

The proposed method can be of practical interest only if computing $f(\mathbf{x})$ is considerably faster than evaluating the function $g$ a number of times. Therefore, we must restrict ourselves to functions that can be computed very efficiently. The family of functions we consider in this paper is discussed next.

## 3 Tree Based Partitioning of the Input Space

In this paper we consider functions $f$ that can be represented as binary trees. These trees partition the feature space to disjoint subsets, and provide an efficient means to locate an unseen example $\mathbf{x}$ in the subset it belongs to. Each subset of the feature space uses a different set of support vectors. The concept is somewhat similar to the decision tree classifier, but it's implementation and use are quite different.

### 3.1 Basic Definitions

Let $\mathcal{T}$ be a binary tree, and denote by $N$ a node of $\mathcal{T}$. With every node $N$ is associated a pair $(a, \mathbf{s}) \in S$. The *node score* of $N$ given by $a_N g(\mathbf{x}, \mathbf{s}_N)$, where $a_N \in \mathbb{R}$ and $\mathbf{s}_N \in \Omega$ are the values associated with $N$, and $g$ is e.g. a kernel function. The set $f(\mathbf{x})$ is found by following a path from the root of $\mathcal{T}$ to a leaf node. Based on the value of the node score at a node $N$ we enter either the left or right subtree of $N$ until a leaf node is reached. When this happens, we sum the node scores on the path from the root to the leaf, and use this as an approximation of the sum in Equation 1. There are some aspects to this that should be emphasized:

1. Unlike with decision and regression trees, branching is not based on the value of a feature, but on the node score $a_N g(\mathbf{x}, \mathbf{s}_N)$. This means the partition of the feature space induced by the tree is *not* in general the disjoint union of axis-aligned (hyper)rectangles.
2. The value computed by the tree is the sum of the node scores on the path from the root to a leaf node. This is in contrast to regression trees where the output is simply a value stored at each leaf. A consequence of this is that two examples, $\mathbf{x}_1, \mathbf{x}_2 \in \Omega$, that both follow the same path and hence end up at the same leaf, may still produce considerably different output values.

We continue with the definition of the support vector tree $\mathcal{T}$.

**Definition 1.** *A support vector tree $\mathcal{T}$ is the tuple $(\mathcal{N}, R, l, r)$, where $\mathcal{N}$ is a set of nodes, $R \in \mathcal{N}$ is the root node of the tree, and $l$ and $r$ are functions mapping the set $\mathcal{N}$ onto $\{\mathcal{N} \cup \emptyset\}$. Given a node $N \in \mathcal{N}$, $l(N)$ and $r(N)$ are the root nodes of the left and right subtrees of $N$, respectively. To each node $N \in \mathcal{N}$ is associated three values: $t_N \in \mathbb{R}$, $a_N \in \mathbb{R}$, and $\mathbf{s}_N \in \Omega$.*

Using $\mathcal{T}$ we find the set $f(\mathbf{x})$ by collecting all $(\mathbf{s}, a)$ pairs that are associated with nodes on a path from the root of $\mathcal{T}$ to a leaf node. At every node $N$ the path goes either in the left or right subtree depending on the value $a_N g(\mathbf{x}, \mathbf{s}_N)$. If this value is less or equal to the node-specific threshold $t_N$ the path continues to the left subtree, otherwise it continues to the right subtree. Note that instead of computing the set $f(\mathbf{x})$, it is more convenient to evaluate the sum in Equation 2 directly over the tree $\mathcal{T}$. We define the following:

**Definition 2.** *Let $g : \Omega \times \Omega \to \mathbb{R}$ be a function, denote by $\mathcal{T} = (\mathcal{N}, R, l, r)$ a support vector tree as defined above, and let $\mathbf{x} \in \Omega$. Denote by $\text{score}_N(\mathbf{x})$ the node score of $\mathbf{x}$ at node $N \in \mathcal{N}$. We let $\text{score}_N(\mathbf{x}) = a_N g(\mathbf{x}, \mathbf{s}_N)$. Denote by $\text{value}_N(\mathbf{x})$ the value of $\mathbf{x}$ in the subtree of $\mathcal{T}$ rooted at node $N \in \mathcal{N}$. We let*

$$\text{value}_N(\mathbf{x}) = \begin{cases} 0 & \text{if } N = \emptyset, \\ \text{score}_N(\mathbf{x}) + \text{value}_{l(N)}(\mathbf{x}) & \text{if } \text{score}_N(\mathbf{x}) \leq t_N, \\ \text{score}_N(\mathbf{x}) + \text{value}_{r(N)}(\mathbf{x}) & \text{if } \text{score}_N(\mathbf{x}) > t_N. \end{cases}$$

*Finally, denote by $\mathcal{T}(\mathbf{x})$ the value of $\mathbf{x}$ in the entire tree $\mathcal{T}$. We let $\mathcal{T}(\mathbf{x}) = \text{value}_R(\mathbf{x})$, where $R$ is the root node of $\mathcal{T}$.*

Definition 1 does not restrict the size of $\mathcal{T}$ in any way. To reduce the number of evaluations of the function $g$, we must constrain the height of the tree. Denote by $\Phi_{\mathcal{T}}^h$ the set of trees where the length of the longest path from the root to a leaf is at most $h$. The general problem we discuss in the remaining of this paper is the following.

*Problem 2.* Given the training data $D \subset \Omega$, the set $S \subset \Omega \times \mathbb{R}$, and the function $g$, find the tree $\mathcal{T} \in \Phi_{\mathcal{T}}^h$ s.t.,

$$\sum_{\mathbf{x} \in D} \big(\mathcal{T}(\mathbf{x}) - \sum_{(\mathbf{s}_j, a_j) \in S} a_j g(\mathbf{s}_j, \mathbf{x})\big)^2 \tag{3}$$

is minimized.

Note that if $D = \{\mathbf{x}\}$, i.e., $D$ contains only one example $\mathbf{x}$, the solution to Problem 2 is a path. Clearly no branching is needed for a single input instance. As discussed above, this problem is related to SUBSET-SUM, and hence it is unlikely that efficient solutions exist for Problem 2. In this paper we consider a greedy heuristic that leads to well performing trees in practice.

### 3.2 A Simple Exact Algorithm for Balanced Trees

Before presenting the main algorithm of this paper, we briefly describe and analyze the trivial algorithm for solving Problem 2 exactly in the special case where the set $\Phi_{\mathcal{T}}^h$ is further restricted to contain only balanced trees, meaning that the number of training examples belonging to the left subtree of a node is the same as the number belonging to the right subtree. For the remaining discussion it is convenient to consider the following matrix:

**Definition 3.** *Given the data* $D = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$, *the set* $S = \{(\mathbf{s}_1, a_1), \ldots, (\mathbf{s}_m, a_m)\}$, *and the function* $g$, *denote by* $\mathbf{X}^{D,S}$ *the* $n \times m$ *matrix with*

$$\mathbf{X}_{ij}^{D,S} = a_j g(\mathbf{x}_i, \mathbf{s}_j).$$

*The* $i$*th row of* $\mathbf{X}^{D,S}$ *is denoted by* $\mathbf{X}_{i\cdot}^{D,S}$, *and the* $j$*th column of* $\mathbf{X}^{D,S}$ *is denoted by* $\mathbf{X}_{\cdot j}^{D,S}$. *Moreover, given the sets* $I$ *and* $J$ *of integers, denote by* $\mathbf{X}_{I,J}^{D,S}$ *the sub-matrix of* $\mathbf{X}^{D,S}$ *containing the rows specified by* $I$ *and columns specified by* $J$.

Let $\mathbf{r}^{D,S}$ be the vector of row sums of $\mathbf{X}^{D,S}$, and denote by $\mathbf{r}_i^{D,S}$ the $i$th component of $\mathbf{r}^{D,S}$. That is, we have $\mathbf{r}_i^{D,S} = \sum_j \mathbf{X}_{ij}^{D,S}$. In the following we write simply $\mathbf{X}$ and $\mathbf{r}$ if $D$ and $S$ are clear from the context or otherwise irrelevant. Clearly for the $i$th item in $D$ the second sum in Equation 3 is precisely $\mathbf{r}_i$.

Expressed in terms of the matrix $\mathbf{X}$, the learning task of Problem 2 is to approximate the vector $\mathbf{r}$ with appropriate subsets of the columns. It is useful to think that to every node $N$ of $\mathcal{T}$ is associated the matrix $\mathbf{X}_{I,\cdot}$, where $I$ is a subset of the row indices of $\mathbf{X}$. To learn $\mathcal{T}$ we must find an optimal split for the rows of $\mathbf{X}_{I,\cdot}$ at each node $N$. Since the $j$th column of $\mathbf{X}^{D,S}$ corresponds to the pair $(a_j, \mathbf{s}_j) \in S$, we can parameterize this optimization problem with two parameters per node $N$: the threshold $t_N$ and column $j_N$. These define a split of $\mathbf{X}_{I,\cdot}$ at node $N$.

Define the sets $L_I(t_N, j_N)$ and $R_I(t_N, j_N)$ so that $L_I(t_N, j_N) = \{i \in I \mid \mathbf{X}_{ij_N} \leq t_N\}$ and $R_I(t_N, j_N) = \{i \in I \mid \mathbf{X}_{ij_N} > t_N\}$. Moreover, let $P(N)$ denote the set of nodes on the path from the root of a tree to the parent of node $N$. Using this, we let

$$\sigma(N)_i = \sum_{N' \in P(N)} X_{ij_{N'}}.$$

That is, $\sigma(N)_i$ is the value we use to approximate the $i$th row sum at the parent node of $N$. The cost of an optimal tree rooted at node $N$ that is associated with the matrix $\mathbf{X}_{I,\cdot}$ is given by the following equation:

$$\mathrm{c}(N, I) = \begin{cases} \min_{t,j} \Big\{ \mathrm{c}\big(l(N), L_I(t,j)\big) + \mathrm{c}\big(r(N), R_I(t,j)\big) \Big\} & \text{if } \mathbf{X}_{I\cdot} \text{ should be spilt,} \\ \min_j \Big\{ \sum_{i \in I} \big(\sigma(N)_i + \mathbf{X}_{ij} - \mathbf{r}_i\big)^2 \Big\} & \text{otherwise.} \end{cases} \tag{4}$$

The cost of the tree $\mathcal{T}$ is $c(R, \{1, \ldots, n\})$, where $R$ is the root of $\mathcal{T}$ and $n$ the number of rows in $\mathbf{X}$. A node $N$ should not be split if $|P(N)| + 1 = h$, but also

in the case where the cost of splitting is larger than not splitting. The latter condition implies, that even after we have split the node $N$ we should check if a solution where $N$ is not split has a lower cost.

The optimal tree for a given matrix $\mathbf{X}$ is thus found by considering all possible splits (defined by $t$ and $j$), and finding the optimal trees for the sub-matrices $\mathbf{X}_{L_I(t,j)}$ and $\mathbf{X}_{R_I(t,j)}$. If we require that the tree is balanced at node $N$, we only have to optimize over $j$, because $t$ is implicitly given by the requirement that $|L_I(t,j)| = |R_I(t,j)|$. A rough outline for an exact algorithm for solving this restricted variant of the problem is shown in Algorithm 1.

**Algorithm 1.** exact-balanced-tree

Input: set of integers $I$

```
if I should not be split then
    return "cost of I"
end if
for j = 1 to number of columns in X do
    t ← median of column X_{I,j}
    c_j ← exact-balanced-tree( L_I(t,j) ) + exact-balanced-tree( R_I(t,j) )
end for
return min{c}
```

Assuming that $|I| = n$, and that $\mathbf{X}$ has $m$ columns, we can express the running time of Algorithm 1 with the recurrence

$$T(n) \leq 2mT(\frac{n}{2}) + cmn. \tag{5}$$

This holds since we make $2m$ recursive calls to exact-balanced-tree with inputs of size $n/2$, and we must find the median (an $O(n)$ operation) $m$ times. Using the Master method [9] it is easy to show that the running time of Algorithm 1 (in terms of $n$) is of order $O(n^{\log n})$. This makes exact-tree a quasi-polynomial time algorithm. It is slower than polynomial, but not exponential. Also note that the exact solution with unbalanced trees is even harder, because we also have to optimize over $t$. Of course this simple analysis does not rule out the existence of efficient solutions for Equation 4, but it suggests that they are not trivial to devise. Therefore, our aim in this paper is not to find trees that are optimal in terms of Equation 4, but instead we propose a heuristic for finding good trees using a greedy algorithm.

## 4 An Inexact Greedy Algorithm

In this section we present a greedy algorithm for learning a tree $\mathcal{T}$ given the matrix $\mathbf{X}^{D,S}$. The algorithm contains parts of which it is not straightforward to analyze the running time, but we will argue in Section 4.2 that if the splits made by the algorithm are not very imbalanced (that is, the sizes of $L_I(t,j)$ and $R_I(t,j)$ are not very different), it's running time is of order $O(n^3)$.

**Algorithm 2.** build-sv-tree

Input: matrix $\mathbf{X}$, vector $\mathbf{r}$, column index $j$
Output: the triple $(N, \mathcal{T}_l, \mathcal{T}_r)$

**if** stopping-condition-met($\mathbf{X}$, $\mathbf{r}$) **then**
  **return** $((j, -1), \emptyset, \emptyset)$
**end if**
$(t, j_l, j_r) \leftarrow$ find-optimal-split( $\mathbf{X}$, $\mathbf{r}$, $j$)
$L \leftarrow \{i \ : \ \mathbf{X}_{ij} \leq t\}$
$R \leftarrow \{i \ : \ \mathbf{X}_{ij} > t\}$
$\mathcal{T}_l \leftarrow$ build-sv-tree( $\mathbf{X}_{L\cdot}$, $(\mathbf{r}_L - \mathbf{X}_{Lj_l})$, $j_l$ )
$\mathcal{T}_r \leftarrow$ build-sv-tree( $\mathbf{X}_{R\cdot}$, $(\mathbf{r}_R - \mathbf{X}_{Lj_r})$, $j_r$ )
**return** $((j, t), \mathcal{T}_l, \mathcal{T}_r)$

### 4.1 Algorithm Description

On a high level the greedy algorithm is similar to the exact one discussed above. Each call to the algorithm finds a split of $\mathbf{X}$ that is in some sense optimal, and then recursively processes the two resulting sub-matrices. However, now we consider a somewhat different notion of optimality of a split induced by $t$ and $j$. With the exact algorithm an optimal split is defined in terms of the optimal costs of the resulting sub-matrices, as can be seen in Equation 4. In particular, Alg. 1 computes the optimal subtrees rooted at a node $N$ to evaluate the cost of splitting $\mathbf{X}$ along the $j$th column. The greedy algorithm takes a "myopic" approach. It only considers subtrees of size 1, meaning that they consist of *only one leaf node* each. In other words, we compute the cost of a split at node $N$ under the restriction that $l(N)$ and $r(N)$ will not be split further, even if these actually are split later.

For any vector $\mathbf{x}$, we have $||\mathbf{x}|| = \mathbf{x}^{\mathrm{T}}\mathbf{x}$. Let $L_I(t, j)$ and $R_I(t, j)$ be defined as above, and suppose for a moment that we are given the column $j$. We define the cost of a "greedy split" as

$$\mathrm{c}(t, j_l, j_r) = ||\mathbf{X}_{L_I(t,j),j_l} - \mathbf{r}_{L_I(t,j)}|| + ||\mathbf{X}_{R_I(t,j),j_r} - \mathbf{r}_{R_I(t,j)}||, \tag{6}$$

where $j_l$ and $j_r$ are the columns that are associated with the left and right leafs, respectively. This is almost the same as Equation 5 if we assume that the child nodes of the current node are not split further. Another difference is that we are only optimizing over $t$, this time the parameter $j$ was assumed to be given a priori. This may seem strange at first, but it is in fact quite natural: When splitting a node $N$ using the cost in Equation 6, we must find the parameters $j_l$ and $j_r$. These are used as the splitting-column when processing $l(N)$ and $r(N)$. The parameter $j_N$ is thus already found when splitting the parent node of $N$. Pseudo-code of the greedy heuristic incorporating this principle is shown in Algorithm 2.

To find the optimal split, we must thus find the threshold $t$, and the column indices $j_l$ and $j_r$. In theory there are $O(nm^2)$ different combinations for an input matrix $\mathbf{X}$ of size $n \times m$. Iterating over all of these is obviously not scalable.

**Algorithm 3.** find-optimal-split

Input: matrix $\mathbf{X}$, vector $\mathbf{r}$, column index $j$
Output: the triple $(t, j_l, j_r)$

1: $t \leftarrow$ random element of $\mathbf{X}_{\cdot j}$
2: $c \leftarrow \infty$
3: **while** cost $c$ is decreasing **do**
4: $\quad (j_l, j_r, c) \leftarrow$ optimize-columns( $\mathbf{X}$, $\mathbf{r}$, $j$, $t$ )
5: $\quad (t, c) \leftarrow$ optimize-threshold( $\mathbf{X}$, $\mathbf{r}$, $j$, $j_l$, $j_r$ )
6: **end while**
7: **return** $(t, j_l, j_r)$

**Algorithm 4.** optimize-columns

Input: matrix $\mathbf{X}$, vector $\mathbf{r}$, column index $j$, threshold $t$
Output: triple $(j_l, j_r, c)$

1: $L \leftarrow \{i : \mathbf{X}_{ij} \leq t\}$
2: $R \leftarrow \{i : \mathbf{X}_{ij} > t\}$
3: $j_l \leftarrow \text{argmin}_{j'} ||\mathbf{X}_{Lj'} - \mathbf{r}_L||_2$
4: $j_r \leftarrow \text{argmin}_{j'} ||\mathbf{X}_{Rj'} - \mathbf{r}_R||_2$
5: $c \leftarrow ||\mathbf{X}_{Lj_l} - \mathbf{r}_L|| + ||\mathbf{X}_{Rj_r} - \mathbf{r}_R||$
6: **return** $(j_l, j_r, c)$

However, we can resort to a local optimization technique, that resembles the EM-algorithm and is of considerably lower, albeit unknown, complexity. Note that finding $j_l$ and $j_r$ is easy if we are given the value of $t$. In that case we simply have to try out the $O(m)$ different alternatives. Likewise, given $j_l$ and $j_r$ it is easy to find an optimal value for $t$ by checking all $n$ possible choices. We can thus alternatively solve for $j_l$ and $j_r$ given $t$, then solve for $t$ given $j_l$ and $j_r$, and continue this until convergence. The method is outlined in Algorithm 3. Algorithms 4 and 5 show the pseudo-code for the two subroutines in find-optimal-split.

### 4.2 Analysis of the Greedy Algorithm

To analyze the complexity of build-sv-tree, we resort again to the Master theorem [9]. This time the recurrence is

$$T(n) \leq 2T(\frac{n}{b}) + \underbrace{ch(n,m)(m+1)n}_{q(n)},$$

where $h(n, m)$ is a function that bounds the number of iterations of the loop on lines 3–5 in Algorithm 3. The optimize-columns algorithm (Alg. 4) runs in time $O(nm)$, while optimize-threshold (Alg. 5) can be implemented to run in time $O(n)$. Since we are not enforcing the split to be balanced, we use the parameter $b$ in the analysis. Also, we assume that $m = n$, which corresponds to the case where all training examples end up as support vectors and the same data is

**Algorithm 5.** optimize-threshold

Input: matrix $\mathbf{X}$, vector $\mathbf{r}$, column indices $j$, $j_l$, $j_r$
Output: pair $(t^*, c^*)$

```
1: t* ← −1, c* ← ∞
2: for h = 1 to number of rows in X do
3:    t ← X_hj
4:    L ← {i : X_ij ≤ t}
5:    R ← {i : X_ij > t}
6:    c ← ||X_{L j_l} − r_L|| + ||X_{R j_r} − r_R||
7:    if c < c* then
8:       c* ← c, t* ← t
9:    end if
10: end for
11: return (t*, c*)
```

used to find $\mathcal{T}$. This way $q(n)$ is of order $n^{2+\gamma}$, where $\gamma$ is dependant on the complexity of the function $h$. That is, if $h$ was of order $O(n)$ we would have $\gamma = 1$, for example.

To use the Master theorem we must compare $q(n)$ with the function $n^{\log_b 2+\epsilon}$. There are three cases based on the value of $\epsilon$ that result in different running times for the algorithm. More precisely, we study for what values of $b$, $\gamma$, and $\epsilon$ it holds that $n^{2+\gamma} = n^{\log_b 2+\epsilon}$. Solving this for $\epsilon$ gives

$$\epsilon = \frac{(2+\gamma)\log_2 b - 1}{\log_2 b}. \tag{7}$$

Now we consider three possible cases for $\epsilon$, that is, $\epsilon < 0$, $\epsilon = 0$, and $\epsilon > 0$. By setting Equation 7 equal to zero and simplifying we obtain $\log_2 b = (2+\gamma)^{-1}$, or $b = 2^{\frac{1}{(2+\gamma)}}$. This gives us a relationship between $b$ and $\gamma$ that we can use to distinguish the different cases of the Master theorem:

- Case 1: $\epsilon < 0 \Leftrightarrow b < 2^{\frac{1}{(2+\gamma)}}$, which implies $T(n) = \Theta(n^{\log_b 2})$.
- Case 2: $\epsilon = 0 \Leftrightarrow b = 2^{\frac{1}{(2+\gamma)}}$, which implies $T(n) = \Theta(n^{\log_b 2} \log_2 n)$.
- Case 3: $\epsilon > 0 \Leftrightarrow b > 2^{\frac{1}{(2+\gamma)}}$, which implies $T(n) = \Theta(q(n))$ if the regularity condition holds as well.

There is thus a threshold for $b$, the value of which will depend on $\gamma$. Of course the actual value of $b$ will vary, as different inputs lead to different splits. Some of these will be more balanced ($b$ close to 2) than others ($b$ close to 1). The value of $\gamma$ depends on the convergence of the optimization in Algorithm 3. We do not know the exact rate of convergence in terms of $n$ and $m$. However, based on empirical observations it is realistic to assume that for most inputs we have $0 < \gamma \leq 1$. I.e., $h(n, m)$ is at most linear in $n$, but possibly sublinear.

Letting $\gamma = 1$ gives us the threshold $2^{1/3} \approx 1.26$. This corresponds to an unbalanced split where roughly 80 percent of the input end up in the same subtree. Case 1 of the Master theorem concerns the case where the split is

even more unbalanced, while Case 3 covers more balanced splits. To analyze the deviation of $b$ from the threshold $2^{1/3}$, we introduce a parameter $\beta$ and let $b = 2^{1/3+\beta}$, so that $\beta < 0$, $\beta = 0$, and $\beta > 0$ correspond to the cases 1, 2, and 3, respectively. Note that for the degree of the polynomial in cases 1 and 2 we obtain $\log_b 2 = (\frac{1}{3} + \beta)^{-1}$.

In Case 2 ($\beta = 0$) the running time of Alg. 2 is therefore simply $\Theta(n^3 \log_2 n)$. For Case 1 ($\beta < 0$) we obtain a running time of $\Theta(n^{\frac{1}{(1/3+\beta)}})$, which is already $\Theta(n^6)$ if we let $\beta = -\frac{1}{6}$. This makes sense as very unbalanced splits are bound to slow down the algorithm considerably. And even with balanced splits represented by Case 3 ($\beta > 0$), the running time is bounded by $\Theta(q(n))$[1], where $q(n)$ is a polynomial of degree 3. That is, given that we assumed $\gamma = 1$, this type of analysis results in a cubic running time of Algorithm 2 even in the "best" case. However, it is still a considerable improvement over the quasi-polynomial exact algorithm discussed in Section 3.2.

### 4.3 Scaling the Columns of $\mathbf{X}^{D,S}$

We can make a small modification to the algorithm that should improve it's performance. Instead of approximating $\mathbf{r}^{D,S}$ directly with the contents of $\mathbf{X}^{D,S}$, we can scale the values on the columns to minimize the difference to $\mathbf{r}^{D,S}$. More precisely, consider the lines 3 and 4 in Algorithm 4. The optimal columns $j_l$ and $j_r$ are found by minimizing the error $||\mathbf{X}_{\cdot j} - \mathbf{r}||$ subject to $j$. We add an additional parameter $c$, so that the new optimization problem becomes $\min_{j,c} ||c\mathbf{X}_{\cdot j} - \mathbf{r}||$.

Moreover, for a given $j$ it is easy to find the optimal $c$. To see this, recall that we have $||c\mathbf{X}_{\cdot j} - \mathbf{r}|| = (c\mathbf{X}_{\cdot j} - \mathbf{r})^{\mathrm{T}}(c\mathbf{X}_{\cdot j} - \mathbf{r})$. Setting the 1st derivative of this with respect to $c$ to zero gives

$$c = \frac{\mathbf{X}_{\cdot j}^{\mathrm{T}}\mathbf{r}}{\mathbf{X}_{\cdot j}^{\mathrm{T}}\mathbf{X}_{\cdot j}}.$$

This obviously also introduces a new parameter, $c_N$, to each node $N$ of $\mathcal{T}$, and the score of node $N$ for an unseen example $\mathbf{x} \in \Omega$ is now computed as $\mathrm{score}_N(\mathbf{x}) = c_N a_N g(\mathbf{x}, \mathbf{s}_N)$. The algorithm we use in the experiments makes use of this additional heuristic.

## 5 Empirical Evaluation

We continue with empirical results. Recall that our main motivation is to speed up classification. Hence we are mostly interested in cases where the support vector tree outperforms either a linear SVM or a sparse SVM.

More precisely, we use five methods in our experiments: a linear SVM, a nonlinear SVM using the RBF kernel, a support vector tree based on the nonlinear SVM, and two variants of the Cutting Plane Subspace Pursuit (cpsp)

[1] The regularity condition required by the theorem is also trivially satisfied for $b > 2^{1/3}$.

**Table 1.** Accuracies and number of support vectors of the various methods on different benchmark data sets. The best performing algorithm in the set {l-svm, sv-tree, cpsp(tr)} is indicated in bold.

| dataset | dummy | l-svm | svm | sv-tree | cpsp | cpsp(tr) |
|---|---|---|---|---|---|---|
| heart | 0.56 | 0.82 | 0.82 | 0.75 | 0.84 | **0.84** |
| | | | 60 | 8.2 | 9 | 9 |
| thyroid | 0.71 | 0.89 | 0.96 | **0.92** | 0.92 | 0.90 |
| | | | 13 | 9.3 | 10 | 10 |
| breastcancer | 0.70 | 0.70 | 0.71 | 0.70 | 0.73 | **0.73** |
| | | | 120 | 9.1 | 10 | 10 |
| waveform | 0.67 | 0.87 | 0.90 | 0.86 | 0.90 | **0.88** |
| | | | 229 | 9.5 | 10 | 10 |
| german | 0.70 | **0.76** | 0.75 | 0.70 | 0.76 | **0.76** |
| | | | 373 | 10.9 | 11 | 11 |
| image | 0.57 | 0.85 | 0.97 | **0.89** | 0.89 | 0.87 |
| | | | 213 | 11.9 | 12 | 12 |
| diabetis | 0.65 | **0.77** | 0.77 | 0.73 | 0.77 | **0.77** |
| | | | 249 | 10.0 | 11 | 11 |
| ringnorm | 0.50 | 0.75 | 0.98 | **0.95** | 0.84 | 0.74 |
| | | | 152 | 10.5 | 11 | 11 |
| splice | 0.51 | **0.83** | 0.89 | 0.68 | 0.86 | 0.77 |
| | | | 588 | 11.4 | 12 | 12 |
| twonorm | 0.50 | **0.97** | 0.98 | 0.91 | 0.98 | **0.97** |
| | | | 263 | 9.5 | 10 | 10 |
| banana | 0.54 | 0.55 | 0.89 | **0.88** | 0.86 | **0.88** |
| | | | 151 | 9.2 | 10 | 10 |

algorithm [16]. The first cpsp variant may use arbitrary basis vectors in the set $S$, while the second one is restricted to choose these from the training data in the style of a classical SVM. As in [16], we denote the 2nd variant by cpsp(tr). To learn the linear and nonlinear SVM we use LibSVM [7], while the cpsp algorithm is implemented in the svm-perf package [14,15,16]. To compare the methods we use standard benchmark data sets that are publicly available[2]. All cases are binary classification problems.

We consider the cpsp(tr) a more interesting comparison as cpsp, since our trees are also restricted to use only examples from the training data. It is obvious from the experiments in [16] and those shown here that the use of arbitrary basis vectors is in many cases beneficial. However, efficient implementations of this approach seem to be rather nontrivial to implement for arbitrary kernel functions as cpsp requires solving the pre-image problem [17]. While it is true that pre-images can be found even for structured objects such as strings [1] and graphs [2], the sv-tree can be seen as a more powerful approach as it is oblivious to the representation of the input examples given any suitable kernel function. For instance, svm-perf can only use RBF kernels with the cpsp algorithm. Hence

[2] `http://ida.first.fraunhofer.de/projects/bench/`

it is more interesting to see if the sv-tree can beat the linear SVM, and achieve at least the performance of the cpsp(tr) algorithm.

Instead of explicitly giving a maximum height for the support vector tree, we use a stopping criteria that is based on the size of the input. The call to stopping-condition-met on line 1 in Algorithm 2 returns true if the number of rows in $\mathbf{X}$ is less or equal to five. We could just as well use a stopping condition based on the height of the tree. However, considering the size of a node has the advantage that we do not split matrices with only a couple of rows, and conversely allow a node to split when there is still enough data to work with. With both SVMs and cpsp we use a grid-search to find good values for the regularization parameter $C$ and the parameter $\gamma$ used by the RBF kernel. Furthermore, to have an interesting comparison with the cpsp method, we set it's budget equal to the average number of kernel evaluations needed by the tree to classify a given test set.

Results for the accuracies and sizes of the model are given in Table 1. The reported numbers are averages over 20 disjoint training-test splits. The 2nd column shows the accuracy of a dummy classifier that simply assigns everything in the test data to the class that was more common in the training data. Of the studied algorithms the linear SVM is the simplest, and is the preferred choice if classification speed is an issue. Indeed, in a few cases the accuracy of l-svm is comparable with that of svm. When compared with the nonlinear SVM, the sv-tree has a lower accuracy with all data sets, which is to be expected. With the 'heart' and 'splice' data sets the sv-tree seems to especially have problems. But the sv-tree in fact outperforms l-svm and cpsp(tr) a number of times. This is the case with the 'thyroid', 'image', 'ringnorm', and 'banana' data sets. Also note that in many cases where cpsp(tr) outperforms sv-tree, the linear SVM outperforms cpsp(tr), and would hence be the method of choice to speed up classification. With the chosen stopping condition for the sv-tree, the number of kernel evaluations is with these data sets reduced by one order of magnitude.

## 6 Conclusion

We have presented an approach to speed up classification with kernel machines based on adaptive selection of basis vectors given the example to be classified. Despite the large body of existing literature on kernel machines this idea seems to be, to the best of our knowledge, novel. To quickly find the subset of basis vectors to be used in the decision rule, we propose the use of a binary tree, the support vector tree, that induces a disjoint partition of the feature space. To learn this tree we propose a greedy heuristic that can result in suboptimal trees but runs in polynomial time. Our experiments suggest that the support vector tree can in some cases outperform existing state-of-the-art algorithms for learning sparse SVMs.

It must be noted that the idea proposed here is not specific to kernel machines. The same approach can be employed also in case of ensemble classifiers. Even though the weak learners (or base classifiers) in general are fast to compute, there can be several hundreds of them. This can become a bottleneck for e.g. ensemble

method based document ranking functions. For instance in [6] the evaluation of the ensemble is interrupted when it becomes unlikely that the outputs of the remaining weak learners would significantly change the already computed score of the document being ranked. Our method could be applied as such in this setting by replacing kernel computations with evaluations of the weak learners.

Obvious future research concerns improved algorithms for finding trees with better accuracy. Alternatively we can consider other representations of the function $f$. One problem with the current approach is that we must first learn a kernel machine using some legacy algorithm. Instead of a post-processing algorithm, we can also devise algorithms that find a tree directly based on training data. Finally, a more detailed experimental evaluation of the current algorithm using large real world data sets is also of interest.

## References

1. Bakir, G., Weston, J., Schölkopf, B.: Learning to find pre-images. In: Advances in Neural Information Processing Systems (NIPS 2003), vol. 16 (2003)
2. Bakir, G., Zien, A., Tsuda, K.: Learning to find graph pre-images. In: Pattern Recognition, 26th DAGM Symposium, pp. 253–261 (2004)
3. Burges, C.: Simplified support vector decision rules. In: Machine Learning, Proceedings of the Thirteenth International Conference (ICML 1996), pp. 71–77 (1996)
4. Burges, C.: A tutorial on support vector machines for pattern recognition. Knowledge Discovery and Data Minnig 2(2), 121–167 (1998)
5. Burges, C., Schölkopf, B.: Improving the accuracy and speed of support vector machines. In: Advances in Neural Information Processing Systems, NIPS, vol. 9, pp. 375–381 (1996)
6. Cambazoglu, B., Zaragoza, H., Chapelle, O.: Early exit optimizations for additive machine learned ranking systems. In: Proceedings of the Third International Conference on Web Search and Web Data Mining, WSDM 2010 (to appear, 2010)
7. Chang, C.-C., Lin, C.-J.: LIBSVM: a library for support vector machines (2001), `http://www.csie.ntu.edu.tw/~cjlin/libsvm`
8. Chen, J.-H., Chen, C.-S.: Reducing SVM classification time using multiple mirror classifiers. IEEE Transactions on Systems, Man, and Cybernetics – Part B: Cybernetics 34(2), 1173–1183 (2004)
9. Cormen, T., Leiserson, C., Rivest, R., Stein, C.: Introduction to Algorithms, 2nd edn. The MIT Press, Cambridge (2001)
10. Dekel, O., Singer, Y.: Support vector machines on a budget. In: Advances in Neural Information Processing Systems, Proceedings of the Twentieth Annual Conference on Neural Information Processing Systems, vol. 19, pp. 345–352 (2006)
11. Downs, T., Gates, K.E., Masters, A.: Exact simplification of support vector solutions. Journal of Machine Learning Research 2, 293–297 (2001)
12. Garey, M.R., Johnson, D.S.: Computers and Intractability – A Guide to the Theory of NP-Completeness. Freeman, New York (1979)
13. Joachims, T.: Optimizing search engines using clickthrough data. In: Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining, pp. 133–142 (2002)
14. Joachims, T.: A support vector method for multivariate performance measures. In: Proceedings of the International Conference on Machine Learning (ICML), pp. 377–384 (2005)

15. Joachims, T.: Training linear svms in linear time. In: Proceedings of the ACM Conference on Knowledge Discovery and Data Mining (KDD), pp. 217–226 (2006)
16. Joachims, T., Yu, C.-N.J.: Sparse kernel svms via cutting-plane training. Machine Learning 76, 179–193 (2009)
17. Kwok, J.T., Tsang, I.W.: The pre-image problem in kernel methods. IEEE Transactions on Neural Networks 15(6), 1517–1525 (2004)
18. Lodhi, H., Saunders, C., Shawe-Taylor, J., Cristianini, N., Watkins, C.: Text classification using string kernels. Journal of Machine Learning Research 2, 419–444 (2002)
19. Nair, P.B., Choudhury, A., Keane, A.J.: Some greedy learning algorithms for sparse regression and classification with mercer kernels. Journal of Machine Learning Research 3, 781–801 (2002)
20. Osuna, E., Girosi, F.: Reducing the run-time complexity in support vector machines. In: Advances in Kernel Methods: Support Sector Learning. MIT Press, Cambridge (1999)
21. Teo, C.H., Vishwanathan, S.: Fast and space efficient string kernels using suffix arrays. In: Proceedings of 23rd International Conference on Machine Learning, pp. 929–936 (2006)
22. Tipping, M.E.: Sparse bayesian learning and the relevance vector machine. Journal of Machine Learning Research 1, 211–244 (2001)
23. Vapnik, V.: The Nature of Statistical Learning Theory. Springer, Heidelberg (1995)
24. Wu, M., Schölkopf, B., Bakir, G.: A direct method for building sparse kernel learning algorithms. Journal of Machine Learning Research 7, 603–624 (2006)

# Author Index