LNCS 4009

Moshe Lewenstein

Gabriel Valiente (E

Combinato

Pattern Ma

17th Annual Symposium, C
Barcelona, Spain, July 200
Proceedings

Lecture Notes in Computer Science 4009

Moshe Lewenstein   Gabriel Valiente (Eds.)

Combinatorial Pattern Matching

17th Annual Symposium, CPM 2006
Barcelona, Spain, July 5-7, 2006
Proceedings

Springer

Volume Editors

Moshe Lewenstein
Bar-Ilan University
Department of Computer Science
Ramat Gan 52900, Israel
E-mail: moshe@cs.biu.ac.il

Gabriel Valiente
Technical University of Catalonia
Department of Software
08034 Barcelona, Spain
E-mail: valiente@lsi.upc.edu

Library of Congress Control Number: 2006927372

CR Subject Classification (1998): F.2.2, I.5.4, I.5.0, I.7.3, H.3.3, J.3, E.4, G.2.1, E.1

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

ISSN 0302-9743
ISBN-10 3-540-35455-7 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-35455-0 Springer Berlin Heidelberg New York

Springer is a part of Springer Science+Business Media

springer.com

Printed in Germany

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 11780441 06/3142 5 4 3 2 1 0

Preface

This volume contains the papers presented at the 17th Annual Symposium on Combinatorial Pattern Matching (CPM 2006) held at the Technical University of Catalonia in Barcelona, Spain, on July 5–7, 2006. They were selected from 88 submissions. Each submission was reviewed by at least three Programme Committee members. The committee decided to accept 33 papers. The programme also included three invited talks, by Amihood Amir (Asynchronous pattern matching), Eran Halperin (SNP and haplotype analysis: Algorithms and applications), and Steven Skiena (News and blog analysis with Lydia).

All papers presented at the conference are original research contributions on combinatorial pattern matching algorithms, indexing data structures, data compression, and applications in molecular biology such as phylogenetic reconstruction, motif search, and RNA and DNA structural analysis and prediction.

The meeting was preceded by a Summer School on Combinatorial Pattern Matching on July 4, 2006, with tutorials by Ricardo Baeza-Yates (Web searching), Moshe Lewenstein (Pattern matching with mismatches), and Alfonso Valencia (Introduction to computational biology).

The Annual Symposium on Combinatorial Pattern Matching started in 1990, and has since taken place every year. Previous CPM meetings were held in Paris, London, Tucson, Padova, Asilomar, Helsinki, Laguna Beach, Aarhus, Piscataway, Warwick, Montreal, Jerusalem, Fukuoka, Morelia, Istanbul, and Jeju Island. Selected papers from the first meeting appeared in volume 92 of *Theoretical Computer Science*, from the 11th meeting in volume 2 of *Journal of Discrete Algorithms*, from the 12th meeting in volume 146 of *Discrete Applied Mathematics*, and from the 14th meeting in volume 3 of *Journal of Discrete Algorithms.* Starting with the 3rd meeting, proceedings of all meetings were published in the LNCS series, volumes 644, 684, 807, 937, 1075, 1264, 1448, 1645, 1848, 2089, 2373, 2676, 3109, 3537, and 4009.

The whole submission and review process, as well as production of this volume, was carried out with the help of the EasyChair system. The conference was sponsored by the Technical University of Catalonia and by the Spanish Ministry of Education and Science.

April 2006

Moshe Lewenstein
Gabriel Valiente

Conference Organization

Programme Chairs

Moshe Lewenstein
Gabriel Valiente

Programme Committee

Ricardo Baeza-Yates
Paolo Ferragina
Leszek Gąsieniec
Raffaele Giancarlo
Roberto Grossi
Dan Gusfield
Tao Jiang
Gad M. Landau
Thierry Lecroq
Ming Li
Stefano Lonardi
Laxmi Parida
Ayumi Shinohara
Jens Stoye
Esko Ukkonen
Kaizhong Zhang
Michal Ziv-Ukelson

Local Organization

Gemma Casas
Liliana Félix
David Garcia
Xavier Messeguer
Marta Moreno
Roman Roset
Romina Royo
Gabriel Valiente (Chair)

External Reviewers

Saïd Abdeddaïm
Gabriela Alexe
José Augusto Amgarten Quitzau
Hideo Bannai
Saugata Basu
Omer Berkman
Paola Bonizzoni
Brona Brejova
Gerth Stølting Brodal
Dan Brown
Jeremy Buhler
Pascal Caron
Xin Chen
Shihyen Chen
Matteo Comin
Maxime Crochemore
Yoan Diekmann
Zihong Ding
Shiri Dori
Alon Efrat
Kimmo Fredriksson
Arie Freund
Zheng Fu
Nicola Galesi
Lilia Greenenko
Antonio Gullì
Eran Halperin
Angele Hamel
Miki Hermann
Danny Hermelin
Jan Holub
Peter Husemann
Shunsuke Inenaga
Wojciech Jawor
Carmel Kent
Shahar Keret
Marcos Kiwi
Stefan Kurtz
Juha Kärkkäinen
Arnaud Lefebvre
Liat Leventhal
Guojun Li
Yu Lin
Chaim Linhart
Lan Liu
Mercè Llabrés
Alex Lopez-Ortiz
Antoni Lozano
Veli Mäkinen
Giovanni Manzini
Julia Mixtacki
Gonzalo Navarro
Mia Persson
Gemma Piella
Nadia Pisanti
Hendrik Purwins
Mathieu Raffinot
Sven Rahmann
Jairo Rocha
Oleg Rokhlenko
Francesc Rosselló
Kunihiko Sadakane
Paul Sant
Klaus-Bernd Schürmann
Marinella Sciortino
Nira Shafrir
Baozhen Shan
Yun Song
Kristian Stevens
Dimitrios M. Thilikos
Hélène Touzet
Lars Ulveland
Vladimir Vacic
Balaji Venkatachalam
Lusheng Wang
Ydo Wexler
Prudence Wong
Yonghui Wu
Yufeng Wu
Jing Xiao
Lei Xin
Sheng Yu
Jie Zheng

Table of Contents

Session 1. Data Structures

Session 2. Indexing Data Structures

Session 3. Probabilistic and Algebraic Techniques

Session 4. Applications in Molecular Biology I

Session 5. String Matching I

Session 6. Applications in Molecular Biology II

Session 7. Applications in Molecular Biology III

Session 8. Data Compression

Session 9. String Matching II

Session 10. Dynamic Programming

Asynchronous Pattern Matching

Amihood Amir

Department of Computer Science, Bar-Ilan University, Ramat-Gan 52900, Israel
and College of Computing, Georgia Tech, Atlanta, GA 30332-0280
+972 3 531-8770
amir@cs.biu.ac.il

Abstract. This paper introduces a new pattern matching model that has been gaining importance recently, that of *Asynchronous Pattern Matching*. Traditional pattern matching has assumed the possibility of errors in the data *content*. We present motivation from text editing, computational biology, and computer architecture, that points to a new paradigm – where the errors occur in the *address*. It turns out that there are differences in techniques, complexities, and tools between the two different models, making it important to recognize their differences.

We motivate and define the new model and present some problems that are worth pursuing.

1 Motivation

Historically, approximate pattern matching grappled with the challenge of coping with errors in the data. The traditional *Hamming distance* problem assumes that some elements in the pattern are erroneous, and one seeks the text locations where this number of errors is small enough [23, 18, 7], or efficiently calculating the Hamming distance at every text location [1, 21, 7]. The *edit distance* problem adds to the assumption that some elemnts of the text are deleted, or that noise is added at some text locations [24, 15]. Indexing and dictionary matching under these errors has also been considered [19, 16, 26, 14].

Implicit in all these problems is the assumption that there may indeed be errors in the **content** of the data, but the **order** of the data is inviolate. Data may be lost or noise may appear, but the relative position of the symbols is unchanged. Data **does not move around**. Even when *don't cares* were added [17], when non-standard models were consideredbak:93,muthu-ramesh:iac95,aaclp:03 the order of the data was assumed to be ironclad.

Nevertheless, some non-conforming problems have been gnawing at the walls of this assumption. Below we introduce some examples for this different phinomenom.

Text Editing: Even in the traditional core of pattern matching motivation - text editing - there crop up some problems dealing with address, rather than content, error. The *swap* error, motivated by the common typing error where two adjacent symbols are exchanged [25, 4], 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

M. Lewenstein and G. Valiente (Eds.): CPM 2006, LNCS 4009, pp. 1–10, 2006.

errors, in that it could be added to the other edit operations and solved with the same dynamic programming [25]. However, when isolated, it turned out to be surprisingly simple to handle. Indeed, the minimum swap distance between a given pattern of length m and every substring of length m in a given text of length n, can be found in time $O(n \text{ polylog } m)$ [6]. This scarcely seems to be the case for indels or mismatch errors.

Computational Biology: Recently, the advent of computational biology has added more problems of order error to our repertoire. In evolution, one envisions a whole piece of genome to "detach" and "reconnect" in a different location, or two pieces of genome to "exchange" places. These phenomena, of course, are assumed to take place simultaneously with traditional data content errors, however, their nature is **rearrangement** of the data, rather than corruption of its contents.

It turns out that the overall problem of adding these new rearrangement operators to the content changing operators is extremely difficult. Thus more simplified problems were considered in the literature. The rearrangement operators were isolated and handled separately. Reversals [11], transpositions [8], and block interchanges [13] were explored. The edit distance problem under these new operations is still too difficult, therefore the *sorting permutation* version of these problems was researched.

This research direction led to interesting paths. First, the tools and techniques used were different from the traditional pattern matching tools. The results also seem more varied. The *sorting by reversal problem* is $\mathcal{NP}$-hard [12]. It is still open whether the *sorting by transposition problem* can be efficiently solved deterministically. Christie [13] gives an $O(n^2)$ algorithm for the *sorting by block interchange problem.*

Computer Architecture: In computer architecture, address errors are of no less concern that content errors [20]. It is by no means taken for granted that when seeking a word from a given address, no errors will occurr in the address bits. This causes the concern with redundancy bits, checksum bits, error detection and correction codes, and communication protocols.

From a purely theoretical point of view, it would be interesting to consider searching where address errors are not corrected at all (say because of applications with an extremely high cost of transmission, e.g. because of transmission in deep space). What are the types of uncorrected address errors that can still be reasonable handled by a search application?

In a recent paper [3], for the first time, this different pattern matching paradigm, that of **errors in the order** rather than error in the content of the data, was explicitely identified and formalized. The advantages in formalizing this paradigm are:

1. Identifying the types of problems and techniques required, rather than than re-inventing ad-hoc solutions.
2. Understanding the theoretical underpinnings of the problem.
3. Generalizing to other possible rearrangements and possibly providing more general solutions.

Two different general directions of research are possible. The first is the need to consider appropriate *distance measures*. The error in content measures are not necessarily meaningful in these circumstances. We will consider some generic error distances, such as minimum L_1 and L_2 distance on the address of the data. We also illustrate the fact that more specific distance measures are necessary for specific applications.

Another possible direction is considering different address bit errors and efficient methods of approximate pattern matching under address errors.

It is exciting to point out that budding research in this area required some techniques that are totally new to pattern matching. This reinforces the realization that this new model is needed, as well as gives hopes to new research directions and paths in the field of pattern matching.

2 Problem Types

The pattern matching community usually handles problems in the following form:

INPUT: Text T of length n and pattern P of length m over alphabet Σ. Matching relation R or distance metric metric d.

OUTPUT: Denote the suffix of T starting at location i as T_i. Denote the appropriate prefix of T_i (appropriate in the sense that its length matches the length of P) as T_i^{pre}.

Output all text locations i where T_i^{pre} matches P under R, or where $d(T_i^{pre}, P)$ is sufficiently small.

In the classic exact matching problem, R is equality and the length of T_i^{pre} is m. In the case of Hamming distance, $d(A, B)$ is the number of mismatches between A and B, with A and B being equal-length strings. Again, the length of T_i^{pre} is m.

It should be noted that many pattern matching problems become trivial if the text and pattern are of *equal* length. Both exact matching and Hamming distance can be immediately solved in linear time for $n = m$. It is also clear that any problem that can be solved for $n = m$ in time $O(f(m))$ can be solved for $n \neq m$ in time $O(nf(m))$.

Another interesting simplification could be to assume that every alphabet symbol occurrs only **once** in the pattern. This means that every two pattern locations have different symbols. This is equivalent to assuming that the pattern is $1, 2, 3, ..., m$.

Even if we take $n \neq m$, both the exact matching problem and the Hamming distance problem can be immediately solved in time $O(n)$ if this condition is assumed.

As we will see, some of the asynchronous matching problems are $\mathcal{NP}$-hard even under the limiting assumptions above. In order to differentiate between the different assumptions, let us agree on a common nomenclature.

Notation

- Denote a problem where the pattern is of the form where every symbol appears exactly once, a *permutation problem.* We will refer to the case where a symbol may appear more than once as a *symbol repetition problem.*
- Denote a problem where the text and pattern sizes are equal, an *equal length problem.* We will refer to the case where $n > m$ as the *string matching problem.*

Of course, we may combine the conditions, i.e. we may consider the permutation version of a string matching problem, or the equal length version of a symbol repetition problem.

3 Metrics

Even a general purpose metric must be based on some assumptions of what causes the errors. Thus, the Hamming distance assumes that the only error are changes to the data, but no new elements can be introduced nor any data lost. The Levenshtein distance assumes that data may be inserted and deleted.

Similarly, when it comes to address errors, the application is the driving force behind the general metric. The transposition operation leads to a metric that counts the number of pairs that interchange. Another option is to count the distance that symbols need to travel in order to arrive at their destination. Finally, it is possible to combine the two, i.e. assume that the rearrangement operation is indeed an exchenge, yet count the distance needed to effect the transition.

We follow [3] in the definition of *rearrangement systems*, and the introduction of the metrics that were considered.

3.1 Rearrangement Distances

Consider a set A and let x and y be two m-tuples over A. We wish to formally define the process of converting x to y through a sequence of *rearrangement* operations. A *rearrangement operator* π is a function $\pi : [0..m-1] \rightarrow [0..m-1]$, with the intuitive meaning being that for each i, π 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 π_j's. We say that s *converts* x *into* y if for any $i \in [0..n-1]$, $x_i = y_{\pi_s(i)}$. That is, y is obtained from x by moving elements according to the designated sequence of rearrangement operations.

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

$$d_{\mathcal{R}}(x, y) = \min\{w(s) | s \text{ from } \mathcal{R} \text{ converts } x \text{ to } y \ \}$$

If there is no sequence that converts x to y then the distance is ∞.

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 i such that $d_{\mathcal{R}}(P, T^{(i)})$ is minimal.

The ℓ_1 and ℓ_2 Rearrangement Distances. The simplest set of rearrangement operations allows any element to be inserted at any other location. Under the ℓ_1 *Rearrangement System*, the cost of such a rearrangement is the sum of the distances the individual elements have been moved. We call the resulting distance the ℓ_1*Rearrangement Distance.* In the ℓ_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.[1] We call the resulting distance the ℓ_2*Rearrangement Distance.*

In [3] it was proven that:

Theorem 1. *For T and P of sizes n and m respectively $(m \leq n)$, the ℓ_1 Rearrangement Distance can be computed in time $O(m(n-m+1))$. For the permutation version, the distance can be computed in time $O(n)$.*

Interestingly, the ℓ_2 distance can be computed much more efficiently:

Theorem 2. *[3] For T and P of sizes n and m respectively $(m \leq n)$ the ℓ_2 Rearrangement Distance can be computed in time $O(n \log m)$.*

The Interchange Distances. Consider the set of rearrangement operators were in each operation the location of exactly two entries can be interchanged. The cost of a sequence is the total number of interchanges. We call the resulting distance the *interchanges distance.* Again in [3] it was shown:

Theorem 3. *For T and P of sizes n and m, respectively $(m \leq n)$, the permutation version of the interchanges distance problem can be computed in time $O(m(n-m+1))$.*

This situation is one where the permutation requirement is extremely necessary. In [5] it was shown that:

Theorem 4. *For T and P of sizes n and m, respectively $(m \leq n)$, the symbol repetition version of the interchanges distance problem is $\mathcal{NP}$-hard.*

Next consider the case were multiple pairs can be interchanged in parallel, i.e. in any given step an element can participate in at most one interchange. The cost of a sequence is the number of parallel steps. Call the resulting distance the *parallel interchanges distance*, denoted by $d_{p\text{-}interchange}(\cdot, \cdot)$. The following surprising result was shown in [3]:

[1] For simplicity of exposition we omit the square root usually used in the ℓ_2 distance. This does not change the complexity, since the square root operation is monotone, and can be computed at the end.

Theorem 5. *For any two tuples 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 the following was proven [3]:

Theorem 6. *For T and P of sizes n and m respectively ($m \leq n$), if there are k distinct entries in P, then the parallel interchanges distance can be computed deterministically in time $O(k^2 n \log m)$.*

Theorem 7. *For T and P of sizes m and n respectively ($m \leq n$), the parallel interchanges distance can be computed randomly in expected time $O(n \log m)$.*

In [5] the following hybrid metric was introduced: In this rearrangement system the operation is still an interchange, but the cost is not the number of operations. Instead, each interchange has a *weight*, and the cost of a sequence of interchanges is the sum of these interchanges weights. We define the weight of an interchange of elements at positions i and j to be $|i-j|$. This definition of the weight reflects that interchanges of close elements are preferred. Given a text T and a pattern P of sizes n and m, respectively, ($m \leq n$) the *weighted-interchange distance problem* is to find the text location closest to the pattern under the weighted-interchange distance.

The following was proven in [5]:

Theorem 8. *There exist an $O(m(n-m+1))$ algorithm that solves the weighted-interchange distance problem for the symbol repetition version in the string matching paradigm.*

Length-weighted genome rearrangements were recently claimed to be biological meaningful and preferred over the traditional assumption giving each operation a unit cost (see [10], [9]). The weighted-interchange defined above was inspired by these claims. Actually, even in the regular sorting situation the unit-cost model is not completely defensible. On the contrary, it makes sense to assume that interchanging far elements costs more than interchanging close elements. It is interesting to point out that, similar to [9], [5] found out that the weighted version of the problem is polynomial in contrast with the non-weighted version. In fact, these results together with [9] might indicate a general phenomenon about length-weighted distances that should be further studied.

4 Address Errors

In this section we suggest another broad class of location errors wherein the *names* of the locations have been altered. We call this type of errors *renaming errors*, defined as follows. A string $x \in A^m$ can be viewed as a set of pairs (*address*, *value*). A *renaming* π gives "new names" to the addresses. Formally, a renaming is a function $\pi : [1..m] \rightarrow [1..m]$. Under the renaming π, the string

x is converted into a new set of pairs, where the pair (a, v) is converted into the pair $(\pi(a), v)$. A *renaming scheme* is a set Π of renamings. Formally, any set of functions on $[1..m]$ can constitute a renaming scheme. In [2] the renaming schemes studied were such that the renamings all have some well-defined structure, transforming one naming convention to another. Specifically, they consider renaming schemes which arise from a process of flipping some or all of the bits in the binary representation of $[1..m]$.

In practice, renaming errors may arise in situations where the text and the pattern are generated by two different systems, which may use different naming conventions. Alternatively, renaming errors may result 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). Finally, renaming errors may actually not constitute an error, but rather represent different legitimate ways to order the given set of elements.

4.1 Address Bit Flips Errors

In this section we focus on renaming schemes resulting from address bit errors. The conventional addressing model assumes the user puts an address in the address register. From there the address follows an address bus to the desired location. We are assuming no redundacy bits, and no checksums nor error detection and correction codes.

Various types of faults may be considered:

1. A faulty bit consistently flips the value put there. If the in value is 0 the out value is 1 and vice versa.
2. A faulty bit may flip the value put there or may not. The non-faulty bits always output the value put in. This is a common phenomenon, caused by a loose connection.
3. No bits are faulty. However, due to outside transient conditions, such as noise, the value on any wire may flip.

In our model we read the pattern from memory and search for it in the text. However, our address register is faulty thus the pattern we get is a scrambled version of the “real” pattern. We seek, for every text location, the smallest number of inconsistent faulty bits, in the sense of error type 2 above, that would enable the pattern to match the text at that location (if such a matching exists). We call this problem *the faulty bits distance problem.* A naive check of each possible set of faulty bits yields an $O(nm^2)$ time algorithm. [2] provide an $O(nm^{\log_2 3})$ time algorithm for patterns with a bounded alphabet, and a randomized $O(nm^{\log_2 3} \log m)$ time algorithm for patterns with an unbounded alphabet.

The case of error type 1 above is called in [2] the *consistent bit flip renaming.* In this renaming scheme full consistency is assumed, i.e., that some of the bits, in all the addresses, may have been flipped. For example, suppose that $m = 64$. Then, six bits are used in the binary representation of the addresses. In a bit-flip renaming, some or all of these bits may be inverted in a consistent manner. For example, bits $2, 3$ and 5 may be always inverted, resulting in 000000 becoming

011010, and 010101 becoming 001111. Given two strings $x, y \in A^m$, we wish to know if there is a way to consistently flip some or all of the address bits in order to convert x into y, and if so, what are these bits? Note that there are $2^{\log m} = m$ different possible renamings in this scheme. Naively checking each of these possibilities would necessitate $O(m^2)$ work. [2] obtain an algorithm that works in time $O(m \log m)$.

Different type of address schemes may also be considered. In various parallel architectures, as well as some attempts to try to even the address wire lengths, various tree structures are used (e.g. the pyramid architecture). Consider a simplified address scheme where the m processors in a network architecture, or memory elements, are leaves of a full binary tree of height $\log m$. Thus, reaching the value in a certain address means following the path from the root down. At every level we take a left turn if the next address bit is 0 and a right turn if the address bit is 1. See Figure 1 for a schematic.

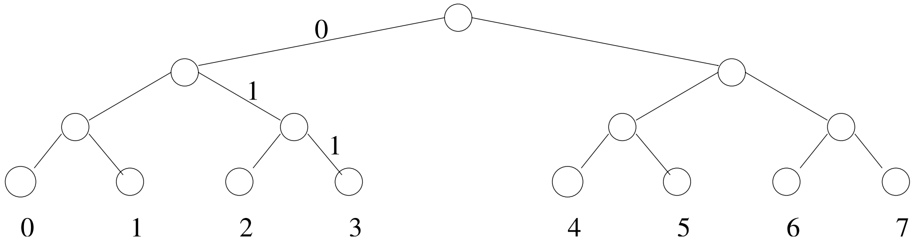


Fig. 1. An 8 element memory. The binary representation of 3 is 011. From the root, taking a left turn, then two right turns, brings us to location 3. If a node is consistently faulty, then we make a wrong turn every time we pass through that node.

Note that in this case the number of possible renamings is exponential, so a naive solution would be infeasible. Using a combination of methods borrowed from tree isomorphism algorithms and the Karp-Miller-Rosenberg string matching algorithm [22], [2] obtain an algorithm that solves the problem in $O(n \log m)$ steps, for a pattern of length m and text of length n.

Acknowledgement. The author was partially supported by NSF grant CCR-01-04494 and ISF grant 35/05. The author is indebted to the following friends who have all contributed to the formulation and development of this exciting new model: Yonatan Aumann, Gary Benson, Tzvika Hartman, Oren Kapah, Avivit Levy, Gadi Landau, Ohad Lipsky, Ely Porat, and Steven Skiena.

References

1. K. Abrahamson. Generalized string matching. *SIAM J. Comp.*, 16(6):1039–1051, 1987.
2. A. Amir, A. Aumann, and A. Levy. Pattern matching with address bit errors. Submitted for publication, 2006.

3. A. Amir, Y. Aumann, G. Benson, A. Levy, O. Lipsky, E. Porat, S. Skiena, and U. Vishne. Pattern matching with address errors: rearrangement distances. In *Proc. 17th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1221–1229, 2006.
4. A. Amir, R. Cole, R. Hariharan, M. Lewenstein, and E. Porat. Overlap matching. *Information and Computation*, 181(1):57–74, 2003.
5. A. Amir, T. Hartman, O. Kapah, and A. Levy. Interchange and weighted-interchange rearrangement distances in strings. submitted for publication, 2006.
6. A. Amir, M. Lewenstein, and E. Porat. Approximate swapped matching. *Information Processing Letters*, 83(1):33–39, 2002.
7. A. Amir, M. Lewenstein, and E. Porat. Faster algorithms for string matching with k mismatches. *J. Algorithms*, 2004.
8. V. Bafna and P.A. Pevzner. Sorting by transpositions. *SIAM J. on Discrete Mathematics*, 11:221–240, 1998.
9. M. A. Bender, D. Ge, S. He, H. Hu, R. Y. Pinter, S. Skiena, and F. Swidan. Improved bounds on sorting with length-weighted reversals. In *Proc. 15th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 912–921, 2004.
10. M. A. Bender, D. Ge, S. He, H. Hu, R. Y. Pinter, and F. Swidan. Sorting by length-weighted reversals: Dealing with signs and circularity. In *Proc. 15th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 3109 of *LNCS*, pages 32–46. Springer, 2004.
11. P. Berman and S. Hannenhalli. Fast sorting by reversal. In D.S. Hirschberg and E.W. Myers, editors, *Proc. 8th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 1075 of *LNCS*, pages 168–185. Springer, 1996.
12. A. Carpara. Sorting by reversals is difficult. In *Proc. 1st Annual Intl. Conf. on Research in Computational Biology (RECOMB)*, pages 75–83. ACM Press, 1997.
13. D. A. Christie. Sorting by block-interchanges. *Information Processing Letters*, 60:165–169, 1996.
14. R. Cole, L. Gottlieb, and M. Lewenstein. Dictionary matching and indexing with errors and don't cares. In *Proc. 36th annual ACM Symposium on the Theory of Computing (STOC)*, pages 91–100. ACM Press, 2004.
15. R. Cole and R. Hariharan. Approximate string matching: A faster simpler algorithm. In *Proc. 9th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 463–472, 1998.
16. P. Ferragina and R. Grossi. Fast incremental text editing. *Proc. 7th ACM-SIAM Symposium on Discrete Algorithms*, pages 531–540, 1995.
17. M.J. Fischer and M.S. Paterson. String matching and other products. *Complexity of Computation, R.M. Karp (editor), SIAM-AMS Proceedings*, 7:113–125, 1974.
18. Z. Galil and R. Giancarlo. Improved string matching with k mismatches. *SIGACT News*, 17(4):52–54, 1986.
19. M. Gu, M. Farach, and R. Beigel. An efficient algorithm for dynamic text indexing. *Proc. 5th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 697–704, 1994.
20. J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kauffmann, 3rd edition, 2002.
21. H. Karloff. Fast algorithms for approximately counting mismatches. *Information Processing Letters*, 48(2):53–60, 1993.

22. R. Karp, R. Miller, and A. Rosenberg. Rapid identification of repeated patterns in strings, arrays and trees. *Symposium on the Theory of Computing*, 4:125–136, 1972.
23. G. M. Landau and U. Vishkin. Efficient string matching with k mismatches. *Theoretical Computer Science*, 43:239–249, 1986.
24. V. I. Levenshtein. Binary codes capable of correcting, deletions, insertions and reversals. *Soviet Phys. Dokl.*, 10:707–710, 1966.
25. R. Lowrance and R. A. Wagner. An extension of the string-to-string correction problem. *J. of the ACM*, pages 177–183, 1975.
26. S. C. Sahinalp and U. Vishkin. Efficient approximate and dynamic matching of patterns using a labeling paradigm. *Proc. 37th FOCS*, pages 320–328, 1996.

SNP and Haplotype Analysis – Algorithms and Applications

Eran Halperin

International Computer Science Institute, Berkeley, CA, 94709
heran@icsi.berkeley.edu

Abstract. The recent release of the Haplotype Mapping project (Nature, Oct. 26, 2005 - see also, e.g., NY Times, Oct. 27), and the rapid reduction in genotyping costs open new directions and opportunities in the study of complex genetic disease such as cancer or Alzheimer's disease. The datasets collected for these studies are DNA sequences, with some noise and ambiguous information.

In this talk I will discuss some of the algorithmic issues of disambiguating these DNA sequences, and the current and potential impact of these algorithms on genetics and medicine. In particular, I will discuss some of the problems in the field, such as genotype phasing, tag SNP selection (e.g. feature selection), and population stratification issues (e.g. clustering).

The talk will be self contained, although some introductory material for the biological terminology and the HapMap project can be found at http://www.hapmap.org/whatishapmap.html.

M. Lewenstein and G. Valiente (Eds.): CPM 2006, LNCS 4009, p. 11, 2006.

Identifying Co-referential Names Across Large Corpora

Levon Lloyd, Andrew Mehler, and Steven Skiena

Department of Computer Science
State University of New York at Stony Brook
Stony Brook, NY 11794-4400
{lloyd, mehler, skiena}@cs.sunysb.edu

Abstract. A single logical entity can be referred to by several different names over a large text corpus. We present our algorithm for finding all such co-reference sets in a large corpus. Our algorithm involves three steps: morphological similarity detection, contextual similarity analysis, and clustering. Finally, we present experimental results on over large corpus of real news text to analyze the performance our techniques.

1 Introduction

A single logical entity can be referred to by several different names over a large text corpus. For example, *George Bush* is often referred to as *Bush*, *President Bush*, *George W. Bush*, or *"W"*, even among polite company. However, morphologically-similar names like *George H.W. Bush* can refer to different entities. Accurately identifying the members of the *co-reference set* for a given entity is an important problem in text mining and natural language processing.

Our interest in identifying such co-reference sets arises in the context of our system *Lydia* [1, 2, 3, 4], which seeks to build a relational model of people, places, and things through natural language processing of news sources. Indeed, we encourage the reader to visit our website (`http://www.textmap.com`) to study our analysis of recent news obtained from over 500 daily online news sources. In particular, we display the members of each of the 100,000 synsets we reconstruct daily (on a single commodity computer) from the roughly 150,000 entity-names we currently track.

Our algorithm for identifying co-referring name sets accurately and efficiently on a large scale involves optimizing our algorithm's three steps:

1. *Morphological Similarity* – The scale of our problem makes it infeasible to explicitly compare each pair of names for possible co-reference. First, we narrow our search space by identifying candidate pairs for analysis on a strictly syntactic basis via morphologically-sound hashing techniques.
2. *Contextual Similarity* – Next, we determine how similar a pair of names is based on the contexts in which they are used. The scale of our problem makes it infeasible to explicitly analyze all text references associated with each pair

M. Lewenstein and G. Valiente (Eds.): CPM 2006, LNCS 4009, pp. 12–23, 2006.

of candidate names. Instead, we propose methods using co-occurrence analysis to *other* entities to determine the probability that they are co-referent by context.

3. *Evidence Combination and Clustering* – Finally, we combine our measures of contextual and morphological similarity in order to cluster the names. The problem of clustering names is complicated by the vast difference in the number of references between popular and infrequently-used names. The strength of our contextual evidence is thus substantially weaker for unpopular names. We propose and evaluate methods for dealing with this problem.

Our problem is different from traditional cross-document co-reference analysis (see Section 2.1). In that problem, there is a set of documents that all mention the *same* name and the difficulty is clustering the documents into sets that are mentioning the same entity. In our problem, there is a set of documents that mention the many entities each possibly with multiple names and we want to cluster the names. This difference, combined with our need to manage the daily flow and scale of the news presented challenges that separate us in the following ways: (1) the use of entity co-occurrence lists as the sole feature for contextual analysis, (2) our high-speed dimension reduction techniques (based on k-means clustering and graph partitioning algorithms) to improve the quality of our contextual analysis and the efficiency of our algorithms, (3) our use of morphological similarity hashing techniques to avoid the need for pairwise-similarity testing of all name pairs, and (4) our use of *variable precision phonetic hashing* in order to tune the performance of our morphological similarity phase.

The rest of this paper is organized as follows. Section 2 surveys previous work on this and other problems. Section 3 discusses notions of morphological similarity, while Section 4 shows how we compute the probability that two names are co-referential from their respective co-occurrence lists. Section 5 discusses issues that arise in clustering. Experimental results are given in Section 6. We present our conclusions in Section 7.

2 Related Work

The problem of identifying co-reference sets has been widely studied in a variety of different contexts. In this section, we survey related work.

We now describe work on three related problems in the subsections below, namely, cross-document and in-document co-reference resolution in natural language processing and record linkage in databases.

2.1 Cross Document Co-reference Resolution

The complementary problem of cross-document co-reference has been examined fairly extensively.

Bagga and Baldwin [5] present an algorithm which extracts each sentence in each document that contains an ambiguous name and forms a summary of the document with respect to the entity. They then use the vector space model to compute the similarity of two such summaries. If the similarity of the two documents is above a threshold, then they consider the two documents to be referring the same person. They concluded that good results could be achieved by looking at the context surrounding the occurrences of the name and comparing documents using techniques from information retrieval.

Mann and Yarowsky [6] present a partially supervised algorithm for this problem. The algorithm takes as input either a small set of seed tuples for each of a small set of personal attributes from which it generates extraction patterns or a set of hand-crafted extractions for each of the personal attributes. Next, it uses these values along with other contextual clues as the feature vector for each document before using bottom-up centroid agglomerative clustering.

Gooi and Allan [7] study statistical techniques for cross-document co-reference resolution. Like Bagga and Baldwin, they use snippets of text around each mention of the ambiguous name. They compare *agglomerative clustering*, repeatedly merging the closest pair of clusters, with *incremental clustering*, either adding each point to an existing cluster or starting a new singleton cluster, and KL-divergence as a distance function with cosine similarity. They conclude that agglomerative clustering performs better than incremental clustering, however incremental clustering is much more time efficient. They also conclude that cosine similarity performs better using KL-divergence.

2.2 Within Document Co-reference Resolution

The natural language processing community has extensively studied the problem of within document co-reference resolution, finding chains of noun phrases that refer to the same things. For example, in a news article, *Dick Cheney* may later be referred to as *Vice President*, *he*, or *Mr. Cheney*.

Ng and Cardie [8] present a supervised machine learning-based algorithm for within document co-reference resolution. They use a decision tree classifier to classify each pair of noun phrases in a document as either co-referring or not and a clustering algorithm to resolve conflicting classifications. They experiment with different feature sets, clustering algorithms, and training set selection algorithms. They conclude that linking a proper noun phrase to its most probable previously occurring co-referring phrase is a better way of clustering, that a training set selection algorithm that is designed for this clustering algorithm is superior, and while adding features can be helpful, too many can degrade performance.

Bean and Riloff [9] present an unsupervised approach to co-reference resolution that uses contextual role knowledge to determine if two noun phrases co-refer. First they identify easy-to-resolve co-referring pairs and use them as training data. Information extraction patterns are then used to generate information about the role each noun phrase plays in the text. The information extracted from the training data is used to help resolve the other pairs in the corpus. They show that this phase increases recall substantially with just a slight decrease in precision.

2.3 Record Linkage

Our co-reference set identification problem is similar to the record linkage problem from data mining. The problem arises when there is no shared, error-free key field to join on across databases. Consider two tables containing information about people from two different databases. Even if both databases used the person's name and address as the primary key, conventions concerning abbreviations and word usage may differ, and typos and misspellings may appear in either field. The goal is to identify which records correspond to the same entities.

Hernandez and Stolfo [10] present two different techniques for large databases. The first approach sorts the data on some key and only considers two records for a merge if they are in a small neighborhood of each other. The second clusters the records in such a way that two records will be in the same cluster if they are potentially referring to the same entity. Finally, they propose taking the transitive closure of independent runs of the above algorithms, with independent key fields, as the final merge. They show that this multi-pass algorithm is superior to all the other algorithms that they consider.

Cohen and Richman [11] consider two problems: (1) taking in a pair of lists of names and determining which pairs of names in the different lists are the same and (2) taking in a single list of names and partitioning them into clusters that refer to the same entity. They propose adaptive learning-based matching and clustering methods to solve either of these problems. Their feature vector includes whether one string is a substring of the other and the edit distance between the two strings.

3 Morphological Similarity

With hundreds of thousands of names occurring in a large corpus, it is intractable to compare every pair as potentially co-referential. Further, most of these comparisons are clearly spurious, and thus would increase the possibility of false positives. We propose that most pairs of co-referential names result from the following set of morphological transformations:

- *Subsequence Similarity* – Taking a string subsequence of a name is one way of generating aliases of that name. For example, *Ford Motor Co.* is often referred to as *Ford* and *George W. Bush* is also called *George Bush.* To identify these pairs, we examine all 2^n possible string subsequences of each n-word name, hashing the name on each of its subsequences. Note that n, the number of words in a name, is bounded by about 10. Any subsequence matching another name implies potential morphological compatibility.
- *Pronunciation Similarity* – The Metaphone [12] algorithm returns a hash code of a word such that two words share the same hash code if they have similar English pronunciations. Here we say that two names are morphologically-compatible if they have the same metaphone hash code. Metaphone is

useful in identifying different spellings of foreign language names (e.g. *Victor Yanukovich* and *Viktor Yanukovych*) as possibly co-referential. In Section 3.1, we detail our methods for tuning the performance of this aspect of morphological similarity using variable precision phonetic hashing.

- *Stemming* – We use a Porter stemmer [13] to stem each word of each name and use the stem as a hash code for each name. A hash code collision means that two names have morphologically-compatible names. Stemming can be used to identify pairs like *New York Yankee* and *New York Yankees.*
- *Abbreviations* – If one name is an abbreviation of another, then we say that they are morphologically compatible. For example JFK and John F. Kennedy are both co-referential with John Fitzgerald Kennedy. To find all names that are abbreviations of an name, we check if any of the 2^n possible abbreviations of the name's n-words are also in our corpus.

We observe that there is a notion of degree of morphological similarity. For example, *George Bush* is more likely to be co-referential with *George W. Bush* than *U.S.* is with *Assistant U.S. Attorney Richard Convertino.* For each of our notions of morphological similarity we have a different measure of the degree of similarity. For example, for pronunciation similarity, we model the generation of aliases as a stochastic "typing" process where the probability of a mis-type is a constant. Then we compute the probability that one name was "typed-in" when the other was intended.

3.1 Variable Precision Phonetic Hashing

Several (e.g. [12, 14, 15]) phonetic hashing schemes have been developed to work well on a specific data set or for specific performance levels. No methods exist that allow the hashing scheme to be parameterized to give different precision/recall tradeoffs. In this section we investigate phonetic hashing schemes that have an adjustable parameter giving a range of operating points with different precision/recall tradeoffs.

Given a query string, we envision a sequence of transformations from the query string to an empty or null string, where each transformation is a new version of the string that has had some tokenization or weakening applied to it. We can model the space of transformations on the universe of strings as a graph. For example the name 'Wright', is shown in Figure 1, with a possible transformation sequence.

The weight of each change is determined by how drastic it is. So the distance from 'Writ' to 'Rit' should be relatively small when compared with the distance from 'Rt' to 'R'. This tokenization path gives us different versions of the query name to use in different tolerances of the hashing function. We also see that the path for the name 'Rite' eventually joins the path of 'Wright'. The name 'Reston' similarly joins the path, but lower down; suggesting that 'Rite' and 'Wright' are closer to each other then to 'Reston'.

A particular tokenizer in this scheme specifies a set of n-gram substitution rules, along with weights for the rules. The rules are applied in a lowest cost rule

Reston → Restn ⇝ Rst ↘
Wright → Writ → Rit → Rt → R →
Rite ↗

Fig. 1. Tokenization Path of the Name 'Wright'

first order. An example set of rules that could have generated Figure 1 is shown below. This table says the cheapest rule is substituting a 't' for 'ght'. The next cheapest is substituting an 'r' for 'wr' only if at the start of a query. Finally there are three deletion rules. The vowel deletion is considered less destructive, and is given a lower weight then the consonant deletion.

- ght → t;0.2
- _wr → r;0.3
- (a|e|i|o|u) → ;1
- (t|r) → ;5

To complete the definition of the hash function we must specify how to select the point on the tokenization path to operate at. Among the many candidates for these scoring methods, our experimentation showed that selecting the code that is a fixed distance from the null string works best.

Table 1 shows how we can vary the precision and recall of our hashing algorithm to get different tradeoffs. For a hand-created set of names extracted from our test set (see Section 6), we measured the precision and recall of our hashing algorithm at a range of its operating points. For comparison, we also show the precision and recall of three other phonetic hashing algorithms. It shows how we can use our algorithm to dial in the precision and recall of our notion of pronunciation similarity.

Table 1. Precision and Recall for our Variable Precision Phonetic Hashing and fixed precision hashing

Code Weight	Precision	Recall
0	0.002	1
120	0.150	0.909
121	0.139	0.818
141	0.157	0.727
146	0.293	0.636
167	0.360	0.545
172	0.442	0.454
187	0.662	0.363
229	1.000	0.090
Metaphone	0.715	0.732
Soundex	0.468	0.797
NYSIIS	0.814	0.672

4 Contextual Similarity

Our mental model of where an entity fits into the world depends largely upon how it relates to other entities.

We predict that the co-occurrences associated with two co-referential names (say *Martin Luther King* and *MLK*) would be far more similar than those of morphologically-similar but not co-referential pairs (say *Martin Luther King* and *Martin Luther*). Thus we use the vector of co-occurrence counts for each name as our feature space for contextual similarity.

We identified two primary technical issues in determining contextual similarity using this feature space: (1) dimension reduction and (2) functions for computing the similarity of two co-occurrence lists. Each of these will be described in the following subsections.

4.1 Dimension Reduction

In the experimental run of 88, 097 newspaper days of text we used throughout our experiments (details in Section 6), we encountered 174, 191 different names that occurred more than 5 times. This implies an extremely sparse, high-dimensional feature space – large because each additional entity name represents a new dimension, and sparse because a typical entity only interacts with a few hundred or so other entities even in a large text corpus.

Our experiments show that simple techniques which hunted for identical terms among the 100 or so most significant entries on each co-occurrence list failed, because the most significantly co-occurring terms for an name were highly unstable, particularly for low frequency names. Much more consistent were "themes" of co-occurring terms. In other words, while the most significant associations of *George Bush* and *"W"* might have relatively few names in common, both will be strongly associated with "Republican" and "Texas" themes.

Dimension-reduction techniques provide a way to capture such themes, and can improve both recognition accuracy and the computational efficiency of co-reference set construction. We examined two different dimension-reduction techniques based on creating crude clusters of names, then project our co-occurrence lists onto this smaller space.

- *K-means clustering* – This widely-used clustering method is simple and performs well in practice. Beginning with k randomly selected names as initial cluster centroids, we assign each name to its closest centroid (using cosine similarity of co-occurrence lists) and recompute centroids. After repeating for a given number of iterations (5, in our case) we assign each name to its closest centroid and take this as our final clustering.
- *Graph partitioning* – The problem of graph partitioning seeks to partition the vertices of a graph into a small number of large components by cutting a small number of edges. Such components in a graph of co-occurrences should correspond to "themes", subsets of terms which more strongly associate with themselves than the world at large. Thus we propose graph partitioning as a

potential dimension reduction technique for such relational data – the names in each component will collapse to a single dimension.

Although graph partitioning is NP-complete [16], reasonable heuristics exist. In particular, we used METIS[17], a well-known program for efficiently partitioning large weighted graphs into k high-weight subgraphs, with k being a user-specified parameter. Our graph contains a node for every name and an edge between every pair of nodes (x, y) if they co-occur with each other at least once. The weight assigned to each edge is the cosine similarity between the co-occurrence lists of x and y.

4.2 Measuring Contextual Similarity

Given two names, with their co-occurrence lists projected onto our reduced dimensional space, we now want a measure of how similar they are. We consider two different approaches: (1) they can be viewed as probability distributions and be compared by *KL-divergence* or (2) they can be viewed as vectors and compared by the cosine of the angle between the vectors. We detail each of these potential measures here.

KL-Divergence. The KL-Divergence is an information theoretic measure of the dissimilarity between two probability distributions. Given two distributions, the KL-Divergence of them is defined by

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

To use this measure, we turn each co-occurrence list into a probability distribution for each name i,

$$\hat{p}_i(j) = \frac{\text{number of co-occurences between i and j}}{\text{total number of co-occurrences for i}}$$

As a discounting method for probability-0 pairs, we do linear smoothing of all probabilities with the background distribution setting

$$p_i(j) = \alpha \hat{p}_i(j) + (1 - \alpha) bg(j)$$

where

$$bg(j) = \frac{\text{total occurrences of names in cluster j}}{\text{total number of entity occurrences in corpus}}$$

Cosine Similarity. A standard way of comparing contexts views the two contexts as vectors in a high dimensional space and computes the cosine of the angle between them. [5] proposed this technique for the similar problem of personal name disambiguation. We use the *term frequency-inverse document frequency* of each vector position, we weight each term in the vector by the inverse of the number of occurrences it has in the corpus. Letting N be the number of sentences in the corpus, our score is

$$d(x,y) = \sum_{i=0}^{k} jp_x^*(i) \cdot jp_y^*(i)$$

where $jp_x(i)$ the number of co-occurrences between i and x, weighted by log(N/ number of occurrences of i), and

$$jp_x^*(i) = \frac{jp_x(i)}{\|jp_x\|}$$

5 Issues in Clustering

Now that we know which pairs of names are morphologically-similar and their degrees of morphological and contextual similarity, we need: (1) a way of combining morphological and contextual similarities into a single probability that two names are co-referential and (2) a method to cluster names into co-reference sets. We discuss each problem below.

5.1 Combining Notions of Similarity

For each pair of morphologically-related names, we have measures of their morphological and contextual similarities. We need a way to combine them into a meaningful probability that the two names are co-referential.

For each measure of contextual similarity and for edit distance, we computed the precision curve on our experimental corpus (see Section 6). Since the precision at a measure of similarity is the probability that a pair from the test set with this amount of similarity were co-referential, we use these curves to turn each of our notions of similarity into a probability. Assuming that these two probabilities are independent, we now can compute the probability that these two names are co-referential by multiplying the probabilities given by their morphological and contextual similarities.

5.2 Clustering Algorithms

Once we have probabilities associated with each pair of morphologically related names, we need to group them into co-reference sets. Because our system must be able to handle large numbers of names, we must be careful of what kind of clustering algorithm we choose. We experimented with two algorithms:

- *Single link* – Here we merge the clusters that two names are in if the probability that they are co-referential is above a threshold.
- *Average link* – Our algorithm merges two clusters if the weighted average probability between names in each of the clusters is above a threshold.

6 Experimental Results

In order to optimize various parameters, decide which methods work best, and verify our techniques, we ran a set of experiments against the same test set that was used to produce the precision curves described in section 5.1. Each of these experiments is described below.

All of the experiments in this paper where conducted on a test set of $88,097$ newspaper-days worth of text, partitioned among 604 distinct publications. These were taken from spidering that was performed between April 11, 2005 and November 5, 2005. We used a hand-crafted set of roughly 320 co-reference sets from the entities in this corpus.

In Section 6.1, precision is given by $\frac{tp}{tp+fp}$, recall by $\frac{tp}{tp+fn}$, and f-score by $\frac{1}{\alpha\frac{1}{P}+(1-\alpha)\frac{1}{R}}$ where tp $=$ true positives, fp $=$ false positives, and fn $=$ false negatives.

In Section 6.2 these measures are given by the B-cubed algorithm introduced in [5]. For each name

$$\text{Precision} = \frac{\|\text{intersection of propsed set and true set}\|}{\|\text{proposed set}\|}$$

$$\text{Recall} = \frac{\|\text{intersection of proposed set and true set}\|}{\|\text{true set}\|}$$

and overall precision and recall are the averages of these values.

6.1 Optimizing Contextual Similarity Measure

Optimizing our contextual similarity phase involves the proper choice of (1) dimension reduction algorithm, (2) number of dimensions, and (3) contextual similarity measure. For both of the dimension reduction algorithms(k-means, METIS) and both of the distance measures(KL-Divergence, Cosine similarity), we recorded the peak F-score as a function of number of dimensions from 10 to 290.

Figures 2 shows this plot. It shows that while the peak performance of all four combinations is to be comparable, KL-Divergence with METIS dimension

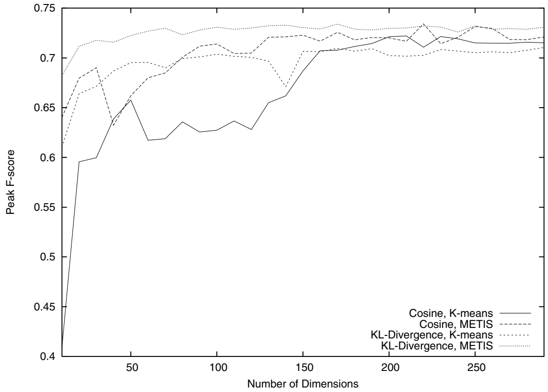
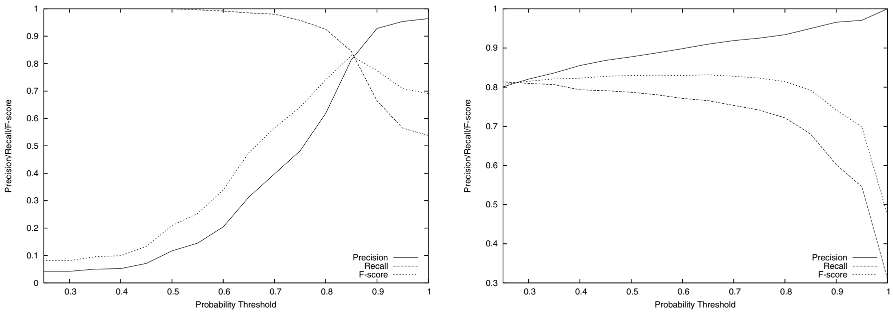


Fig. 2. Number of Clusters vs. Peak F-score for our dimension reduction algorithms and distance measures



(a) Single-link clustering

(b) Average-link clustering

Fig. 3. Threshold vs. Precision, Recall, and F-score for our clustering algorithms

reduction is to be the most robust to changes in k. For the rest of the analysis in this paper, we used KL-divergence, METIS dimension reduction, and 150 dimensions.

6.2 Clustering Methods

The first clustering algorithm that we tried was simple single link clustering. Figure 3(a) shows that it has decent peak performance, but is not very robust to the setting of the threshold. Further, manual evaluation of the clusters that are produced shows that it tends to create very long clusters, putting many things into the same cluster that should not even be considered. For example, the sequence *George Bush* → *Bush* → *Bush-Cheney* → *Cheney* → *Dick Cheney* leads to *George Bush* and *Dick Cheney* being called co-referential.

The next clustering algorithm that we tried was weighted-average link. Figure 3(b) shows that this has slightly better peak performance than single-link clustering, but is much more robust in the setting of the threshold.

7 Conclusion

In this paper we present an algorithm to find sets of co-referential names. We introduce the idea of morphological similarity, the notion that two names are potentially co-referential based on the text that comprises the name. Then we discuss the issues surrounding computing the contextual similarity of two names and give two different measures. Clustering names given their morphological and contextual similarities was discussed and we presented experimental results for our system.

References

1. Lloyd, L., Kechagias, D., Skiena, S.: Lydia: A system for large-scale news analysis. In: String Processing and Information Retrieval (SPIRE 2005). Volume Lecture Notes in Computer Science, 3772. (2005) 161–166

2. Lloyd, L., Kaulgud, P., Skiena, S.: Newspapers vs. blogs: Who gets the scoop? In: Computational Approaches to Analyzing Weblogs (AAAI-CAAW 2006). Volume AAAI Press, Technical Report SS-06-03. (2006) 117–124
3. Kil, J., Lloyd, L., Skiena, S.: Question answering with lydia. 14th Text REtrieval Conference (TREC 2005) (2005)
4. Mehler, A., Bao, Y., Li, X., Wang, Y., Skiena, S.: Spatial analysis of news sources. submitted for publication (2006)
5. Bagga, A., Baldwin, B.: Entity-based cross-document coreferencing using the vector space model. In Boitet, C., Whitelock, P., eds.: Proceedings of the Thirty-Sixth Annual Meeting of the Association for Computational Linguistics and Seventeenth International Conference on Computational Linguistics, San Francisco, California, Morgan Kaufmann Publishers (1998) 79–85
6. Mann, G., D.Yarowsky: Unsupervised personal name disambiguation. In: CoNLL, Edmonton, Alberta, Canada (2003) 33–40
7. Gooi, C., Allan, J.: Cross-document coreference on a large scale corpus. In: Human Language Technology Conf. North American Chapter Association for Computational Linguistics, Boston, Massachusetts, USA (2004) 9–16
8. Ng, V., Cardie, C.: Improving machine learning approaches to coreference resolution. In: 40th Annual Meeting of the Association for Computational Linguistics, Philadelphia, Pennsylvania, USA (2002) 104–111
9. Bean, D., Riloff, E.: Unsupervised learning of contextual role knowledge for coreference resolution. In: Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics, Boston, Massachusetts, USA (2004) 297–304
10. Hernandez, M., Stolfo, S.: The merge/purge problem for large databases. In: Proceedings of the 1995 ACM SIGMOD International Conference on the Management of Data, San Jose, California, USA (1995) 127–138
11. Cohen, W., Richman, J.: Learning to match and cluster large high-dimensional data sets for data integration. In: Eighth ACM SIGKDD Conf. Knowledge Discovery and Data Mining. (2002) 475–480
12. Philips, L.: Hanging on the Metaphone. Computer Language **7**(12) (1990) 39–43
13. Porter, M.: An algorithm for suffix stripping. http://www.tartarus.org/∼martin/PorterStemmer/def.txt (1980)
14. Taft, R.: Name search techniques. New York State Identification and Intelligence Systems, Special Report No. 1, Albany, New York. (1970)
15. Borgman, C., Siegfried, S.: Getty's synoname and its cousins: A survey of applications of personal name-matching algorithms. JASIS **43**(7) (1992) 459–476
16. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the theory of NP-completeness. W. H. Freeman, San Francisco (1979)
17. Karypis, G., Kumar, V.: METIS: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. http://www-users.cs.umn.edu/ karypis/metis (2003)

Adaptive Searching in Succinctly Encoded Binary Relations and Tree-Structured Documents

Jérémy Barbay[1], Alexander Golynski[1], J. Ian Munro[1], and S. Srinivasa Rao[2]

[1] David R. Cheriton School of Computer Science
University of Waterloo, Canada
[2] Computational Logic and Algorithms group
IT University of Copenhagen, Denmark

Abstract. The most heavily used methods to answer conjunctive queries on binary relations (such as the one associating keywords with web pages) are based on inverted lists stored in sorted arrays and use variants of binary search. We show that a succinct representation of the binary relation permits much better results, while using space within a lower order term of the optimal. We apply our results not only to conjunctive queries on binary relations, but also to queries on semi-structured documents such as XML documents or file-system indexes, using a variant of an adaptive algorithm used to solve conjunctive queries on binary relations.

Keywords: conjunctive queries, intersection problem, succinct data structures, labeled trees, multi-labeled trees.

1 Introduction

Consider the task of a search engine answering conjunctive queries: given a set of keywords, it must return a list of references to the objects relevant to all those keywords. These objects can be web-pages as in the case of a web search engine like Google, or documents as in a search engine on a file system, or any kind of data searched by keywords in general. Rather than roam the set of all objects (which is usually huge — think about the set of web-pages indexed by Google), a good search engine uses a *precomputed index*, which represents the binary relation between the space of objects $\{1, \ldots, n\} = [n]$ and the space of admissible keywords $\{1, \ldots, \sigma\} = [\sigma]$, so that it can be easily searched.

Usually, such an index is coded as a set of sorted arrays, so that the answer to conjunctive queries is the intersection of those arrays. This intersection can then be computed in time linear in the sum of the sizes of the array, but several adaptive algorithms have been studied for the easier case where a small number of comparisons permits to check the result, with much better results than linear [2, 3, 5, 6]. These intersection algorithms are all based on variants of the binary search algorithm: as the cost of a search is logarithmic in the size of the

M. Lewenstein and G. Valiente (Eds.): CPM 2006, LNCS 4009, pp. 24–35, 2006.

array, this impacts on their complexity, in particular on "easy" instances where the intersection is empty or where only a few comparisons are sufficient to check the result of the intersection.

Our results are threefold:

- First, observing that the use of inverted lists in sorted arrays is far from being a mandatory step to compute the intersection, we consider instead succinct data structures to encode the binary relation, which also permits much faster searches. We give two representations (Theorem 1) for binary relations associating n objects with σ labels in t pairs from $[n]\times[\sigma]$. Each of these representations uses $t\big(\lg\sigma + o(\lg\sigma)\big)$ bits, and supports queries in time $\mathcal{O}(\lg\lg\sigma)$ or better (depending on the operator and on the encoding), thus generalizing the results from Golynski *et al.* [9] on strings on large alphabets. These results can be directly applied to the intersection problem (Theorem 3), to improve the time complexity of the algorithm from Barbay and Kenyon [3], and thus to reduce the time required to answer a conjunctive query.
- Second, observing that a labeled tree is simply a tree in which each node is associated with a label through a binary relation, we give a representation for labeled trees (Theorem 2). This uses $n\big(\lg\sigma + o(\lg\sigma)\big)$ bits and supports both structure-based navigation operators in constant time and label-based search operators in time $\mathcal{O}(\lg\lg\sigma)$ or better, improving on the space used by the solutions from both Geary *et al.* [8] and Ferragina *et al.* [7] on labeled trees. These results can be immediately generalized to multi-labeled trees (such as XML documents or file-system indexes) where nodes are associated with zero or more labels in t pairs (rather than only n pairs in labeled trees), giving a representation (Corollary 1) which uses $t\big(\lg\sigma + o(\lg\sigma)\big)$ bits and supports the same operators in the same time.
- Third, observing the similarity between conjunctive queries and *unordered path-subset* queries on labeled and multi-labeled trees, we prove tight upper (Theorem 4) and lower (Theorem 5) bounds on the complexity of any randomized algorithm solving these queries, hence extending the results of Barbay and Kenyon on the intersection problem [3] to unordered path-subset queries on multi-labeled trees.

All our results concerning the running time of operators and algorithms are expressed in the RAM model, where words of size $\Theta(\lg(\max\{n,\sigma\}))$ can be accessed and processed in constant time.

The paper is organized as follows. In Section 2, we present our succinct data structures for the three objects considered: binary relations in Section 2.1, labeled trees in Section 2.2, and multi-labeled trees in Section 2.3. The encoding of binary relations and the encoding of labeled trees are combined to encode multi-labeled trees. We describe in Section 3 the algorithms that search the objects efficiently using those data structures: the adaptive algorithm for the intersection using our encoding of binary relations in Section 3.1, and our new adaptive algorithm for searching multi-labeled trees in Section 3.2. We conclude in Section 4 with some perspectives on future work.

2 Succinct Indexes

2.1 Binary Relations

Consider a binary relation R between an ordered set of n objects and an ordered set of σ labels. Let t denote the cardinality of R, i.e. the number of pairs (object, label) that are in R. In the context in which objects are references to web-pages, and labels are keywords associated with the web-pages, such relations are used to answer conjunctive queries, i.e. for a given set of keywords, to return all pages that are associated with all the keywords in the set. Typically, such a relation is encoded as a collection of *postings lists*, in which each list associates a sorted list of web pages (objects) to a keyword (label), which can be intersected [2, 3, 5, 6] to answer conjunctive queries.

Let α be a label from $[\sigma]$, x be an object from $[n]$, and r be an integer. We propose a succinct encoding of the relation R that takes asymptotically minimal space and supports the following operators:

- `label_rank`(α, x), the number of objects labeled α preceding x;
- `label_select`(α, r), the r-th object labeled α, if any, or ∞ otherwise;
- `label_nb`(α), the number of objects labeled α;
- `object_rank`(x, α), the number of labels associated with object x preceding label α;
- `object_select`(x, r), the r-th label associated with object x, if any, or ∞ otherwise;
- `object_nb`(x), the number of labels associated with object x;
- `table_access`(x, α), checks whether object x is associated with label α.

The naive encoding of such lists as sorted arrays uses $t \lg n + \sigma \lg t$ bits of space and supports `label_select`(α, r) in constant time, but `label_rank`(α, x) requires time logarithmic in the number of objects associated with label α. It is not clear how to support `object_rank`(x, α) and `object_select`(x, r) with such an encoding. Each posting list can also be represented by a binary string of length n, and encoded using Clark and Munro's [4] encoding to support the operators `label_rank` and `label_select` in constant time. However, this representation uses a total of $\sigma n + o(\sigma n)$ bits, which is too much in practice, especially when the number of pairs t is much smaller than σn.

The operators `label_rank` and `label_select` are extensions of the operators `string_rank` and `string_select` defined by Golynski *et al.* [9], who only considered the case of strings, or in other words, the case where each object (i.e. position in a string) is associated with exactly one label (i.e. a character from an alphabet of size σ, that occurs at the given position in the string). We support the `label_rank` and `label_select` operators in the same time as theirs. The operators `object_rank`, `object_select` are extensions of `string_access`: `string_access`(x) gives the label associated with x (i.e., the character at position x), the operators `object_rank` and `object_select` are used to navigate in the set of labels that are associated with a given object. The techniques from Golynski *et al.* are not directly applicable to the case of binary relations, however we use similar ideas and obtain an efficient implementation of the new operators `object_rank`, `object_select`, `label_nb`, `object_nb` and `table_access`. In what follows, we use two encodings described by Golynski *et al.*: `select` encoding and `access` encoding, and extend them to binary relations.

Theorem 1. *Consider a binary relation on* $[n] \times [\sigma]$ *of cardinality* t*. Assume that each object is associated with at least one label and each label is associated with at last one object. Then there are two encodings (named* `label` *encoding and* `object` *encoding), each using* $t\big(\lg\sigma + o(\lg\sigma)\big)$ *bits, that support the defined operators with the following run-times:*

	`label`	`object`
`label_rank`(α, x)	$\mathcal{O}(\lg\lg\sigma)$	$\mathcal{O}(\lg\lg\sigma\lg\lg\lg\sigma)$
`label_select`(α, r)	$\mathcal{O}(1)$	$\mathcal{O}(\lg\lg\sigma)$
`label_nb`(α)	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`object_rank`(x, α)	$\mathcal{O}\big((\lg\lg\sigma)^2\big)$	$\mathcal{O}(\lg\lg\sigma)$
`object_select`(x, r)	$\mathcal{O}(\lg\lg\sigma)$	$\mathcal{O}(1)$
`object_nb`(x)	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`table_access`(α, x)	$\mathcal{O}(\lg\lg\sigma)$	$\mathcal{O}(\lg\lg\sigma)$

where $x \in [n]$*,* $\alpha \in [\sigma]$*, and* r *is a positive integer.*

Proof (sketch). Without loss of generality, we assume that $\sigma \leq n$: the construction is similar in the symmetric case. We reduce the problem of encoding a binary matrix of size $\sigma \times n$ to the encoding of n/σ matrices of size $\sigma \times \sigma$ each, using the same technique as Golynski *et al* [9]: we call this step a *domain reduction.* Let t_M denote the number of ones in one of the smaller matrix M, and let the operators `row_rank`, `row_select`, `column_rank` and `column_select` have the same functionalities as the operators `label_rank`, `label_select`, `object_rank` and `object_select` respectively, but restricted to the smaller matrices, e.g. `row_rank`(i, j) is defined only for $j \leq \sigma$. This reduction allows the implementation of the operators `label_rank`, `label_select`, `object_rank`, `object_select` using the operators `row_rank`, `row_select`, `column_rank`, `column_select` with an acceptable space and time overhead.

We represent a boolean matrix M of size $\sigma \times \sigma$ by two strings: `COLUMNS`, on alphabet $[\sigma]$ and of length t_M, such that the k-th symbol of `COLUMNS` corresponds to the column index of the k-th pair in the row-major order[1] traversal of M; and `ROWS`, a binary string of length $t_M + \sigma$, such that the number of zeros between the i-th and the $i+1$-st one indicates how many ones are in the i-th row of M. We say that `COLUMNS` is divided into σ *parts* by `ROWS`. See the following example:

$$M = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix} \qquad \begin{array}{ll} \texttt{COLUMNS} & = 2, \quad 1, 2, 3, \quad 1, 4, \quad 2 \\ \texttt{ROWS} & = 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1 \end{array}$$

[1] *Row-major* order lists the elements from the first row, then from the second row, and so on.

We encode COLUMNS using one of the two encodings from Golynski *et al* [9] depending on the preferred time tradeoffs between different operators as mentioned in the statement of the theorem. These encodings use $t_M(\lg\sigma + o(\lg\sigma))$ bits of space with the following time tradeoffs:

	select encoding	access encoding
string_access	$\mathcal{O}(\lg\lg\sigma)$	$\mathcal{O}(1)$
string_select	$\mathcal{O}(1)$	$\mathcal{O}(\lg\lg\sigma)$
string_rank	$\mathcal{O}(\lg\lg\sigma)$	$\mathcal{O}(\lg\lg\sigma\lg\lg\lg\sigma)$

The vector ROWS can be encoded using any succinct fully indexable dictionary that supports in constant time the operators bin_rank and bin_select, the rank and select operators on binary strings introduced by Jacobson [10] and improved by Clark and Munro [4]. The operators column_select(i, j) and column_rank(i, j) are based on searching for occurrences of symbol j in the string COLUMNS, which is done through the string_rank and string_select operators on COLUMNS and bin_rank and bin_select operators on ROWS. The operator row_select(i, j) corresponds to a call to the string_select operator on the i-th part of COLUMNS.

A naive implementation of the operator row_rank(i, j) using a binary search on the i-th part of COLUMNS takes $\mathcal{O}(\lg x \cdot$ complexity of string_access) time, where $x \leq \sigma$ is the length of the i-th part of COLUMNS, which is not good enough. We use a sparsification idea similar to the one introduced by Golynski *et al* [9], fixing the parameter $z = \lg\sigma$ and encoding every z-th character of the i-th part of COLUMNS using a y-fast trie (as defined by Willard [13]). This structure supports the rank operator in the "sparsified" string Y in time $\mathcal{O}(\lg\lg\sigma)$ using $\mathcal{O}(x/z\sigma) = \mathcal{O}(x)$ bits (which is $\mathcal{O}(t)$ for all values of i together). Note that row_rank$(i, j) \in [z\,\mathtt{rank}_Y(j), z\,(\mathtt{rank}_Y(j) + 1)]$, where $\mathtt{rank}_Y$ is the *set rank*, which denotes how many elements in Y are smaller than j. The result of row_rank(i, j) can be computed using a binary search in an interval of size $\lg\sigma$ in time $\mathcal{O}(\lg\lg\sigma \cdot$ complexity of string_access).

The operator label_nb(α) can be done in constant time using ROWS. The operator object_nb(x) can also be done in constant time by maintaining an additional bit vector similar to ROWS that counts the numbers of occurrences for each column. The operator table_access(i, j) can be computed either as the difference between row_rank$(i, j + 1)$ and row_rank(i, j), or equivalently as the difference between column_rank$(i + 1, j)$ and column_rank(i, j).

The encoding of COLUMNS uses $t(\lg\sigma + o(\lg\sigma))$ bits (summed over all M). The encodings of y-fast tries and ROWS vectors use $\mathcal{O}(t + n)$ bits in total, hence the total space of $t\big(\lg\sigma + o(\lg\sigma)\big)$ bits for each encoding. □

Note, that the operators described above are "symmetrical" with respect to interchanging roles of objects and labels, so that we can assume that $n \geq \sigma$. The space used by the above data structure is almost optimal (equal to the information-theoretical minimum plus a lower order term) under the assumption that the average number of ones per column is small, namely if $t/n = \sigma^{o(1)}$. In this case the lower bound suggested by information theory, equal to $\lg\binom{n\sigma}{t}$, is roughly $t\big(\lg(n\sigma) - \lg t + \mathcal{O}(1)\big) = t\big(\lg\sigma - o(\lg\sigma)\big)$, which is close to the space used by our encodings, $t(\lg\sigma + o(\lg\sigma))$.

2.2 Labeled Trees

An *ordinal tree* is a rooted tree in which the children of a node are ordered and specified by their rank. Geary *et al.* [8] proposed an encoding for ordinal trees which supports in constant time the following operators, called *navigation operators*:

- `tree_ancestor`(x, i), the i-th ancestor of node x for $i \geq 0$;
- `tree_rank`$_{pre/post}(x)$, the position of node x in the *pre* or *post* order traversal of the tree;
- `tree_select`$_{pre/post}(r)$, the r-th node in the *pre* or *post* order traversal of the tree;
- `tree_child`(x, i), the i-th child of node x for $i \geq 1$;
- `tree_child_rank`(x), the number of left siblings of node x;
- `tree_depth`(x), the depth of x (number of edges in the path from root to x);
- `tree_nbdesc`(x), the number of descendants of x;
- `tree_deg`(x), the degree of x, i.e. its number of children.

Consider a set of σ labels, and an ordinal tree of n nodes such that each node is assigned a label: this is a *labeled tree* [7, 8]. Let α be a label from $[\sigma]$ and x be a node from $[n]$. We define the following operators on labeled trees, for the pre-order traversal of the tree:

- `labeltree_desc`(α, x), the first descendant of x which is labeled α, or ∞ if there is none;
- `labeltree_nbdesc`(α, x), the number of descendants of x that are labeled α;
- `labeltree_anc`(α, x), the ancestor of x which is labeled α and closest to the root, or ∞ if there is none;

In a manner similar to Ferragina *et al.* [7], we encode the structure of the tree separately from the labels, but we encode it as the trace of the pre-order traversal of the tree, and we encode the structure of the tree using Geary *et al.*'s [8] encoding for unlabeled trees.

Theorem 2. *Consider a labeled tree of n nodes and σ labels. There is an encoding using $n\big(\lg\sigma + o(\lg\sigma)\big)$ bits that supports in constant time the navigation operators on the structure of the tree and in time $\mathcal{O}(\lg\lg\sigma)$ the operators* `labeltree_anc`, `labeltree_desc` *and* `labeltree_nbdesc` *along with the operators* `string_rank`, `string_select` *and* `string_access` *on the pre-order traversal of the labels of the tree.*

Proof (sketch). Represent the structure of the tree as an ordinal tree encoded using the encoding for unlabeled ordinal trees defined by Geary *et al.* [8]: this takes $2n + o(n)$ bits, and supports the navigation operators on the tree structure in constant time.

The labels are extended by one bit (i.e. to an alphabet of size 2σ) such that any node x originally labeled α is now labeled:

- α_* if x has no ancestor labeled α (but eventually some descendants);
- α_*^+ if x has at least one ancestor labeled α.

The sequence of extended labels is encoded in pre-order, using the representation of Golynski *et al.* [9] which uses $n(\lg(2\sigma)+o(\lg(2\sigma))) = n(\lg\sigma+o(\lg\sigma))$ bits and supports the operators `string_access`, `string_select` and `string_rank` on the pre-order traversal of the labels of the tree in the times claimed.

The operator `labeltree_anc`(α, x) is supported by checking for the last node y labeled α_* in pre-order before x, which takes time $\mathcal{O}(\lg\lg\sigma)$, and checking that y is an ancestor of x, which takes constant time. The symmetric operator `labeltree_desc`(α, x) is supported by checking for the first node y labeled α_* or α_*^+ in pre-order after x, which takes time $\mathcal{O}(\lg\lg\sigma)$, and checking that y is a descendant of x, which takes constant time. The operator `labeltree_nbdesc`(α, x) is easily supported via a combination of calls to the navigation operators, and two calls to the operator `string_rank`. Overall, the encoding uses $2n + o(n) + n(\lg\sigma + o(\lg\sigma)) = n\big(\lg\sigma + o(\lg\sigma)\big)$ bits. □

The information-theoretic lower bound for storing a labeled tree on n nodes with σ labels is asymptotically $n\big(\lg\sigma - o(\lg\sigma)\big)$. Hence our encoding, which uses $n\big(\lg\sigma+o(\lg\sigma)\big)$ bits, differ from this bound by a lower order term in σ. Note that other encodings with similar results can be obtained using the other encodings proposed by Golynski *et al.*; we developed here only the most appropriate for our specific application.

2.3 Multi-labeled Trees

XML documents and file systems can be seen as tree-structured documents, but the labeled tree model described in the previous section is too restrictive to represent them, as several labels are associated with each leaf in XML documents, and several labels are associated with each internal node (folder) or leaf (file) in a file system. We consider an extension of labeled trees where any number of labels can be associated with each node.

Definition 1. *A* **multi-labeled tree** *is an ordinal tree on n nodes together with a set of σ labels, and a set of t pairs from $[n] \times [\sigma]$. The operators are the same as those on labeled trees: structure-based navigation operators (as defined by Geary* et al. *[8]) and label-based operators (as defined in Theorem 2).*

The results on binary relations from Section 2.1 combine very easily with the results on labeled trees from Section 2.2 to give an encoding supporting efficiently the operators on multi-labeled trees:

Corollary 1. *Consider a multi-labeled tree on n nodes and σ labels, associated in t pairs. There is an encoding using $t\big(\lg\sigma + o(\lg\sigma)\big)$ bits and supporting the same operators as the encoding of Theorem 2 and in the same time.*

Proof. The operators supported on labeled trees are extended to multi-labeled trees by replacing each operator defined on strings [9] by its equivalent on binary relations as defined in Theorem 1, in the first encoding (named `label`), which supports all operators in time $\mathcal{O}(\lg\lg\sigma)$ or better. □

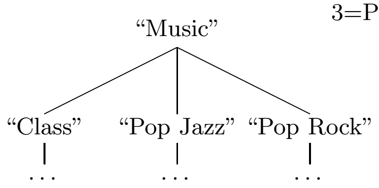


Fig. 1. A simplistic example of File System

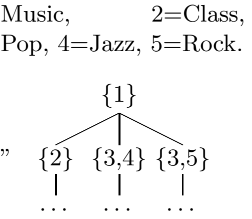


Fig. 2. The corresponding Multi-Labeled Tree

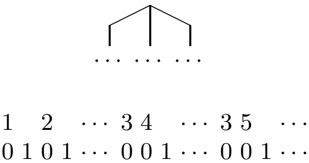


Fig. 3. The corresponding succinct encoding

Figure 1 represents a simplistic view of a personal file system organizing music files. Figure 2 shows its representation as a multi-labeled tree, where the text associated with each node is replaced by numbers from the range $[1, \sigma]$. Figure 3 shows the succinct encoding of this multi-labeled tree: the structure of the ordinal tree, the string representing the labels in pre-order, and a binary string where ones separate sequences of zeroes encoding the number of labels associated to a node. As in Section 2.1, the space used by our structure is optimal under the assumption that $t/n = \sigma^{o(1)}$.

3 Applications

3.1 Efficient Posting Lists

Several algorithms have been proposed for computing the answer to conjunctive queries on a binary relation, through the intersection of inverted lists in sorted arrays. The intersection of sorted arrays has been studied from several points of view, all of which are based on various search methods in sorted arrays: Several people have studied the intersection of a pair of sorted arrays, Baeza-Yates [1] being the most recent. Other efforts have been considering the intersection of a larger number of sorted arrays [2, 3, 5, 6], measuring the performance of the algorithms relative to the complexity of the description of a *certificate* of the intersection, such as the set of comparisons performed by a non-deterministic algorithm to check the result of the instance. We refer the reader to Figure 4 for a simple example, and to the cited papers for more details.

These search methods are limited to a complexity logarithmic in the size of the array. But the use of inverted lists in sorted arrays is far from being a mandatory step to computing the intersection. Our implementation for binary relations described in Section 2.1 permits us to search faster in the list of references

Music	→ A_1	1	8	10	12	15	17	19	→	1							8		10		12			15		17		19		
Jazz	→ A_4	2	4	6	9	11	13	20	→		2		4		6			9		11		13							20	
Rock	→ A_5	3	5	7	14	16	18	21	→			3		5		7							14		16		18			21

Fig. 4. An example of how a conjunctive query corresponds to the intersection of sets. A non-deterministic algorithm can check that the intersection is empty in $\delta = 4$ comparisons ($1 < 2, 7 < 8, 13 < 14, 19 < 20$). Barbay and Kenyon's algorithm performs $8 < \delta k$ searches ($1 < 2 < 3 < 8 < 9 < 14 < 15 < 20 < 21$). Most intersection algorithms use variants of binary search in the sorted array. We propose to use the rank operator on a succinct encoding of the binary relation.

associated with an object, and hence improves the performance of intersection algorithms.

Theorem 3. *Consider a set of objects $[n]$ and a set of labels $[\sigma]$, associated in t pairs from $[n] \times [\sigma]$, and a conjunctive query Q composed of k labels from $[\sigma]$. There is a deterministic algorithm solving Q in time $\mathcal{O}(\delta k \lg \lg \sigma)$, where δ is the minimum number of operations performed by any non-deterministic algorithm to check the result of Q.*

Proof (sketch). Barbay and Kenyon [3, Theorem 3.3] proposed a deterministic algorithm for the conjunctive query that uses $\mathcal{O}(\delta k)$ doubling searches. We replace the doubling search by a combination of `label_rank`, `label_select` and `label_access` operators, and the result follows. Suppose that x is initialized as the first object of $[n]$, and α as the first label of the query. If we introduce the bogus object ∞, which matches all labels and is a successor to all objects, the algorithm now goes as follows:

1. **If** $x = \infty$, exit;
2. **If** k labels are matched, output x, set it to the next object matching α, and go to 1;
   Otherwise, set α to the next label from Q, in cyclic order;
3. **If** x has matches α, go to 2;
   Otherwise, set x to the next object matching α, and go to 1. □

3.2 File System Search

We introduce a new type of query to search in labeled and multi-labeled trees, that corresponds to one of the most natural search query that one can perform in a file-system.

Definition 2 (Unordered Path-Subset Query). *Given a multi-labeled tree and a set Q of k labels, find the set of nodes x, such that:*

1. *the rooted path to x contains nodes matching all the labels from Q; and,*
2. *this path contains no node satisfying* (1) *other than x.*

Such queries are motivated by the search in file systems, where the result corresponds to folders or files whose path matches the set of keywords. Multi-labeled trees associate several keywords with each folder or file (such as the words and extension composing its name) in an index of the file-system. Using techniques similar to those used for the intersection problem, we prove the following result:

Theorem 4. *Consider a Multi-Labeled Tree of n nodes and σ labels, associated by t pairs. Given an unordered path-subset query composed of k labels, there is an algorithm solving it which performs $\mathcal{O}(\delta k)$ operator calls in time $\mathcal{O}(\delta k \lg\lg\sigma)$, where δ is the minimum number of operation performed by a non-deterministic algorithm to solve the query.*

Proof (sketch). Suppose that x is initialized to the root of the tree and that α is initialized to the first label of the query. If we consider the nodes in pre-order, and introduce the bogus node ∞ that matches all labels and is a successor to all nodes, our algorithm proceeds as follow:

1. **If** $x = \infty$, exit;
2. **If** k labels are matched, output x, set it to the next node matching α, and go to 1; **Otherwise**, set α to the next label from Q in cyclic order;
3. **If** x has an ancestor labeled α, go to 2;
4. **If** x has a descendant labeled α, set it to the first such descendant, and go to 2; **Otherwise**, set x to the next node matching α, and go to 1.

This algorithm cycles through the labels in the query set, maintains in x the lowest node of the current potential match, counts how many labels are currently matched, and eventually outputs the nodes matching the query.

The pre-order rank of successive nodes pointed to by x is strictly increasing at each update, so that at any time, all pre-order predecessors of x have been considered and have been output when adequate. Every k iterations of the loop the algorithm considered at least as many nodes as a non-deterministic algorithm would have in a single operation: it takes at most k steps to eliminate as many potential result nodes as a non-deterministic algorithm, which can "guess" which operation to perform to eliminate the largest number of potential result nodes.

When the pre-order rank of x reaches its final value, all nodes have been considered (hence the correctness), and the algorithm has performed $2\delta k$ operator calls where a non-deterministic algorithm would have performed at least δ (hence the complexity result). □

We now prove that the number of operator calls performed by the above algorithm is optimal for deterministic algorithms:

Lemma 1. *Consider any deterministic algorithm Alg solving unordered path-subset queries, and $\delta \geq 1$, $k \geq 2$, $n \geq \delta(2k{+}1) + 1$, $\sigma \geq 2k + 1$, and $t \geq n$. There is a random distribution $\mathcal{D}$ on multi-labeled trees of $\mathcal{O}(n)$ objects and $\mathcal{O}(\sigma)$ labels, associated with $\mathcal{O}(t)$ pairs, and an unordered path-subset query composed of k labels which can be solved by a non-deterministic algorithm in at most $\mathcal{O}(\delta)$ operations on any multi-labeled trees from $\mathcal{D}$, such that Alg performs $\Omega(\delta k)$ operator calls on average to solve instances from $\mathcal{D}$.*

Proof (sketch). We define a distribution $\mathcal{D}$ on multi-labeled trees with δ branches of $2k+1$ nodes such that any non-deterministic algorithm can show that the unordered path-subset query composed of labels $\{1, \ldots, k\}$ has no match in δ operations. We prove the lower bound by showing that no deterministic algorithm can check that this query has no match in less than δk operations on average. □

The result on deterministic algorithms from Lemma 1 combines trivially with the Yao-von Neumann principle [11, 12, 14] to prove a lower bound on the complexity of any randomized algorithm:

Theorem 5. *Consider any randomized algorithm RandAlg solving unordered path-subset queries, and $\delta \geq 1$, $n \geq \delta(2k+1)+1$, $k \geq 2$, $\sigma \geq 2k+1$, and $t \geq n$. There is a Multi-Labeled tree of $\mathcal{O}(n)$ nodes and $\mathcal{O}(\sigma)$ labels, associated in $\mathcal{O}(t)$ pairs, and an unordered path-subset query composed of k labels which can be solved by a non-deterministic algorithm in at most $\mathcal{O}(\delta)$ operations, such that RandAlg performs on average $\Omega(\delta k)$ operator calls to answer the query.*

The proofs of these results is similar to their counterparts on the intersection problem [3]. In particular, Theorems 4 and 5 show that a deterministic algorithm performs as well as any randomized algorithm for unordered path-subset queries, in terms of the number of operator calls.

4 Conclusion

We considered succinct data structures for binary relations, labeled trees and multi-labeled trees, and their application to search algorithms in those structures. Our results are threefold:

- first, we give two succinct encodings for binary relations using asymptotically optimal space and efficiently supporting in different time trade-offs the rank and select operators on the rows and columns of the relation;
- second, we give a new representation for labeled trees, that we combine with binary relations to represent multi-labeled trees;
- Third, we show that those encodings have applications to conjunctive queries in binary relations and unordered path-subset queries in multi-labeled trees, such as XML documents or file-system indexes.

Obvious research prospects are to extend the range of operators supported (e.g. labeled child queries), and to apply similar encodings to other types of queries (e.g. ordered sub-path, Twig Pattern and XPath queries).

References

1. R. A. Baeza-Yates. A fast set intersection algorithm for sorted sequences. In *Proceedings of the 15th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 3109 of *LNCS*, pages 400–408. Springer, 2004.
2. J. Barbay. Optimality of randomized algorithms for the intersection problem. In A. Albrecht, editor, *Proceedings of the Symposium on Stochastic Algorithms, Foundations and Applications (SAGA)*, volume 2827 / 2003, pages 26–38. Springer-Verlag Heidelberg, 2003.

3. J. Barbay and C. Kenyon. Adaptive intersection and t-threshold problems. In *Proceedings of the 13th ACM-SIAM Symposium On Discrete Algorithms (SODA)*, pages 390–399, 2002.
4. D. R. Clark and J. I. Munro. Efficient suffix trees on secondary storage. In *Proceedings of the 7th annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 383–391, Philadelphia, PA, USA, 1996.
5. E. D. Demaine, A. López-Ortiz, and J. I. Munro. Adaptive set intersections, unions, and differences. In *Proceedings of the 11th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 743–752, 2000.
6. E. D. Demaine, A. López-Ortiz, and J. I. Munro. Experiments on adaptive set intersections for text retrieval systems. In *Proceedings of the 3rd Workshop on Algorithm Engineering and Experiments, Lecture Notes in Computer Science*, pages 5–6, Washington DC, January 2001.
7. P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Structuring labeled trees for optimal succinctness, and beyond. In *Proc. 46th IEEE Symposium on Foundations of Computer Science (FOCS '05)*, pages 184–196, 2005.
8. R. F. Geary, R. Raman, and V. Raman. Succinct ordinal trees with level-ancestor queries. In *Proceedings of the 15th annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1–10. Society for Industrial and Applied Mathematics, 2004.
9. A. Golynski, J. I. Munro, and S. S. Rao. Rank/select operations on large alphabets: a tool for text indexing. In *Proceedings of the 17th annual ACM-SIAM symposium on Discrete algorithm (SODA)*, pages 368–373, 2006.
10. G. Jacobson. Space-efficient static trees and graphs. In *Proceedings of the 30th annual Symposium on Foundations of Computer Science (FOCS'89)*, pages 549–554, 1989.
11. J. V. Neumann and O. Morgenstern. *Theory of games and economic behavior*. 1st ed. Princeton University Press, 1944.
12. M. Sion. On general minimax theorems. *Pacific Journal of Mathematics*, pages 171–176, 1958.
13. D. E. Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. *Information Processing Letters*, 17(2):81–84, Aug. 1983.
14. A. C. Yao. Probabilistic computations: Toward a unified measure of complexity. In *Proc. 18th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 222–227, 1977.

Theoretical and Practical Improvements on the RMQ-Problem, with Applications to LCA and LCE

Johannes Fischer and Volker Heun

Institut für Informatik der Ludwig-Maximilians-Universität München
Amalienstr. 17, D-80333 München, Germany
{Johannes.Fischer, Volker.Heun}@bio.ifi.lmu.de

Abstract. The Range-Minimum-Query-Problem is to preprocess an array such that the position of the minimum element between two specified indices can be obtained efficiently. We present a direct algorithm for the general RMQ-problem with linear preprocessing time and constant query time, without making use of any dynamic data structure. It consumes less than half of the space that is needed by the method by Berkman and Vishkin. We use our new algorithm for RMQ to improve on LCA-computation for binary trees, and further give a constant-time LCE-algorithm solely based on arrays. Both LCA and LCE have important applications, e.g., in computational biology. Experimental studies show that our new method is almost twice as fast in practice as previous approaches, and asymptotically slower variants of the constant-time algorithms perform even better for today's common problem sizes.

1 Introduction

The problem of finding the *lowest common ancestor* (LCA) of a pair of nodes in a tree has attracted much attention in the past three decades, starting with Aho et al. [1]. It is not only algorithmically beautiful, but also has numerous applications, most importantly in the area of string processing and computational biology, where LCA is often used in conjunction with suffix trees. There are several variants of the problem (see [2]), the most prominent being the one where the tree is static and known in advance, and there are several queries to be answered on-line. In this case it makes sense to spend some time on *preprocessing* the tree in order to answer future queries faster. In their seminal paper [2], Harel and Tarjan showed that an intrinsic preprocessing in time *linear* in the size of the tree is sufficient to answer LCA-queries in *constant* time. Their algorithm was later simplified by Schieber and Vishkin [3], but remained rather complicated.

A major breakthrough in practicable constant-time LCA-computation was made by Berkman and Vishkin [4], and again, in a simplified presentation, by Bender et al. [5, 6]. The key idea for this algorithm is the connection between LCA-queries on trees and *range minimum queries* on arrays (RMQs). Basically, an RMQ asks for the position of the minimum element between two specified indices, and this problem was shown to be linearly equivalent to the LCA-problem

M. Lewenstein and G. Valiente (Eds.): CPM 2006, LNCS 4009, pp. 36–48, 2006.

by Gabow et al. [7], in the sense that both problems can be transformed into each other in time linear in the size of the input. The reduction from LCA to RMQ is in fact a reduction to a *restricted* version of RMQ, where consecutive array elements differ by exactly 1. The authors give an algorithm for this restricted version of RMQ, which is then used to answer LCA-queries.

However, RMQs are not only of interest because they can be used to answer LCA-queries, but have their own right to exist. A recent trend in text indexing tries to substitute the powerful but rather space-consuming *suffix tree* by alternative array-based data structures, most prominently the *suffix array*, discovered independently by Gonnet et al. [8] and by Manber and Myers [9]. While this data structure supports string searches in time almost as good as suffix trees, Kasai et al. [10] and Abouelhoda et al. [11] went one step further and showed that the addition of another array to the suffix array, namely the LCP-array, is sufficient to simulate full tree traversals of the suffix tree. It is thus possible to change many (but not all) algorithms based on suffix trees such that they operate on arrays only. One important exception to this are algorithms that rely on constant-time LCA-retrieval, such as computing longest common extensions of strings (LCEs), and all algorithms based on constant-time LCE-computations.

It is well-known that LCA-queries on the leaves of a suffix tree correspond to RMQs on the LCP-array. So an algorithm that solves the RMQ-problem would make it possible to re-formulate many algorithms based on suffix trees *and* LCA-retrieval such that they operate on arrays only. Unfortunately, the LCP-array does not exhibit the nice property that subsequent elements differ by exactly one, so the algorithm for the restricted RMQ-problem cannot immediately be used for this purpose. Gabow et al. [7] give an algorithm to reduce the general RMQ-problem to the LCA-problem by transforming the array into a special kind of tree. Their method, explained in more detail in Sect. 2.2, has two major drawbacks: First, it doubles the size of the input, and second, even more importantly, it relies on dynamic structures (trees) during the preprocessing. This resembles the suffix-tree/suffix-array duality: It *is* possible to infer the array from the tree; nevertheless, direct construction algorithms for the array are well studied.

Our paper overcomes this very dilemma by presenting the first[1] *direct* algorithm for the general RMQ-problem with linear preprocessing time and constant query time, without making use of *any* dynamic data structure (Sect. 3). It is also less space-consuming than previous approaches, as it uses only $4n + O(\sqrt{n} \log n)$ words of extra space, a major improvement compared with the $9n + O(\sqrt{n} \log^2 n)$ words plus the space for the tree used by the currently best algorithm. (Both O-constants are small.) In Sect. 4, we stress the impact of our new method by showing that it leads to improvements in the LCA-computation for binary trees, and further to the first constant-time LCE-algorithm solely based on arrays. In Sect. 5, we show that our RMQ-method is faster in practice than previous constant-time approaches (and therefore also the methods from Sect. 4). We will also see that for today's common problem sizes it makes more sense to use methods that answer long queries in constant time, but short queries in time logarithmic in

[1] By the time of writing we were unaware of another direct algorithm for RMQ [12].

the query length. These asymptotically *slower* RMQ-algorithms are slightly less space consuming than the constant-time approaches, and also faster in practice.

2 Definitions and Previous Results

The *Range Minimum Query* (RMQ) problem is defined as follows: given an array $A[1,n]$ of elements from a totally ordered set (with order relation "$\leq$"), $\textsc{rmq}_A(i,j)$ returns the index of a smallest element in $A[i,j]$, i.e., $\textsc{rmq}_A(i,j) = \arg\min_{k\in\{i,\dots,j\}}\{A[k]\}$. (The subscript A will be omitted if the context is clear.) The most naive algorithm for this problem searches the array from i to j each time a query is presented, resulting in $O(n)$ query time. As mentioned in the introduction, we consider the variant where A is first preprocessed in order to answer future queries faster. Following the notation from [6], we say that an algorithm with preprocessing time $p(n)$ and query time $q(n)$ has complexity $\langle p(n), q(n)\rangle$. Thus, the naive method described above would be $\langle O(1), O(n)\rangle$, because it requires no preprocessing.

The following definition [13] will be central for both our algorithm and that of [4].

Definition 1. *A* Cartesian Tree *of an array $A[l,r]$ is a binary tree $\mathcal{C}(A)$ whose root is a minimum element of A, labeled with the position i of this minimum. The left child of the root is the Cartesian Tree of $A[l,i-1]$ if $i > l$, otherwise it has no left child. The right child is defined similarly for $A[i+1,r]$.*

Note that $\mathcal{C}(A)$ is not necessarily unique if A contains equal elements. To overcome this problem, we impose a *strong* total order "$\prec$" on A by defining $A[i] \prec A[j]$ iff $A[i] < A[j]$, or $A[i] = A[j]$ and $i < j$. The effect of this definition is just to consider the 'first' occurrence of equal elements in A as being the 'smallest'. Defining a Cartesian Tree over A using the $\prec$-order gives a *unique* tree $\mathcal{C}^{\text{can}}(A)$, which we call the *Canonical Cartesian Tree*. Note also that this order results in unique answers for the RMQ-problem, because the minimum is unique.

In [6] an algorithm for constructing $\mathcal{C}^{\text{can}}(A)$ is given as follows. Let $\mathcal{C}_i^{\text{can}}(A)$ be the Canonical Cartesian Tree for $A[1,i]$. Then $\mathcal{C}_{i+1}^{\text{can}}(A)$ is obtained by climbing up from the rightmost leaf of $\mathcal{C}_i^{\text{can}}(A)$ to the root, thereby finding the position where $A[i+1]$ belongs. To be precise, let $v_1,\dots,v_k$ be the nodes of the rightmost path in $\mathcal{C}_i^{\text{can}}(A)$ with labels $l_1,\dots,l_k$, respectively, where v_1 is the root and v_k is the rightmost leaf. Let m be defined such that $A[l_m] \leq A[i+1]$ and $A[l_{m'}] > A[i+1]$ for all $m < m' \leq k$. To build $\mathcal{C}_{i+1}^{\text{can}}(A)$, create a new node w with label $i+1$ which becomes the right child of v_m, and the subtree rooted at v_{m+1} becomes the left child of w. This process inserts each element to the rightmost path exactly once, and each comparison removes one element from the rightmost path, resulting in a total $O(n)$ construction time to build $\mathcal{C}^{\text{can}}(A)$.

2.1 An $\langle O(n\log n), O(1)\rangle$-Algorithm for RMQ

We briefly present a simple method [6] to answer RMQs in constant time using $O(n\log n)$ space. This algorithm will be used to answer 'long' RMQs both in our algorithm and that of [4][2]. The idea is to precompute all RMQs whose length is a power of two. For every $1 \le i \le n$ and every $1 \le j \le \lfloor \log n \rfloor$ compute the position of the minimum in the sub-array $A[i, i+2^j-1]$ and store the result in $M[i][j]$. Table M occupies $O(n \log n)$ space and can be filled in optimal time by using the formula $M[i][j] = \arg\min_{k \in \{M[i][j-1], M[i+2^{j-1}][j-1]\}}\{A[k]\}$. To answer $\text{RMQ}(i,j)$, select two *overlapping* blocks that exactly cover the interval $[i,j]$, and return the position where the overall minimum occurs. Precisely, let $l = \lfloor \log(j-i) \rfloor$. Then $\text{RMQ}(i,j) = \arg\min_{k \in \{M[i][l], M[j-2^l+1][l]\}}\{A[k]\}$.

2.2 The $\langle O(n), O(1) \rangle$-Algorithm for RMQ by Berkman and Vishkin

This section describes the solution to the general RMQ-problem as a combination of the results obtained in [4] and [7]. We follow the presentation from [6].

±1RMQ is a special case of the RMQ-problem, where consecutive array elements differ by exactly 1. The solution to the general RMQ-problem given in [4] (from now on called Berkman-Vishkin algorithm) starts by reducing RMQ to ±1RMQ as follows: given an array $A[1,n]$ to be preprocessed for RMQ, build $\mathcal{C}^{\text{can}}(A)$ as shown above. Then perform a Euler Tour[3] in this tree, storing the labels of the visited nodes in an array $E[1, 2n-1]$, and their respective heights in $H[1, 2n-1]$. Further, store the position of the first occurrence of $A[i]$ in the Euler Tour in a representative array $R[1,n]$. The Cartesian Tree is not needed anymore once the arrays E, H and R are filled, and can thus be deleted. The paper then shows that $\text{RMQ}_A(i,j) = E[\pm 1\text{RMQ}_H(R[i], R[j])]$. Note in particular the doubling of the input when going from A to H; i.e., H has size $n' := 2n-1$. We now sketch the solution to the ±1RMQ-problem.

To solve ±1RMQ on H, partition H into blocks of size $\frac{\log n'}{2}$.[4] Define two arrays A' and B of size $\frac{2n'}{\log n'}$, where $A'[i]$ stores the minimum of the ith block in H, and $B[i]$ stores the position of this minimum in H. Now A' is preprocessed using the algorithm from Sect. 2.1, occupying $O(\frac{2n'}{\log n'} \log \frac{2n'}{\log n'}) = O(n)$ space. This preprocessing enables out-of-block queries (i.e., queries that span over several blocks) to be answered in $O(1)$. It remains to show how in-block-queries are handled. This is done with the so-called Four-Russians-Trick [15] where one precomputes the answers to all possible queries when the number of possible instances is sufficiently small. The authors of [6] noted that due to the ±1-property there are only $O(\sqrt{n'})$ blocks to be precomputed: we can virtually subtract the initial value of a block from each element without changing the answers to the RMQs; this enables us to describe a block by a ±1-vector of length $2^{1/2 \log n' - 1} = O(\sqrt{n'})$. For each such block precompute all $\frac{1}{2}\frac{\log n'}{2}(\frac{\log n'}{2}+1)$ possible RMQs and store them in a table P of total size $O(\sqrt{n'}\log^2 n') = O(n)$. To index table P, precompute the *type* of each block and store it in array $T[1, \frac{2n'}{\log n'}]$.

[2] The original description in [4] used a slightly more complicated algorithm, which is, however, equivalent to the one presented here.

[3] The name "Euler Tour" is derived from the Euler Tour-technique [14], and is not to be confused with a Eulerian circuit.

[4] For a simpler presentation we omit floors and ceilings from now on.

The block type is simply the binary number obtained by comparing subsequent elements in the block, writing a 0 at position i if $H[i+1] = H[i] + 1$ and 1 otherwise. Table 1 summarizes the tables needed for the algorithm and their sizes (ignore the last column for now).

Now, to answer $\textsc{rmq}(i, j)$, if i and j occur in different blocks, compute (1) the minimum from i to the end of i's block using arrays T and P, (2) the minimum of all blocks between i's and j's block using the precomputed queries on A' stored in table M, and (3) the minimum from the beginning of j's block to j, again using T and P. Finally, return the position where the overall minimum occurs, possibly employing B. If i and j occur in the same block, just answer an in-block-query from i to j. In both cases, the time needed for answering the query is constant.

3 An Improved $\langle O(n), O(1)\rangle$-Algorithm for RMQ

Our aim is to solve the general RMQ-problem *without* constructing the Cartesian Tree first; in fact, without employing *any* dynamic data structure such as trees. We also wish to find a solution that does not double the input array, as the Berkman-Vishkin algorithm does. The key to our solution is the following theorem. (From now on, we assume that the $\prec$-relation is used for answering RMQs, such that the answers become unique.)

Theorem 1. *Let A and B be two arrays, both of size n. Then $\textsc{rmq}_A(i, j) = \textsc{rmq}_B(i, j)$ for all $1 \le i \le j \le n$ if and only if $\mathcal{C}^{can}(A) = \mathcal{C}^{can}(B)$.*

Proof. It is easy to see that $\textsc{rmq}_A(i, j) = \textsc{rmq}_B(i, j)$ for all $1 \le i \le j \le n$ iff the following three conditions are satisfied: (i) The minimum under "$\prec$" occurs at the same position m, i.e., $\arg\min A = \arg\min B = m$. (ii) $\forall 1 \le i \le j < m : \textsc{rmq}_{A[1,m-1]}(i, j) = \textsc{rmq}_{B[1,m-1]}(i, j)$. (iii) $\forall m < i \le j \le n : \textsc{rmq}_{A[m+1,n]}(i, j) = \textsc{rmq}_{B[m+1,n]}(i, j)$. Due to the definition of the Canonical Cartesian Tree, points (i)–(iii) are true if and only if the root of $\mathcal{C}^{\text{can}}(A)$ equals the root of $\mathcal{C}^{\text{can}}(B)$, and $\mathcal{C}^{\text{can}}(A[1, m-1]) = \mathcal{C}^{\text{can}}(B[1, m-1])$, and $\mathcal{C}^{\text{can}}(A[m+1, n]) = \mathcal{C}^{\text{can}}(B[m+1, n])$. This is true iff $\mathcal{C}^{\text{can}}(A) = \mathcal{C}^{\text{can}}(B)$. □

It is well known that the number of binary trees with n nodes is C_n, where $C_n = \frac{1}{n+1}\binom{2n}{n} = 4^n/(\sqrt{\pi} n^{3/2})(1 + o(1))$ is the nth *Catalan Number.*

Lemma 1. *It is possible to precompute the answers to all possible range minimum queries on arrays of size s in a table P of size $O(4^s\sqrt{s})$.*

Proof. Because the Cartesian Tree is a binary tree with s nodes, table P has $O(\frac{4^s}{s^{3/2}})$ rows for each possible type of block. For each type we need to precompute $\textsc{rmq}(i, j)$ for all $1 \le i \le j \le s$, so the number of columns in P is $O(s^2)$. □

We now come to the description of our $\langle O(n), O(1)\rangle$-algorithm for the general RMQ-problem. Like the ±1RMQ-algorithm presented in Sect. 2.2 it is an application of the Four-Russians-Trick. However, Lemma 1 allows us to apply the trick

Table 1. Additional space needed by the $\langle O(n), O(1)\rangle$-algorithms for RMQ (in words)

Array/Table	Berkman-Vishkin	our algorithm
E, H, R	$2(2n-1)+n=5n-2$	(arrays not needed)
$A\ , B, T$	$3\frac{2n}{\log(2n)/2}=12n/\log(2n)$	$3\frac{n}{\log(n)/4}=12n/\log n$
M	$4n+4n/\log(2n)-4n\log\log(2n)/\log(2n)$	$4n+8n/\log n-4n\log\log n/\log n$
P	$\sqrt{n}\log^2 n/(8\sqrt{2})(1+o(1))$	$\sqrt{n}\log^{1/2} n/(4\sqrt{\pi})(1+o(1))$
total (simpl.)	$9n+O(\sqrt{n}\log^2 n)$	$4n+O(\sqrt{n}\log n)$

to *any* array (not only to those with the ±1-property), which leads to substantial improvements. Start by partitioning the array A into blocks $B_1, \ldots, B_{n/s}$ of size $s := \frac{\log n}{4}$. Define two arrays A' and B of size $n/s = \frac{4n}{\log n}$, where $A'[i]$ stores the minimum of block B_i, and $B[i]$ stores the position of this minimum in A. Now A' is preprocessed using the algorithm from Sect. 2.1, occupying $O(\frac{4n}{\log n}\log\frac{4n}{\log n}) = O(n)$ space. Then precompute the answers to all possible queries on arrays of size s and store the results in a table P. According to Lemma 1, this table occupies $O(4^{(\log n)/4}(\frac{\log n}{4})^{1/2}) = O(n)$ space. Finally, compute the type of each block in A and store these values in array $T[1, \frac{4n}{\log n}]$. As this is not as obvious as in Sect. 2.2, it is explained in detail in the following subsection. A query $\textsc{rmq}(i, j)$ is now answered exactly as explained in the last paragraph of Sect. 2.2, namely by comparing at most three minima, depending on the blocks where i and j occur. Again, the time for answering a query is constant, leading to the $\langle O(n), O(1)\rangle$ time bounds stated before. See Table 1 for a comparison of the two methods (space for $\mathcal{C}(A)$ not included).

3.1 Computing the Block Types

In order to index table P, it remains to show how to fill array T; i.e., how to compute the types of the blocks B_i occurring in A in linear time. Thm. 1 implies that there are only C_s different types of arrays of size s, so we are looking for a surjection

$$\mathit{type}\colon \mathcal{A}_s \to \{0, \ldots, C_s - 1\},\ \text{and}\ \mathit{type}(B_i)=\mathit{type}(B_j)\ \text{iff}\ \mathcal{C}^{\text{can}}(B_i)=\mathcal{C}^{\text{can}}(B_j), \quad (1)$$

where $\mathcal{A}_s$ is the set of arrays of size s. The reason for requiring that B_i and B_j have the same Canonical Cartesian Tree is given by Thm. 1 which tells us that in such a case both blocks share the same RMQs. The most naive way to calculate the type would be to actually *construct* the Cartesian Tree of each block, and then use an inverse enumeration of binary trees [16] to compute its type. This, however, would counteract our aim to avoid dynamic data structures. The algorithm in Fig. 1 shows how to compute the block type directly. It makes use of the so-called *ballot numbers* C_{pq} [16], defined by

$$C_{00} = 1, C_{pq} = C_{p(q-1)} + C_{(p-1)q},\ \text{if}\ 0 \le p \le q \ne 0,\ \text{and}\ C_{pq} = 0\ \text{otherwise.} \quad (2)$$

It can be proved that a closed formula for C_{pq} is given by $\frac{q-p+1}{q+1}\binom{p+q}{p}$ [16], which immediately implies that C_{ss} equals the s'th Catalan number C_s. If we look at

```
   Input: block $B_j$ of size $s$
   Output: $type(B_j)$
 1 let $rp$ be an array of size $s+1$
 2 $rp[1] \leftarrow -\infty$
 3 $q \leftarrow s, N \leftarrow 0$
 4 for $i \leftarrow 1,\ldots,s$ do
 5     while $rp[q+i-s] > B_j[i]$ do
 6         $N \leftarrow N + C_{(s\ \ i)q}$
 7         $q \leftarrow q-1$
 8     end
 9     $rp[q+i+1-s] \leftarrow B_j[i]$
10 end
11 return $N$
```

Fig. 1. An algorithm to compute the type of a block

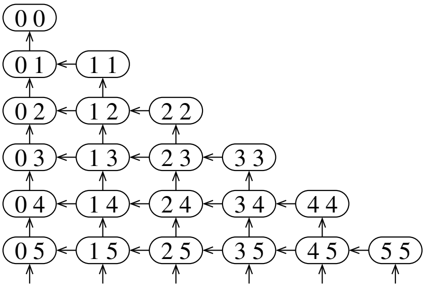


Fig. 2. The infinite graph arising from the definition of the ballot numbers. Its vertices are $\boxed{p\ q}$ for all $0 \le p \le q$. There is an edge from $\boxed{p\ q}$ to $\boxed{(p-1)\ q}$ if $p > 0$ and to $\boxed{p\ (q-1)}$ if $q > p$.

the infinite directed graph shown in Fig. 2 then C_{pq} is clearly the number of paths from $\boxed{p\ q}$ to $\boxed{0\ 0}$, because of (2). This interpretation will be important for the proof of the following

Theorem 2. *The algorithm in Fig. 1 correctly computes the type of a block B_j of size s in $O(s)$ time, i.e., it computes a function satisfying the conditions given in* (1)*.*

Proof. (Sketch.) Intuitively, the algorithm simulates the algorithm for constructing $\mathcal{C}^{\text{can}}(B_j)$ given in Sect. 2. First note that array $rp[1, s+1]$ simulates the stack containing the labels of the nodes on the rightmost path of the partial Canonical Cartesian Tree $\mathcal{C}_i^{\text{can}}(B_j)$, with $q+i-s$ pointing to the top of the stack (i.e., the rightmost leaf), and $rp[1]$ acting as a 'stopper'. Now let l_i be the number of times the while-loop (lines 5–8) is executed during the ith iteration of the outer for-loop. Note that l_i equals the number of elements that are removed from the rightmost path when going from $\mathcal{C}_{i-1}^{\text{can}}(B_j)$ to $\mathcal{C}_i^{\text{can}}(B_j)$. Because one cannot remove more elements from the rightmost path than one has inserted, and each element is removed at most once, we have $\sum_{k=1}^{i} l_k \le i$ for all $1 \le i \le s$. Thus, the sequence $l_1, \ldots, l_s$ corresponds to a path from $\boxed{s\ s}$ to $\boxed{0\ 0}$ in Fig. 2 (and vice versa): in step i, go l_i steps upwards and one step to the left, and after step s go upwards until reaching $\boxed{0\ 0}$. The current position in the graph is $\boxed{(s-i+1)\ q}$, so every time one makes an upward step, N is incremented by the number of paths that have been 'skipped' by going upwards (line 6). This is exactly $C_{(s-i)q}$, the value of the cell to the left of the current one. The effect of this incrementation is that paths going from the current position to the left are assigned lower numbers than paths going upwards.

The proof is completed by noting that a Canonical Cartesian Tree can be uniquely described by $l_1, \dots, l_s$ satisfying $\sum_{k=1}^{i} l_k \leq i$ for all $1 \leq i \leq s$. □

4 Applications

This section sketches two easy (but non-trivial) new results on LCA and LCE that can be obtained with our RMQ-algorithm. Apart from yielding simpler and less space-consuming methods, we will see in Sect. 5 that one can also expect improvements in running times.

4.1 A Space Saving Algorithm for LCA on Binary Trees

The LCA-problem [1] is formally defined as follows: given a rooted tree T with n nodes and two vertices v and w, find the deepest node $\textsc{lca}_T(v, w)$ which is an ancestor of both v and w. Again, we consider the variant where T is static and the queries are posed on-line. As mentioned in the introduction, the RMQ- and the LCA-problem are closely related. In [7], it has been shown that an LCA-query on T basically corresponds to a ±1RMQ-query on the heights of the nodes visited during an Euler-Tour in T. Because the size of an Euler-Tour is exactly $2n - 1$, this leads to an input doubling. We show in this section that using the algorithm presented in Sect. 3 overcomes this problem for binary trees.

Let T be a rooted binary tree with n nodes. First perform an *inorder tree walk* in T and store it in an array $I[1, n]$. Further, store the heights of each node in $H[1, n]$, i.e., $H[i]$ is the height of node $I[i]$ in T. Finally, let R be the inverse array of I, i.e., $I[R[i]] = i$. It is then easy to see that $\textsc{lca}_T(v, w) = I[\textsc{rmq}_H(R[v], R[w])]$: the elements in I between $R[v]$ and $R[w]$ are exactly the nodes encountered between v to w during an inorder tree walk in T, so the range minimum query returns the position k in H of the shallowest such nodes. As the LCA of v and w must be encountered between v and w during the inorder tree walk, $\textsc{lca}(v, w)$ is given by $I[k]$.

The extra space needed is $7n + O(\sqrt{n \log n})$ words: $4n + O(\sqrt{n \log n})$ words from Table 1 for the RMQ-preprocessing, plus $3n$ words for the arrays I, H and R. This *is* an improvement compared with the $9n + O(\sqrt{n} \log^2 n)$ words needed if one were to use the LCA-algorithm presented in [4]. We note that our result could also be generalized to arbitrary trees; the space reduction, however, is only relevant if the number of internal nodes is relatively close to the number of leaves.

4.2 An Improved Algorithm for Longest Common Extensions

The problem of *longest common extensions* is defined for a static string t of size n: given two indices i and j, $\textsc{lce}_t(i, j)$ returns in $O(1)$ the length of the longest common prefix of t's suffixes starting at position i and j; i.e., $\textsc{lce}_s(i, j) = \max\{k : t_{i,\dots,k} = t_{j,\dots,k}\}$.[5] The problem has numerous applications in string

[5] LCE is often defined for *two* strings t and t s.th. i is an index in t and j in t. This can be transformed to our definition by setting $t = t \# t$, where # is a new symbol.

matching, e.g., for tandem repeats [17, 18], approximate tandem repeats [19], and inexact pattern matching [20, 21]. The easiest solution [22] to LCE combines suffix trees with constant-time LCA-retrieval: build a suffix tree T for t and preprocess it for LCA-queries. Then $\textsc{lce}(i, j)$ is given by the height of node $\textsc{lca}(v_i, v_j)$, where v_i and v_j are the leaves corresponding to suffix i and j, respectively.

The crucial point to observe is that the LCA-queries are only posed on the *leaves* of the suffix tree T for t. It is well-known [22,11] that there is a one-to-one correspondence between the leaves of T and the elements of the corresponding *suffix array* [8,9] SA, and also between the heights of T's internal nodes and the *LCP-array* LCP for SA. Basically, SA describes the order of the suffixes of t, and LCP stores the lengths of the longest common prefixes of t that are consecutive in SA. This gives us all the ingredients we need for our new LCE-algorithm: compute SA and its inverse SA^{-1} for t.[6] Further, compute the LCP-array for t in linear time [10, 24] and store it in LCP. (SA is not further needed at this point and can thus be deleted.) Then prepare LCP for RMQs as presented in Sect. 3. It is now easy to see that $\textsc{lce}(i, j) = \textsc{rmq}_{\mathsf{LCP}}(\mathsf{SA}^{-1}[i] + 1, \mathsf{SA}^{-1}[j])$.

Note that this is the first algorithm that solves the LCE-problem without using trees of any form.[7] Apart from SA^{-1} and LCP, the space needed is $4n + O(\sqrt{n \log n})$ words. Compare this with $9n + O(\sqrt{n} \log^2 n)$ words plus the space for the Cartesian Tree that would be needed if one were to preprocess LCP for RMQ using the Berkman-Vishkin algorithm (not to talk about the solution based on suffix trees).

5 Practical Considerations

We now wish to evaluate the practical performance of our new algorithm by comparing it with the Berkman-Vishkin algorithm. We further include three non-$\langle O(n), O(1)\rangle$-algorithms in our evaluation:

1. An algorithm that divides the array into blocks of size $\frac{\log n}{2}$ and preprocesses the block-minima for the out-of-block queries (i.e., it creates table A', B, T and M). The in-block-queries are handled naively (i.e., table P is *not* created). Call this method $\langle O(n), O(\log n)\rangle_2$.
2. The same as above with block size $\frac{\log n}{4}$. Call this method $\langle O(n), O(\log n)\rangle_4$.
3. The naive $\langle O(1), O(n)\rangle$-algorithm that requires no preprocessing.

We performed all tests on an Athlon XP3000 with 2GB of RAM under Linux. All programs were written in C++ and compiled using the same compiler options. All our figures are averages over 5 repetitions of each experiment.

[6] There are fast algorithms that construct SA and its inverse with only $o(n)$ extra space, e.g., [23].

[7] While this has the consequence that the algorithms [17,19,20,21] can be implemented without trees, it is not immediately obvious how to do this for [18] because it uses the tree structure also for representing the tandem repeats.

Fig. 3 shows the time spent on preprocessing by all methods except the naive one, because the latter does no preprocessing. As expected, the Berkman-Vishkin method is the slowest, which is due to the explicit construction of the Cartesian Tree. The preprocessing times for the other three methods are within the same order of magnitude, where our method is slightly slower than the two $\langle O(n), O(\log n)\rangle$-algorithms, as expected.

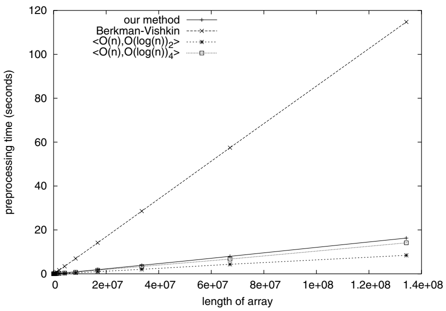


Fig. 3. Preprocessing times for varying array lengths

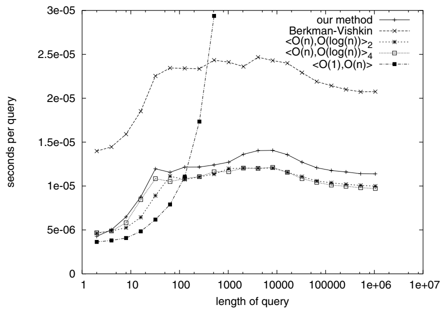
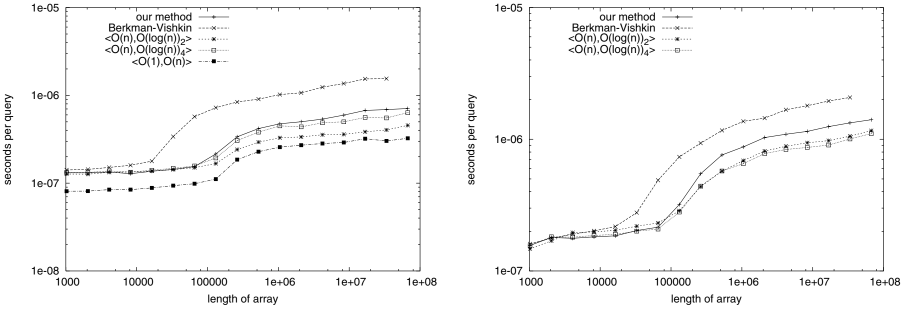


Fig. 4. The influence of different query lengths on the query time (w/o preproc.)

The next test was to evaluate the influence of the query length on the query time. We took a random array of length $n = 10^7$ and posed 10^6 random RMQs on this array. Fig. 4 shows the average query time for all five methods, and there are several points to note.

- As expected, the $\langle O(1), O(n)\rangle$-algorithm behaves linearly in the query length (note the logarithmic x-axis). It is very fast for short queries (up to length 100), but out of the questions for longer queries.
- Our $\langle O(n), O(1)\rangle$-algorithm is about twice as fast as the one by Berkman and Vishkin.
- The two methods with $O(\log n)$ query time are even slightly faster than our constant-time method. This is because quite some arithmetic is necessary to answer the in-block-queries in constant time. With block size $\frac{\log 10^7}{2} \approx 11$ the overhead for this is much too big.
- For all methods except the naive one the query time levels off for very long queries. We can only speculate that this is due to caching phenomena.

In a last test we checked up on the influence of the array length n on the query time. We performed separate tests for short and long random queries, where *short* means to be of length $\log n/2$ such that only in-block-queries are to be handled. *Long* queries were of length $n/100$. The largest arrays that we were able to handle on our computer were of length $\approx 6 \times 10^7$ for both tests. (Because of the input-doubling, the largest array length for the Berkman-Vishkin method was $\approx 3 \times 10^7$ for both tests.) See Fig. 5(a)–(b) for the results. In (a), the naive method is the best, for the same reasons as given before. The other four methods show the same performance as in Fig. 4. For the long queries in (b), the naive method was excluded for obvious reasons. Again, the two $\langle O(n), O(\log n)\rangle$-algorithms perform better than the $\langle O(n), O(1)\rangle$-methods, but our method is about twice as fast as the Berkman-Vishkin algorithm. It is interesting to see that both in (a) and (b) all methods exhibit a significant increase in running time at some point. This happens at roughly $n = 10^5$, whereas the Berkman-Vishkin method has this increase earlier. The effect can most likely be explained by the second-level-cache of the processor. Because of the input-doubling in the Berkman-Vishkin algorithm the cache size is reached earlier for this method. In summary, all our tests show that for practical applications with arrays up to length 10^8 or so it is advisable to use the $\langle O(n), O(\log n)\rangle_2$-algorithm. Unfortunately, our computer is not large enough to test when our algorithm becomes faster than the $\langle O(n), O(\log n)\rangle$-algorithms.



(a) Short queries of length $0.5 \log n$.

(b) Long queries of length $n/100$.

Fig. 5. The influence of different array lengths on the query time (w/o preprocessing)

6 Summary and Discussion

We have seen a new method to answer range minimum queries in constant time after a linear preprocessing step. The key to our algorithm was the strong connection between Cartesian Trees and RMQs, reflected in the employment of the Catalan- and ballot numbers. This led to substantial improvements over previous RMQ-algorithms, namely a space reduction of more than 50%, the complete absence of dynamic data structures, and a boost in query time. We have also seen how our method leads to space reductions in the computation of lowest common ancestors in binary trees, and to an improved algorithm for the computation of longest common extensions in strings. On the practical side, we have seen that it is sometimes wiser to spend a little bit less effort in preprocessing, because even for large problem sizes (arrays up to length 10^8) asymptotically slower algorithms may perform faster in practice.

We finally note that our approach can be combined with the ideas from [25] to give the first *succinct* data structure for constant time RMQ, in the sense that the extra space needed is only $O(n)$ bits. We will elaborate on this in future work.

References

1. Aho, A.V., Hopcroft, J.E., Ullman, J.D.: On finding lowest common ancestors in trees. SIAM J. Comput. **5** (1976) 115–132
2. Harel, D., Tarjan, R.E.: Fast algorithms for finding nearest common ancestors. SIAM J. Comput. **13** (1984) 338–355
3. Schieber, B., Vishkin, U.: On finding lowest common ancestors: Simplification and parallelization. SIAM J. Comput. **17** (1988) 1253–1262
4. Berkman, O., Vishkin, U.: Recursive star-tree parallel data structure. SIAM J. Comput. **22** (1993)
5. Bender, M.A., Farach-Colton, M.: The LCA problem revisited. In: Proc. LATIN, LNCS 1776, Springer (2000) 88–94
6. Bender, M.A., Farach-Colton, M., Pemmasani, G., Skiena, S., Sumazin, P.: Lowest common ancestors in trees and directed acyclic graphs. J. Algorithms **57** (2005) 75–94
7. Gabow, H.N., Bentley, J.L., Tarjan, R.E.: Scaling and related techniques for geometry problems. In: Proc. of the ACM STOC, ACM Press (1984) 135–143
8. Gonnet, G.H., Baeza-Yates, R.A., Snider, T.: New indices for text: PAT trees and PAT arrays. In Frakes, W.B., Baeza-Yates, R.A., eds.: Information Retrieval: Data Structures and Algorithms. Prentice-Hall (1992) 66–82
9. Manber, U., Myers, E.W.: Suffix arrays: A new method for on-line string searches. SIAM J. Comput. **22** (1993) 935–948
10. Kasai, T., Lee, G., Arimura, H., Arikawa, S., Park, K.: Linear-time longest-common-prefix computation in suffix arrays and its applications. In: Proc. CPM, LNCS 2089, Springer (2001) 181–192
11. Abouelhoda, M.I., Kurtz, S., Ohlebusch, E.: Replacing suffix trees with enhanced suffix arrays. J. Discrete Algorithms **2** (2004) 53–86
12. Alstrup, S., Gavoille, C., Kaplan, H., Rauhe, T.: Nearest common ancestors: A survey and a new distributed algorithm. In: Proc. SPAA, ACM Press (2002) 258–264
13. Vuillemin, J.: A unifying look at data structures. Comm. ACM **23** (1980) 229–239
14. Tarjan, R.E., Vishkin, U.: An efficient parallel biconnectivity algorithm. SIAM J. Comput. **14** (1985) 862–874
15. Arlazarov, V.L., Dinic, E.A., Kronrod, M.A., Faradzev, I.A.: On economic construction of the transitive closure of a directed graph. Dokl. Acad. Nauk. SSSR **194** (1970, in Russian) 487–488, Engl. transl. in *Soviet Math. Dokl.*, **11** 1209–1210, 1975
16. Knuth, D.E.: The Art of Computer Programming Vol. 4, Fasc. 4: Generating All Trees; History of Combinatorial Generation. Addison-Wesley (2006)
17. Main, M.G., Lorentz, R.J.: An $O(n \log n)$ algorithm for finding all repetitions in a string. J. Algorithms **5** (1984) 422–432
18. Gusfield, D., Stoye, J.: Linear time algorithm for finding and representing all tandem repeats in a string. J. Comput. Syst. Sci. **69** (2004) 525–546
19. Landau, G., Schmidt, J.P., Sokol, D.: An algorithm for approximate tandem repeats. J. Comput. Biol. **8** (2001) 1–18
20. Myers, E.W.: An $O(nd)$ difference algorithm and its variations. Algorithmica **1** (1986) 251–266

21. Landau, G., Vishkin, U.: Introducing efficient parallelism into approximate string matching and a new serial algorithm. In: Proc. STOC, ACM Press (1986) 220–230
22. Gusfield, D.: Algorithms on Strings, Trees, and Sequences. Cambridge University Press (1997)
23. Schürmann, K.B., Stoye, J.: An incomplex algorithm for fast suffix array construction. In: Proc, ALENEX/ANALCO, SIAM Press (2005) 77–85.
24. Manzini, G.: Two space saving tricks for linear time lcp array computation. In: Proc. SWAT. LNCS 3111, Springer (2004) 372–383
25. Sadakane, K.: Succinct representations of lcp information and improvements in the compressed suffix arrays. In: Proc. SODA, ACM/SIAM (2002) 225–237

A Linear Size Index for Approximate Pattern Matching

Ho-Leung Chan[1], Tak-Wah Lam[1], Wing-Kin Sung[2],
Siu-Lung Tam[1], and Swee-Seong Wong[2]

[1] Department of Computer Science, University of Hong Kong
{hlchan, twlam, sltam}@cs.hku.hk
[2] Department of Computer Science, National University of Singapore
{ksung, wongss}@comp.nus.edu.sg

Abstract. This paper revisits the problem of indexing a text $S[1..n]$ to support searching substrings in S that match a given pattern $P[1..m]$ with at most k errors. A naive solution either has a worst-case matching time complexity of $\Omega(m^k)$ or requires $\Omega(n^k)$ space. Devising a solution with better performance has been a challenge until Cole et al. [5] showed an $O(n \log^k n)$-space index that can support k-error matching in $O(m+occ+\log^k n \log\log n)$ time, where occ is the number of occurrences. Motivated by the indexing of DNA, we investigate in this paper the feasibility of devising a linear-size index that still has a time complexity linear in m. In particular, we give an $O(n)$-space index that supports k-error matching in $O(m + occ + (\log n)^{k(k+1)} \log\log n)$ worst-case time. Furthermore, the index can be compressed from $O(n)$ words into $O(n)$ bits with a slight increase in the time complexity.

1 Introduction

In this paper, we consider the indexing problem for k-approximate matching: given an integer $k \geq 0$ and a text $S[1..n]$ over a finite alphabet Σ, we want to build an index for S such that for any query pattern $P[1..m]$, we can report efficiently all locations in S that match P with at most k errors. The number of errors is measured in terms of either the Hamming distance (number of character substitutions) or the edit distance (number of character substitutions, insertions or deletions). The major concern is how to achieve efficient matching without using a large amount of space for indexing. Typical applications include the indexing of DNA or protein sequences for biological research.

To support exact matching (i.e., $k = 0$), suffix trees and suffix arrays are the most well-known indexes. Suffix trees [15,12] occupy $O(n)$ space and achieve the optimal matching time, i.e., $O(m + occ)$, where occ is the number occurrences of P in S.[1] For suffix arrays [11], the space requirement is also $O(n)$ space (but with a smaller constant), and the matching time is $O(m + occ + \log n)$. Recently, two

[1] Unless otherwise stated, the space complexity is measured in terms of the number of words, where a word can store $O(\log n)$ bits.

M. Lewenstein and G. Valiente (Eds.): CPM 2006, LNCS 4009, pp. 49–59, 2006.

compressed solutions, namely, compressed suffix arrays [7] and FM-index [6], have been proposed; they requires $O(n)$ bits only and the matching time is $O(m + occ \log^{\epsilon} n)$, where $\epsilon > 0$.

Indexing a string for approximate matching is a challenging problem. Even the special case where only one error is allowed (i.e., $k = 1$) has attracted a lot of attention. A simple solution is to use the suffix tree of S and repeatedly search for every 1-error modification of the query pattern; this solution uses $O(n)$ space and the matching time is $O(m^2 + occ)$ [4]. With a bigger index of size $O(n \log n)$, the matching time complexity has been improved tremendously by a chain of results to $O(m \log n \log\log n + occ)$ [1], $O(m \log\log n + occ)$ [2], and finally $O(m + occ + \log n \log\log n)$ [5]. It is also known that indexes using $O(n)$ space takes $O(m \log n + occ)$ time [8] and $O(m \log\log n + occ)$ time [9] for 1-error matching. These two indexes can also be compressed to $O(n)$ bits, and the 1-error matching time is $O(m \log^2 n + occ \log n)$ and $O((m \log\log n + occ) \log^{\epsilon} n)$, respectively, where $\epsilon < 1$.

To cater for $k = O(1)$ errors, one can perform a brute-force search on an one-error index (i.e., repeatedly modify the pattern at different $k - 1$ positions and search for one-error matches); the matching becomes very inefficient, involving a factor of m^k in the time complexity. Alternatively, one can improve the matching time by including all possible erroneous substrings into the index; yet this seems to require $\Omega(n^k)$ space. It has been open whether there exists an index with performance better than a navie solution. The breakthrough is due to Cole et al. [5], who are able to avoid brute-force matching of a pattern with a moderate increase in the index size. Precisely, their index occupies $O(\frac{d^k}{k!} n \log^k n)$ space and supports k-error matching in $O(m + occ + \frac{c^k}{k!} \log^k n \log\log n)$ time for Hamming distance, where d and c are some constants. The term occ is replaced with $occ \cdot 3^k$ for edit distance. This solution gives an obvious improvement to the matching efficiency. The space requirement is acceptable for many applications, but it may be too demanding for indexing DNA sequences or webpages. [2]

In this paper, we focus on indexes that use only $O(n)$ words or $O(n)$ bits for k-error matching, and we hope that the time complexity can be better than $O(m^k)$. Prior to our work, indexes using $O(n)$ words to answer a k-error query takes $O((cm)^k \log n + occ))$ time [8] or takes $O((cm)^k \log\log n + occ))$ time [9]. Indexes using $O(n)$ bits have a slightly worse time complexity [8,9]. See Table 1 for a summary of results. The main results of this paper are as follows.

(i) We give an $O(n)$-word index that supports k-error matching in $O(m + occ + (c \log n)^{k(k+1)} \log\log n)$ time, where c is a constant. Furthermore, if the pattern is known to be long (precisely, $\Omega(\log^{k+1} n)$), the matching time can be improved to $O(m + occ + (c \log n)^{2k+1} \log\log n)$. The term occ becomes $occ \cdot k^3 3^k$ if edit distance is in concern.

[2] For example, consider $k = 2$, the index requires $O(n \log^2 n)$ words, which means tens of gigabytes of memory for a text of a few million characters. Indexing a human chromosome or genome (typically a few hundred million to a few billion characters) is not feasible.

Table 1. Known results for k-error matching. Results given in this paper are marked with †. c and ϵ are positive constants.

Space	$k = 1$	
$O(n \log^2 n)$ words	$O(m \log n \log\log n + occ)$	[1]
$O(n \log n)$ words	$O(m \log\log n + occ)$	[2]
	$O(m + occ + \log n \log\log n)$	[5]
$O(n)$ words	$O(\min\{n, m^2\} + occ)$	[4]
	$O(m \log n + occ)$	[8]
	$O(m \log\log n + occ)$	[9]
	$O(m + occ + \log^3 n \log\log n)$	†
$O(n)$ bits	$O(m \log^2 n + occ \log n)$	[8]
	$O((m \log\log n + occ) \log^\epsilon n)$	[9]
	$O((m + occ + \log^4 n \log\log n) \log^\epsilon n)$	†

Space	$k \geq 2$	
$O(n \log^k n)$ words	$O(m + occ + \frac{1}{k!}(c \log n)^k \log\log n)$	[5]
$O(n)$ words	$O(\min\{n, m^{k+1}\} + occ)$	[4]
	$O((cm)^k \log n + occ)$	[8]
	$O((cm)^k \log\log n + occ)$	[9]
	$O(m + occ + (c \log n)^{k(k+1)} \log\log n)$	†
$O(n)$ bits	$O((cm)^k \log^2 n + occ \log n)$	[8]
	$O(((cm)^k \log\log n + occ) \log^\epsilon n)$	[9]
	$O((m + occ + (c \log n)^{k(k+2)} \log\log n) \log^\epsilon n)$	†

This index also admits a simple tradeoff between space and time. I.e., the matching can be speeded up if more space is used. Roughly speaking, for any $h \leq k$, if $O(n \log^{k-h+1} n)$ space is used, then a k-error query can be answered in $O(m + occ + c^{k^2} \log^{\max\{kh, k+h\}} n \log\log n)$ time. For example, choosing $h = 3$ gives an $O(n \log^{k-2} n)$-word index with matching time $O(m + occ + c^k \log^{3k} n \log\log n)$.

(ii) The $O(n)$-word index can be compressed to occupy $O(n)$ bits only, with k-error matching time increasing to $O((m + occ + (c \log n)^{k(k+2)} \log\log n) \log^\epsilon n)$, where $\epsilon < 1$. In particular, when $k = 1$, the $O(n)$-bit index achieves matching in $O((m + occ + \log^4 n \log\log n) \log^\epsilon n)$ time.

Other related results. Note that the above results concern worst-case performance. The literature also contains several interesting results on average-case performance (see, e.g., [13, 10, 3]).

2 An $O(n)$-Word Index for k-Error Matching

This section considers Hamming distance only and presents an $O(n)$-word index for a text $S[1..n]$. Given any pattern $P[1..m]$, the index finds all substrings of S matching P within k errors, in $O(m + occ + \text{polylog } n)$ time. We call these substrings the k-error matches of P.

The index handles long patterns and short patterns separately. Intuitively, short patterns can be handled easily. For example, a pattern of length $\log n$ can be handled in polylog n time even with the naive $\Omega(m^k)$ time methods. The main novelty of our index is a check-point technique for handling long patterns: we define some locations of S to be *check-points*. Special indexing are done for suffixes and prefixes of S terminating at these check-points. For long patterns, their k-error matches in S will certainly contain some check-points, so the special indexing at the check-points suffices for finding the matches efficiently.

We now describe how to handle long patterns. Consider a text $S[1..n]$. Let β be a positive integer, which will be fixed later to $k3^k \log^{k+1} n$. Intuitively, a pattern is long if its length is at least β. For each $a = \beta, 2\beta, 3\beta, \ldots$, we call $S[a]$ a check-point.

Observation 1. *Let $P[1..m]$ be a pattern with $m \geq \beta$. For any k-error match $S[j_1..j_2]$ of P, there exists an integer a, $j_1 \leq a \leq j_2$ such that $S[a]$ is a check-point and $0 \leq a - j_1 \leq \beta - 1$.*

Furthermore, let $i = a - j_1 + 1$. There exist integers $k_1, k_2 \geq 0$, such that (1) $S[a..n]$ has a prefix matching $P[i..m]$ with k_1 errors, (2) $S[1..a-1]$ has a suffix matching $P[1..i-1]$ with k_2 errors, and (3) $k_1 + k_2 \leq k$.

Let TAIL be the set of suffixes of S beginning at a check-point, i.e., TAIL $= \{S[a..n] \mid a = \beta, 2\beta, \ldots\}$. Similarly, let HEAD be the set of prefixes of S ending just before a check-point, i.e., HEAD $= \{S[1..a-1] \mid a = \beta, 2\beta, \ldots\}$. Observation 1 suggests finding the k-error matches of P as follows.

Algorithm 1. k-MATCH(P): finds all k-error matches of P in S, for $|P| \geq \beta$.

```
For each $i = 1, \ldots, \beta$, cut $P$ into $P[1..i-1]$ and $P[i..m]$. Try all possible
$k_1, k_2 \geq 0$ such that $k_1 + k_2 \leq k$, and perform the following.
Step 1. Find all $S[a..n] \in$ TAIL that have a prefix matching $P[i..m]$ with exactly
    $k_1$ errors. Let tail$_{i,k_1}$ be the set of these suffixes.
Step 2. Find all $S[1..b] \in$ HEAD that have a suffix matching $P[1..i-1]$ with
    exactly $k_2$ errors. Let head$_{i,k_2}$ be the set of these prefixes.
Step 3. For each $S[a..n] \in$ tail$_{i,k_1}$ and $S[1..b] \in$ head$_{i,k_2}$, we call them a connecting pair
    if $a = b + 1$. For each connecting pair, we report a $k$-error match
    of $P$ starting at $S[a - i + 1]$.
```

We first prove the correctness of the algorithm. Details of the implementation are given in the coming subsections.

Lemma 1. *Let $P[1..m]$ be a pattern with $m \geq \beta$. k-MATCH(S, P) finds all k-error matches of P in S.*

Proof. For each k-error match $S[j_1..j_2]$ of P, Observation 1 states that there is a check-point $S[a]$ contained in $S[j_1..j_2]$ and $0 \leq a - j_1 \leq \beta - 1$.

Consider aligning $P[1..m]$ with $S[j_1..j_2]$. The suffix $S[a..n]$ has a prefix matching $P[i'..m]$ with k'_1 errors, where $i' = a - j_1 + 1$ and k'_1 is some integer between

0 and k. Thus, $S[a..n]$ will be included in $\text{tail}_{i',k_1'}$. Similarly, $S[1..a-1]$ will be included in $\text{head}_{i',k_2'}$, where k_2' is some integer between 0 and k, and $k_1' + k_2' \leq k$. They form a connecting pair, so $S[j_1..j_2]$ will be reported.

There are only n/β suffixes and prefixes in TAIL and HEAD, respectively, so we can build more complicated data structures to support the above steps efficiently, while maintaining a small space requirement. In the following subsections, we present the actual data structures. Then, we will give analysis for the total space and time complexity of the index.

2.1 Indexes for Finding Tail_{i,k_1} and Head_{i,k_2}

We want to find tail_{i,k_1} efficiently for any pattern $P[1..m]$, $i = 1, \ldots, \beta$ and $k_1 = 0, \ldots, k$. We do it by storing an ℓ-error-tree [5] for TAIL, for each $\ell = 0, \ldots, k$. The performance guarantee provided by an ℓ-error tree is stated in the following lemma.

Lemma 2. [5] Let Z be any collection of suffixes of a text $S[1..n]$. For any integer $\ell \geq 0$, an ℓ-error-tree for Z has the following properties.

1. The ℓ-error tree is a collection of trees with totally $O(|Z| 3^\ell \log^\ell n)$ nodes. Each leaf represents a suffix in Z and at most $O(3^\ell \log^\ell n)$ leaves represent the same suffix.
2. The ℓ-error tree takes $O(|Z| 3^\ell \log^\ell n)$-word space.
3. For any pattern $Q[1..m']$, there exist $O(6^\ell \log^\ell n)$ nodes in the ℓ-error-tree, such that each leaf under the nodes represents a distinct suffix in Z that has a prefix matching Q with exactly ℓ errors. It takes $O(6^\ell \log^\ell n \log\log n)$ time to find these nodes, after preprocessing all suffixes of Q with the suffix tree of S in totally $O(m')$ time.

For each $\ell = 0, 1, \ldots, k$, We store an ℓ-error tree for TAIL, calling them T-error-tree$_0$, T-error-tree$_1$, …, T-error-tree$_k$. Furthermore, we store a suffix tree for S.

For any i and k_1, the above lemma implies that there exist $O(6^{k_1} \log^{k_1} n)$ nodes in T-error-tree$_{k_1}$ such that the leaves under them represent the distinct suffixes in tail_{i,k_1}. We called these nodes the *covering nodes* for tail_{i,k_1}. For time efficiency, we will not find tail_{i,k_1} explicitly, instead we only find the covering nodes to represent tail_{i,k_1} implicitly. Using the error-tree data structures, we have the following performance on finding the covering nodes.

Lemma 3. *We can build an $O(n + n/\beta \times 3^k \log^k n)$-word data structure for TAIL. For any pattern $P[1..m]$, we preprocess P in $O(m)$ time. Then, for any $i = 1, \ldots, \beta$ and $k_1 = 0, \ldots, k$, we can find $O(6^{k_1} \log^{k_1} n)$ covering nodes for tail_{i,k_1} in T-error-tree$_{k_1}$ in $O(6^{k_1} \log^{k_1} n \log\log n)$ time.*

Proof. The suffix tree of S takes $O(n)$ words and T-error-tree$_0$, T-error-tree$_1$, …, T-error-tree$_k$ take totally $\sum_{\ell=0}^{k} O(n/\beta \times 3^\ell \log^\ell n) = O(n/\beta \times 3^k \log^k n)$ words.

Given any $P[1..m]$, we preprocess all suffixes of P with the suffix tree of S in totally $O(m)$ time. It implies preprocessing all suffixes of $P[i..m]$ with the suffix tree. Thus, finding the covering nodes for tail_{i,k_1} can be done in $O(6^{k_1} \log^{k_1} n \log\log n)$ time using T-error-tree$_{k_1}$.

Note that there can be more than one set of covering nodes for tail_{i,k_1}, and any set of covering nodes is sufficient for our algorithm to find the k-error matches of P.

The case for finding head_{i,k_2} is symmetric. For each $\ell = 0, 1, \ldots, k$, we store an ℓ-error-tree for HEAD, calling them H-error-tree$_0$, …, H-error-tree$_k$. We also store the suffix tree for the reverse of S. Finding covering nodes for head_{i,k_2}, for any i and k_2 takes $O(6^{k_2} \log^{k_2} n \log\log n)$ time, after an $O(m)$ time preprocessing of P with the suffix tree for the reverse of S.

2.2 Indexes for Finding Connecting Pairs

Consider certain i, k_1 and k_2 where $k_1 + k_2 \leq k$. Assume that tail_{i,k_1} is found implicitly, represented by a set of covering nodes U in T-error-tree$_{k_1}$. Similarly, assume that head_{i,k_2} is represented by a set of covering nodes W in H-error-tree$_{k_2}$. To find the k-error matches of P, we want to find all suffixes $S[a..n] \in \text{tail}_{i,k_1}$ and prefixes $S[1..b] \in \text{head}_{i,k_2}$ that are connecting pairs, i.e., $a = b + 1$.

We observe that this can be done as follows. We preprocess T-error-tree$_{k_1}$ with H-error-tree$_{k_2}$. For each leaf in T-error-tree$_{k_1}$ representing a suffix $S[a..n]$ and for each leaf in H-error-tree$_{k_2}$ representing a prefix $S[1..b]$, we draw an imaginary edge between them if $a = b + 1$. Then, to find the connecting pairs between tail_{i,k_1} and head_{i,k_2}, we try each pair of $u \in U$ and $w \in W$ and perform the following EdgeReport(u, w) query: Given $u \in U$ and $w \in W$, find all leaf pairs (x, y) such that x and y are descendents of u and w, respectively, and x, y are connected by an imaginary edge.

While T-error-tree$_{k_1}$ is a collection of trees, we can always convert it into a single tree by linking all trees to a new root. Similarly, we convert H-error-tree$_{k_2}$ into a single tree. Then, we store a tree-cross-product data structure [2] for T-error-tree$_{k_1}$ and H-error-tree$_{k_2}$ to support the EdgeReport(u, w) query efficiently, which has the following performance.

Lemma 4. *[2] Let $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ be two trees. Let $V = V_1 \cup V_2$ and let $I \subseteq V_1 \times V_2$ be a set of imaginary edges connecting some nodes in V_1 and V_2. We can build an $O(|I| \log |V|)$-word index for T_1 and T_2 such that for any $u \in V_1$ and $w \in V_2$, the EdgeReport(u, w) query takes $O(\log\log |V| + occ')$ time, where occ' is the number of imaginary edges reported.*

For each pair of error-trees T-error-tree$_{k_1}$ and H-error-tree$_{k_2}$, where $k_1 + k_2 \leq k$, we create the imaginary edges and build the tree-cross-product data structure. It allows us to find the connecting pairs efficiently. We assume that tail_{i,k_1} and head_{i,k_2} are represented by $O(6^{k_1} \log^{k_1} n)$ and $O(6^{k_2} \log^{k_2} n)$ covering nodes in the corresponding error-trees, respectively, which is the case during the execution of the Algorithm k-MATCH.

Lemma 5. *We can store an* $O(k \times n/\beta \times 3^k \log^{k+1} n)$*-word data structure for the error-trees. Then, for any* i*,* k_1 *and* k_2 *with* $k_1+k_2 \leq k$*, we can find all connecting pairs between* $tail_{i,k_1}$ *and* $head_{i,k_2}$ *in* $O(6^{k_1+k_2} \log^{k_1+k_2} n \log\log n + occ')$ *time, where* occ' *is the number of connecting pairs found.*

Proof. Consider T-error-tree$_{k_1}$ and H-error-tree$_{k_2}$, where $k_1+k_2 = c$ for some $c \leq k$. There are $O(n/\beta \times 3^{k_1} \log^{k_1} n)$ leaves in T-error-tree$_{k_1}$. For each leaf representing a suffix $S[a..n]$, the prefix $S[1..a-1]$ is represented by at most $O(3^{k_2} \log^{k_2} n)$ leaves in H-error-tree$_{k_2}$. So, the number of imaginary edges between the two error-trees is $O(n/\beta \times 3^{k_1+k_2} \log^{k_1+k_2} n)$, and the tree-cross-product data structure takes $O(n/\beta \times 3^c \log^{c+1} n)$ words. For any c, there are at most $k+1$ pairs of possible (k_1, k_2), and we store tree-cross product data structures for $c = 0, 1, \ldots, k$, so the total space needed is $\sum_{c=0}^{k} O(k \times n/\beta \times 3^c \log^{c+1} n) = O(k \times n/\beta \times 3^k \log^{k+1} n)$ words.

For any tail$_{i,k_1}$ and head$_{i,k_2}$, where $k_1 + k_2 \leq k$, let U and W be the corresponding set of covering nodes. Finding the connecting pairs is done by performing an EdgeReport(u, v) query for each $u \in U$ and $w \in W$. There are $O(6^{k_1} \log^{k_1} n \times 6^{k_2} \log^{k_2} n)$ queries, and the total query time is $O(6^{k_1+k_2} \times \log^{k_1+k_2} n \log\log n + occ')$ time.

2.3 Total Time and Space Complexity

With Lemma 3 and 5, we can analyse the space and time complexity of our data structure.

Theorem 1. *We can build an* $O(n + k \times n/\beta \times 3^k \log^{k+1} n)$*-word index for* $S[1..n]$*. For any pattern* $P[1..m]$*,* $m \geq \beta$*, we can find all* k*-error matches of* P *in* S *in* $O(m + occ + \beta k 6^k \log^k n \log\log n)$ *time, where* occ *is the number of matches.*

Proof. We only need to store the data structures specified in Lemma 3 and 5, so the total space is $O(n + k \times n/\beta \times 3^k \log^{k+1} n)$ words.

To find the k-error matches of P, we perform an $O(m)$ time preprocessing of P, as required by Lemma 3. Then, we iterate for $i = 1, 2, \ldots, \beta$ and $c = 0, 1, \ldots, k$. For each i and c, there are at most $k+1$ pairs of $k_1, k_2 \geq 0$ such that $k_1 + k_2 = c$. Finding the covering nodes for tail$_{i,k_1}$ and head$_{i,k_2}$ takes $O(6^{k_1} \log^{k_1} n \log\log n + 6^{k_2} \log^{k_2} n \log\log n)$. Finding the connecting pairs between tail$_{i,k_1}$ and head$_{i,k_2}$ takes $O(6^{k_1+k_2} \log^{k_1+k_2} n \log\log n + occ')$ time, where occ' is the number of connecting pairs found. Thus, for any fixed i and c, the runtime is $O(k \times 6^c \log^c n \log\log n + occ')$ time.

We try i from 1 to β and c from 0 to k, so the total time complexity is $O(m + \beta \times \sum_{c=0}^{k} k \times 6^c \log^c n \log\log n + occ)$ $O(m + \beta k 6^k \log^k n \log\log n + occ)$.

By putting $\beta = k 3^k \log^{k+1} n$, we obtain an $O(n)$-word index for handling long patterns. For short patterns, we can use the $O(n)$-word data structure of Lam et al. [9] which find the k-error matches of a pattern $P[1..m]$ in $O(|\Sigma|^k m^k \log\log n + occ)$ time, where $|\Sigma|$ is the size of the alphabet.

Corollary 1. *For any constant k, we can build an $O(n)$-word index for $S[1..n]$. For any pattern $P[1..m]$, finding the k-error matches of P in S takes $O(m + occ + (c \log n)^{\max\{k(k+1), 2k+1\}} \log\log n)$ time.*

Proof. We put $\beta = k3^k \log^{k+1} n$ to Theorem 1 to obtain an $O(n)$-word index. We also store the $O(n)$-word data structure of Lam et al. [9].

For pattern of length at least $k3^k \log^{k+1} n$, finding the k-error matches takes $O(m + occ + k^2 18^k \log^{2k+1} n \log\log n)$ time. For pattern of length less than $k3^k \log^{k+1} n$, finding the k-error matches takes $O(occ + |\Sigma|^k m^k \log\log n) = O(occ + |\Sigma|^k k^k 3^{k^2} \log^{k(k+1)} n \log\log n)$ time.

Reducing the polylog n term in matching time. The polylog n term in the matching time is biggest for patterns with length slightly less than $k3^k \log^{k+1} n$, in which we use the brute-force method to obtain a runtime of $O(occ + |\Sigma|^k k^k 3^{k^2} \times \log^{k^2+k} n \log\log n)$. We can reduce the polylog n term by a small trick. To ease the discussion, we remove the constant factors $|\Sigma|$ and k from the asymptotic analysis.

We improve the matching time for patterns of length between $O(\log^k n)$ and $O(\log^{k+1} n)$ by choosing a smaller value of β. In particular, we choose β to be $O(\log^k n)$, but we only build an data structure for finding $(k-1)$-error matches. By Theorem 1, the index takes only $O(n)$ words. To find the k-error matches, we explicitly try different positions on the pattern and modify that position with a different character. Then, we search for $(k-1)$-error matches for each of the modified patterns, which will be the k-error matches of the pattern. This gives a runtime of $O(m \times (m + \beta \log^{k-1} n \log\log n + occ)) = O(\log^{2k+2} n + \log^{3k} n \log\log n + occ \times \log^{k+1} n)$. The multiplicative term for occ can be removed by careful book-keeping to avoid reporting the same occurrence for multiple times. It reduces the matching time from $O(occ + \log^{k^2+k} n \log\log n)$ to $O(occ + \max\{\log^{2k+2} n, \log^{3k} n \log\log n\})$, for patterns of length $O(\log^k n)$ to $O(\log^{k+1} n)$.

We can continue to apply this technique for other range of pattern length, and it can reduce the polylog n term in the matching time to $\log^{k^2/2+O(1)k} n$ in the worst case.

3 Tradeoff Between Space and Time

Our data structure allows a tradeoff between space and time. We notice that the value β controls the number of check-points in S, which is equivalent to the number of suffixes of S on which special indexes are built. Choosing a smaller β generates more check-points and increases the index size, but it allows patterns of shorter length to be handled and reduces the matching time. On the other hand, choosing a bigger β reduces the number of check-points such that we can even obtain an $O(n)$-bit data structure for k-error matching, at the cost of increasing the matching time. This section presents the results for this tradeoff.

3.1 Improved Searching with More Space

We choose $\beta = k3^k \log^h n$, where h is any integer, $0 \leq h \leq k$. Note that a smaller h generates more check-points and bigger index size. By Theorem 1, it gives an $O(n \log^{k-h+1} n)$-word index, which finds the k-error matches of $P[1..m]$, $m \geq k3^k \log^h n$, in $O(m + occ + k^2 18^k \log^{h+k} n \log\log n)$ time.

For patterns of length less than $k3^k \log^h n$, we use the $O(n)$-word data structure of Lam et al. [9], which gives a matching time of $O(|\Sigma|^k k^k 3^{k^2} \log^{hk} n \log\log n + occ)$.

Theorem 2. *For any constant h and k such that $0 \leq h \leq k$, we can build an index for $S[1..n]$ using $O(n \log^{k-h+1} n)$ space. For any pattern $P[1..m]$, we can find all k-error matches of P in S in $O(m+occ+c^{k^2}(\log n)^{\max\{hk,h+k\}} \log\log n)$ time where occ is the number of occurrences found and c is some constant.*

3.2 Reducing to $O(n)$-Bit Space

We can choose $\beta = k3^k \log^{k+2} n$. Then, the error-trees and the tree-cross-product data structures takes $O(n)$-bit space. We can replace the suffix tree of S by a compressed suffix tree [14], which supports each of the suffix tree operations in $O(\log^\epsilon n)$ time. Thus, the preprocessing of P takes $O(m \log^\epsilon n)$ time. The matching time for pattern of length at least $k3^k \log^{k+2} n$ is $O(m \log^\epsilon n + occ + k^2 18^k \log^{2k+2} n \log\log n)$.

For patterns of length less than $k3^k \log^{k+2} n$, we use the $O(n)$-bit data structure of [9], which gives a matching time of $O((|\Sigma|^k k^k 3^{k^2} \log^{k^2+2k} n \log\log n + occ) \log^\epsilon n)$.

Theorem 3. *For any constant k, we can build an index for $S[1..n]$ using $O(n)$-bit space. For any pattern $P[1..m]$, we can find all k-error matches of P in S in $O((m + occ + (c \log n)^{\max\{k^2+2k,2k+2\}} \log\log n) \log^\epsilon n)$ time where occ is the number of occurrences reported, c is some constant, and $\epsilon > 0$.*

4 k-Error Matching in Edit Distance

This section considers edit distance, and an error is an insertion, deletion or substitution. We give an $O(n)$-word data structure for $S[1..n]$ which supports finding the k-error matches of P in S. Precisely, given $P[1..m]$, it finds all starting positions j such that $S[j..n]$ has a prefix matching P with at most k errors, in $O(m + k^3 3^k occ + \text{polylog } n)$ time, where occ is the number of starting positions found.

Similar to the case of Hamming distance, we handle long patterns by the check-point technique, while short patterns are handled by simple brute force methods. We define $S[a]$ to be a check-point for $a = \beta, 2\beta, \ldots$, where β will be set later to $k5^k \log^{k+1} n$.

Observation 2. *Let $P[1..m]$ be a pattern with $m \geq \beta + k$. For any k-error match $S[j_1..j_2]$ of P, there exists an integer a, $j_1 \leq a \leq j_2$ such that $S[a]$ is a check-point and $0 \leq a - j_1 \leq \beta - 1$.*

Furthermore, there exist integers i, $1 \leq i \leq \beta + k$ and $k_1, k_2 \geq 0$, such that (1) $S[a..n]$ has a prefix matching $P[i..m]$ with k_1 errors, (2) $S[1..a-1]$ has a suffix matching $P[1..i-1]$ with k_2 errors, and (3) $k_1 + k_2 \leq k$.

Define HEAD and TAIL as before. Observation 2 suggests the following algorithm.

Algorithm 2. k-EDIT(P), find starting positions of k-error matches of P in S, $|P| \geq \beta + k$.

```
For each $i = 1, \ldots, \beta + k$, cut $P$ into $P[1..i-1]$ and $P[i..m]$. Try all possible
  $k_1, k_2 \geq 0$ such that $k_1 + k_2 \leq k$, and perform the following.
Step 1. Find all $S[a..n] \in$ TAIL that have a prefix matching $P[i..m]$ with exactly
    $k_1$ errors. Let tail$_{i,k_1}$ be the set of these suffixes.
Step 2. Find all $S[1..b] \in$ HEAD that have a suffix matching $P[1..i-1]$ with
    exactly $k_2$ errors. Let head$_{i,k_2}$ be the set of these prefixes.
Step 3. For each $S[a..n] \in$ tail$_{i,k_1}$ and $S[1..b] \in$ head$_{i,k_2}$, we call them a connecting pair
    if $a = b + 1$. For each connecting pair, we find all $j_1$ such that
    $S[j_1..a-1]$ matches $P[1..i-1]$ with exactly $k_2$ errors, and we report each $j_1$
    as an answer.
```

To find tail$_{i,k_1}$ and head$_{i,k_2}$ efficiently for different i, k_1 and k_2, we store another type of error-trees by Cole et al. [5] for TAIL and HEAD, which work for edit distance. We call them edit-trees to avoid confusion. Basically, an edit-tree is similar to an error-trees, which is also built for a collection Z of suffixes of S. Given a pattern $Q[1..m']$, an ℓ-edit tree returns the nodes such that the leaves under the nodes represent all suffixes in Z that has a prefix matching Q with exactly ℓ errors (edit distance). However, an edit-tree may give duplicated answers, i.e., there may be different leaves under these nodes representing the same suffix in Z.

We build T-edit-tree$_0$, ..., T-edit-tree$_k$ for TAIL and H-edit-tree$_0$, ..., H-edit-tree$_k$ for HEAD. We also store the suffix trees for S and the reverse of S. Finally, we build the tree-cross-product data structures for the pair T-edit-tree$_{k_1}$ and H-edit-tree$_{k_2}$, for every k_1, k_2. These data structures can support the Algorithm k-EDIT efficiently.

We can analyse the space and time complexity of the data structures similar to that in Section 2 and we obtain the following theorem. There is a $k^3 3^k$ factor for *occ* because when we find tail$_{i,k_1}$ for some i, k_1, the edit-trees may return the same suffix for multiple times, leading to duplication in the output.

Theorem 4. *We can build an $O(n + k \times n/\beta \times 5^k \log^{k+1} n)$-word index for $S[1..n]$. For any pattern $P[1..m]$, $m \geq \beta$, we can find all j such that $S[j..n]$ has a prefix matching P with at most k errors (in edit distance), in $O(m + k^3 3^k occ + \beta k 6^k \log^k n \log\log n)$ time, where occ is the number of answers found.*

By putting $\beta = k5^k \log^{k+1} n$, and handling short patterns by Lam et al. [9] we obtain an $O(n)$-word index which finds the k-error matches in $O(m + k^3 3^k occ +$ polylog $n)$ time.

References

1. A. Amir, D. Keselman, G. M. Landau, M. Lewenstein, N. Lewenstein, and M. Rodeh. Indexing and dictionary matching with one error. In *Proceedings of Workshop on Algorithms and Data Structures*, 1999, pages 181–192.
2. A. L. Buchsbaum, M. T. Goodrich, and J. R. Westbrook. Range searching over tree cross products. In *Proceedings of European Symposium on Algorithms*, 2000, pages 120–131.
3. E. Chavez and G. Navarro. A metric index for approximate string matching. In *Proceedings of Latin American Theoretical Informatics*, 2002, pages 181-195.
4. A. Cobbs. Fast approximate matching using suffix trees. In *Proceedings of Symposium on Combinatorial Pattern Matching*, 1995, pages 41–54.
5. R. Cole, L. A. Gottlieb, and M. Lewenstein. Dictionary matching and indexing with errors and don't cares. In *Proceedings of Symposium on Theory of Computing*, 2004, pages 91–100.
6. P. Ferragina and G. Manzini. Opportunistic Data Structures with Applications. In *Proceedings of Symposium on Foundations of Computer Science*, pages 390–398, 2000.
7. R. Grossi and J. S. Vitter. Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching. In *Proceedings of Symposium on Theory of Computing*, pages 397–406, 2000.
8. T. N. D. Huynh, W. K. Hon, T. W. Lam, and W. K. Sung. Approximate string matching using compressed suffix arrays. In *Proceedings of Symposium on Combinatorial Pattern Matching*, 2004, pages 434–444.
9. T. W. Lam, W. K. Sung, S. S. Wong. Improved approximate string matching using compressed suffix data structures. In *Proceedings of International Symposium on Algorithms and Computation*, 2005, pages 339–348.
10. M. G. Maaß and J. Nowak. Text indexing with errors. Technical Report TUM-10503, Fakultät für Informatik, TU München, Mar. 2005.
11. U. Manber and G. Myers. Suffix Arrays: A New Method for On-Line String Searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.
12. E. M. McCreight. A Space-economical Suffix Tree Construction Algorithm. *Journal of the ACM*, 23(2):262–272, 1976.
13. G. Navarro and R. Baeza-Yates. A Hybrid Indexing Method for Approximate String Matching. *J. Discrete Algorithms*, 1(1):205-209, 2000. Special issue on Matching Patterns.
14. K. Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*, accepted.
15. P. Weiner. Linear Pattern Matching Algorithms. In *Proceedings of Symposium on Switching and Automata Theory*, 1973, pages 1–11.

On-Line Linear-Time Construction of Word Suffix Trees

Shunsuke Inenaga[1,2] and Masayuki Takeda[2,3]

[1] Japan Society for the Promotion of Science
[2] Department of Informatics, Kyushu University, Fukuoka 812-8581, Japan
{shunsuke.inenaga, takeda}@i.kyushu-u.ac.jp
[3] SORST, Japan Science and Technology Agency (JST)

Abstract. Suffix trees are the key data structure for text string matching, and are used in wide application areas such as bioinformatics and data compression. Sparse suffix trees are kind of suffix trees that represent only a subset of suffixes of the input string. In this paper we study *word suffix trees*, which are one variation of sparse suffix trees. Let D be a dictionary of words and w be a string in D^+, namely, w is a sequence $w_1 \cdots w_k$ of k words in D. The word suffix tree of w w.r.t. D is a path-compressed trie that represents only the k suffixes in the form of $w_i \cdots w_k$. A typical example of its application is word- and phrase-level search on natural language documents. Andersson et al. proposed an algorithm to build word suffix trees in $O(n)$ expected time with $O(k)$ space. In this paper we present a new word suffix tree construction algorithm with $O(n)$ running time and $O(k)$ space in the worst cases. Our algorithm is on-line, which means that it can sequentially process the characters in the input, each by each, from left to right.

1 Introduction

Suffix trees have played a very central role in combinatorial pattern matching as they enable us to solve a multitude of important problems efficiently [3, 8]. To give some examples of applications, suffix trees are utilized in data compression [13, 16, 10] and in bioinformatics such as motif finding [14], regulatory elements discovery [5], and fast protein classification [7]. Suffix trees are fairly useful since they can be constructed in linear time and space with respect to the input string length [19, 15, 18].

On the other hand, there have been great demands to deal with a common case where only certain suffixes of the input string are relevant. Suffix trees that contain only a subset of all suffixes are called *sparse suffix trees*.

The 'sparsity' of the suffix tree varies with the application: In [12] Kärkkäinen and Ukkonen proposed the *evenly spaced sparse suffix tree* which contains every i-th suffix for some fixed positive integer i. Their contribution is an algorithm which allows the original full text to be searched, by using the evenly spaced sparse suffix tree. Clifford and Sergot [6] introduced *distributed suffix trees* whose idea is to partition the original suffix tree into a constant number of subtrees and construct each of them in linear time, in parallel. Their suffix tree is thus

M. Lewenstein and G. Valiente (Eds.): CPM 2006, LNCS 4009, pp. 60–71, 2006.

helpful to index huge genome sequence databases. Also, sparse suffix trees for a set of arbitrary suffixes are used in the core of pattern discovery algorithms from biological sequences [11, 9].

Another type of sparse suffix trees is *word suffix trees* [4]. Let D be a dictionary of words and w be a string in D^+, namely, w is a sequence $w_1 \cdots w_k$ of k words in D. The word suffix tree of w w.r.t. D is a tree structure which represents only the k suffixes in the form of $w_i \cdots w_k$. One typical application of word suffix trees is a word- and phrase-level index for documents written in a natural language. Note that normal suffix trees report *any* occurrences of a keyword in the text string, which may cause unwanted matchings (e.g., an occurrence of "other" in "mother" is possibly retrieved).

This paper investigates word suffix tree construction. The most intuitive solution is to build a normal suffix tree using $O(n)$ time and space, then to prune it so that only the leaves corresponding to the k suffixes remain. However, this approach apparently wastes extra space, as the size of the resulting tree is only $O(k)$. To index large text strings efficiently, we need to handle a restricted situation where only $O(k)$ computational space is available. Still, this is a rather challenging task, as traditional linear suffix tree construction algorithms heavily rely on the fact that *all* suffixes are to be inserted in the tree. On the other hand, it is no more true for word suffix trees.

In [2] Andersson et al. took a first step in this problem: they presented an algorithm to build word suffix trees with $O(k)$ working space in $O(n)$ *expected* running time. This present paper takes a further step and puts a period to this problem - our algorithm constructs word suffix trees with $O(k)$ working space in $O(n)$ running time *in the worst cases*. Remark that this is optimal, since the resulting tree requires $O(k)$ space, and we have to read the whole input string at least once and it takes $O(n)$ time. Our algorithm is based on, and is a generalization of, Ukkonen's on-line suffix tree construction algorithm introduced in [18]. In addition, our algorithm can be seen as a practical solution to efficient construction of general sparse suffix trees.

The rest of the paper is organized as follows. In Section 2 we introduce some definitions and notations. In Section 3 we define word suffix tries and propose an on-line construction algorithm for them. Section 4 presents a word suffix tree construction algorithm, which is the main subject of this paper. Finally, conclusions and further discussions are given in Section 5.

2 Preliminaries

Let Σ be a finite set of symbols, called an *alphabet*. A finite sequence of symbols is called a *string*. We denote the length of a string u by $|u|$. The empty string is denoted by ε, that is, $|\varepsilon| = 0$. Let Σ^* be the set of strings over Σ, and let $\Sigma^+ = \Sigma^* \backslash \{\varepsilon\}$. Strings x, y, and z are said to be a *prefix*, *substring*, and *suffix* of the string $u = xyz$, respectively. A prefix, substring, and suffix of a string u are said to be *proper* if they are not u. Let $\mathit{Prefix}(u)$ and $\mathit{Suffix}(u)$ be the set of prefixes and suffixes of string u, respectively. Let $\mathit{Prefix}(S) = \bigcup_{u \in S} \mathit{Prefix}(u)$ for a set S of strings. The i-th symbol of a string u is denoted by $u[i]$ for $1 \leq i \leq |u|$, and the substring of a string u that begins at position i and ends at position j is denoted by $u[i..j]$ for $1 \leq i \leq j \leq |u|$.

Definition 1 (Prefix property). *A set L of strings is said to have the* prefix property *if no string in L is a proper prefix of another string in L.*

Let $D = \Sigma^* \cdot \#$. Then D is a set of strings each followed by #, and D is called a *dictionary.* We assume that any string w is an element of D^+. This is a very natural assumption, since for example in the European languages the blank character can be regarded as the special character #, and any text is an element of D^+.

A *factorization* of string $w \in D^+$ w.r.t. D is a list $w_1, \ldots, w_k$ of strings in D such that $w = w_1 \cdots w_k$. Note that this factorization is always unique, since $D = \Sigma^* \cdot \#$ clearly satisfies the prefix property because of # not being in Σ. Now, let $\mathit{Suffix}_D(w) = \{w_i \cdots w_k \mid 1 \leq i \leq k+1\}$. Remark that $\mathit{Suffix}_D(w)$ is a subset of $\mathit{Suffix}(w)$ which consists only of the original string w and the suffixes which immediately follow # in w (including the empty suffix ε intended by $w_{k+1}w_k$).

3 Word Suffix Trie

In this section, we present our word suffix *trie* construction algorithm which will be a basis of our word suffix *tree* construction algorithm to be given later as the main topic of this paper.

3.1 Definition

Definition 2 (Word suffix trie). *The* word suffix trie *of a string $w \in D^+$ w.r.t. D, denoted by $\mathit{WSTrie}_D(w)$, is a trie which represents $\mathit{Suffix}_D(w)$.*

Fig. 1 compares the normal suffix trie and the word suffix trie for string w, where $\Sigma = \{\mathtt{a}, \mathtt{b}\}$, $D = \Sigma^* \cdot \#$, and $w = \mathtt{ab\#ab\#a\#}$.

It is easy to see that there is a natural one-to-one correspondence between the nodes of $\mathit{WSTrie}_D(w)$ and the strings in $\mathit{Prefix}(\mathit{Suffix}_D(w))$. Any string u in $\mathit{Prefix}(\mathit{Suffix}_D(w))$ can be written as $u = xy$ such that $x \in D^*$ and y is a *proper* prefix of some string in D. It should be stated that the choice of x and y is unique for each u. Hereafter, we represent a node of $\mathit{WSTrie}_D(w)$ with an ordered pair $\langle x, y \rangle$, as mentioned above.

3.2 Word Suffix Trie Construction Algorithm

Suffix Link. Ukkonen [18] used suffix links for on-line construction of normal suffix tries. Here we give a new definition of suffix links that is suitable for on-line word suffix trie construction.

For dictionary $D = \Sigma^* \cdot \#$, we consider the smallest DFA M_D which accepts D. Clearly it has a unique final state with no outgoing edges (see the left of

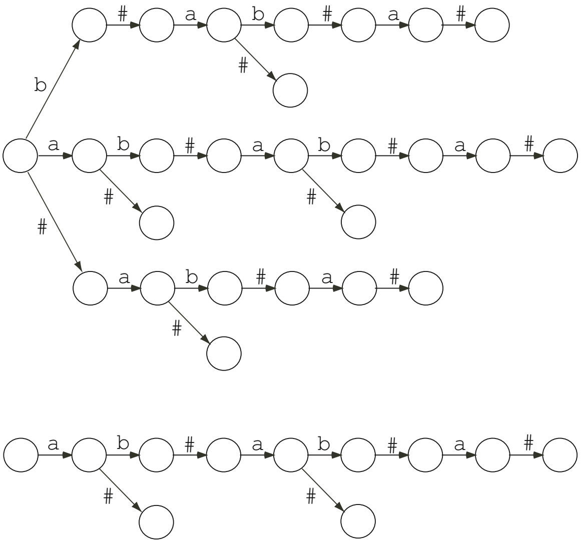


Fig. 1. The normal suffix trie of $w = \mathtt{ab\#ab\#a\#}$ on the upper, and the word suffix trie of w w.r.t. $D = \{\mathtt{a}, \mathtt{b}\} \ \cdot \#$ on the lower. Note that the normal suffix tree represents all the suffixes of w, while the word suffix tree represents only the suffixes $\mathtt{ab\#ab\#a\#}, \mathtt{ab\#a\#}, \mathtt{a\#}, \varepsilon \in \mathit{Suffix}_D(w)$.

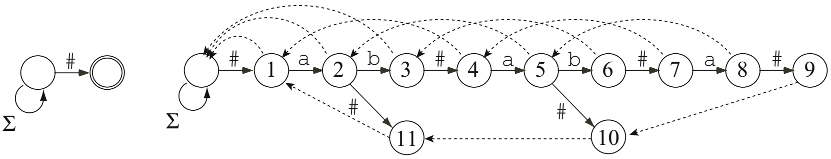


Fig. 2. To the left is the smallest DFA M_D accepting $D = \{\mathtt{a}, \mathtt{b}\} \ \cdot \#$, and to the right is $\mathit{WSTrie}_D(w)$ for $w = \mathtt{ab\#ab\#a\#}$, with M_D and its suffix links (broken arrows) attached. Nodes 4, 5, 6, 7, 8, 9, 10, and 11 are those in Group 1 of Definition 3, and nodes 1, 2, and 3 are those in Group 2.

Fig. 2). Then we attach M_D to the word suffix trie, replacing the unique final state of M_D by the root of the word suffix trie. Now we define the suffix links of word suffix tries as follows:

Definition 3 (Suffix links of word suffix trie). *Let $D = \Sigma^* \cdot \#$ and M_D be the smallest DFA that accepts D. For each node $s = \langle x, y \rangle$ of $\mathit{WSTrie}_D(w)$,*

1. *if $x \in D^+$, the suffix link from s goes to node $\langle x', y \rangle$ such that $x' \in D^*$ and $x = hx'$ for some $h \in D$;*
2. *otherwise (if $x = \varepsilon$), the suffix link from s goes to the initial state of M_D.*

Fig. 2 shows the smallest DFA M_D which accepts $D = \{\mathtt{a}, \mathtt{b}\}^* \cdot \#$, and $\mathit{WSTrie}_D(w)$ for $w = \mathtt{ab\#ab\#a\#}$ with its suffix links.

Algorithm. Fig. 3 shows a pseudo code of our on-line algorithm to build word suffix tries, with the help of DFA M_D and suffix links of Definition 3. Observe that procedure *Update* is identical to that of Ukkonen's on-line normal suffix trie construction algorithm of [18]. The only change is the initialization steps of the main routine where we set the root of the trie to the final state of M_D and the suffix link of the root to the initial state of M_D. This simple modifications make a difference in the resulting data structures. A snapshot of on-line construction of $\mathit{WSTrie}_D(w)$ with the running example is shown in Fig. 4.

```
Input:    $w = w[1..n] \in D^+$ and auxiliary DFA $M_D$.
Output:   Word suffix trie of $w$ w.r.t. $D$.
{
    $root$ = the final state of $M_D$;
    $slink(root)$ = the initial state of $M_D$;
    $top = root$;
    for ($i = 1$; $i \leq n$; $i++$) $top = Update(top, w[i])$;
}

node $Update(top, c)$ {
    $newtop = CreateNewNode()$;
    create a new edge $top \xrightarrow{c} newtop$;
    $prev = newtop$;
    for ($t = slink(top)$; no $c$-edge from $t$; $t = slink(t)$) {
        $new = CreateNewNode()$;
        create a new edge $t \xrightarrow{c} new$;
        $slink(prev) = new$;
        $prev = new$;
    }
    $slink(prev)$ = the initial node of the $c$-edge from $t$;
    return $newtop$;
}
```

Fig. 3. Word suffix trie construction algorithm. For any node v, $slink(v)$ represents the node to which the suffix link of v goes. Remark that function *Update* is identical to that of Ukkonen's normal suffix trie construction algorithm [18]. The initialization step using the auxiliary DFA M_D changes the algorithm so that it builds word suffix tries.

For the correctness of the algorithm of Fig. 3, it suffices to show the following lemma:

Lemma 1. *Let $w \in D^+$, $w_1, \ldots, w_k$ be a unique factorization of w w.r.t. D. Let j be an integer with $0 \leq j \leq |w|$, and u be the prefix of length j of w. Let $u = w_1 \ \cdots w_\ell v$ such that v is a proper prefix of $w_{\ell+1}$. After the j-th call of the Update operation, we have a trie representing the strings*

$$\{w_i \cdots w_\ell \mid 1 \leq i \leq \ell + 1\} \cdot v.$$

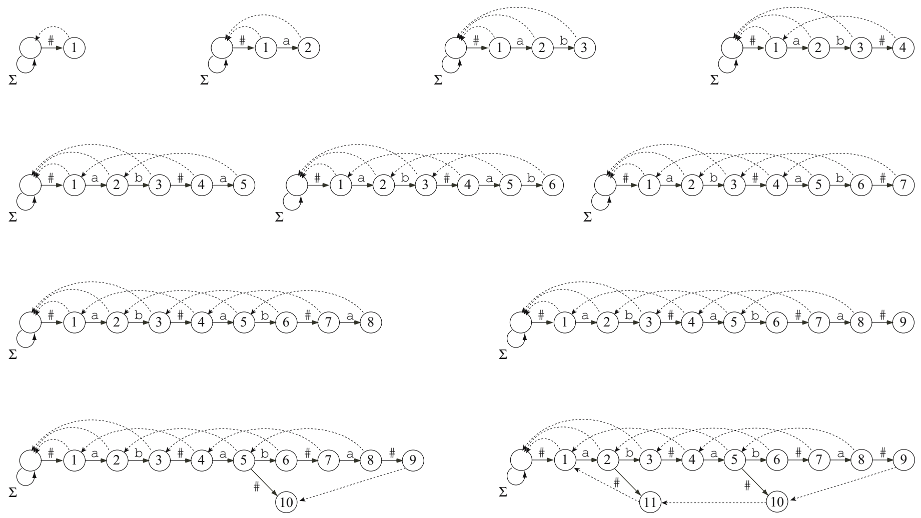


Fig. 4. A snapshot of on-line construction of $\mathit{WSTrie}_D(w)$ with $w = \mathtt{ab\#ab\#a\#}$ and $D = \{\mathtt{a}, \mathtt{b}\} \cdot \#$. The update with the last # is shown in three steps, where we get three new nodes and edges.

The suffix link of the node $\langle w_i \cdots w_\ell, v\rangle$ goes to the node $\langle w_{i+1} \cdots w_\ell, v\rangle$, if $i \le \ell$; and otherwise, goes to the state $\delta(q_0, v)$ of M_D, where δ and q_0 are, respectively, the state-transition function and the initial state of M_D.

Proof. By induction on $j = |u|$. When $|u| = 0$, the lemma trivially holds. We now consider $|u| > 0$. When $v \ne \varepsilon$, let $v = v'b$ with $v' \in \Sigma^*$ and $b \in \Sigma$. By the induction hypothesis, after the $(j-1)$-th call of *Update*, we have a trie representing

$$\{w_i \cdots w_\ell \mid 1 \le i \le \ell + 1\} \cdot v',$$

and the suffix link of node $\langle w_i \cdots w_\ell, v'\rangle$ goes to node $\langle w_{i+1} \cdots w_\ell, v'\rangle$, if $i \le \ell$; and otherwise, goes to the state $\delta(q_0, v')$ of M_D. At the j-th call, the variable *top* is set to the node $\langle w_1 \cdots w_\ell, v'\rangle$ and the node $\langle w_1 \cdots w_\ell, v'b\rangle$ is created (variable *newtop*). In the iteration of the **for** loop, we traverse the suffix links starting at the node $\langle w_1 \cdots w_\ell, v'\rangle$. For each $i = 2, \ldots, \ell$, the node $\langle w_i \cdots w_\ell, v'b\rangle$ is created, if it does not exist. Note that the iteration is guaranteed to halt since the suffix links lead us to the state $\delta(q_0, v')$. During the iteration, the suffix links of the newly created nodes $\langle w_i \cdots w_\ell, v'b\rangle$ are set to the nodes $\langle w_{i+1} \cdots w_\ell, v'b\rangle$, if $i \le \ell$; and to the state $\delta(q_0, v'b)$, otherwise. Thus the lemma holds for the case $v \ne \varepsilon$. Similarly, we can prove the case $v = \varepsilon$. □

Remark 1. Our word suffix trie construction algorithm of Fig. 3 generalizes Ukkonen's normal suffix trie construction algorithm [18]. Assume just for now $D = \Sigma$, and consider a DFA which accepts Σ with only two states that are a single initial state and a single final state. Then this DFA plays the same role as the auxiliary '⊥' node used in Ukkonen's algorithm, and thus our algorithm builds normal suffix tries. The same discussion applies to the word suffix tree construction algorithm to be given in the next section.

4 Word Suffix Tree

In the previous section, we presented our on-line algorithm that constructs word suffix tries. The drawback is, however, that the size of a word suffix trie can be quadratic in the input string length. In this section, we consider the *word suffix tree* whose size is bounded by $O(k)$, where k is the number of words in string w w.r.t. dictionary D. We then propose a new algorithm to build a word suffix tree in $O(n)$ time with $O(k)$ space, where $n = |w|$ and $w = w_1 \cdots w_k$. The advantage of our algorithm to the one by Andersson et al. [2] is that our algorithm runs in $O(n)$ time *in the worst cases*, while their algorithm runs in $O(n)$ time *on the average*.

4.1 Definitions

Definition 4 (Word suffix tree). *The* word suffix tree *of a string $w \in D^+$ w.r.t. D, denoted by $\mathit{WSTree}_D(w)$, is a path-compressed trie which represents $\mathit{Suffix}_D(w)$.*

For any strings x, y, let $\mathit{lcp}(x, y)$ denote the longest common prefix of x and y. Let

$$I = \{\mathit{lcp}(w_i \cdots w_k, w_j \cdots w_k) \mid 1 \leq i \neq j \leq k+1\} \text{ and,}$$
$$E = \{w_i \cdots w_k \mid w_i \cdots w_k \notin \mathit{Prefix}(w_j \cdots w_k) \text{ for any } 1 \leq j < i \leq k\}.$$

Then, there is a one-to-one correspondence between the strings in I and the internal nodes (including the root) of $\mathit{WSTree}_D(w)$, and there is a one-to-one correspondence between the strings in E and the leaves of $\mathit{WSTree}_D(w)$. Hereafter, we sometimes refer to any node s of $\mathit{WSTree}_D(w)$ as the corresponding string in $I \cup E$.

Fig. 5 compares the normal suffix tree and the word suffix tree for string $w = \mathtt{ab\#ab\#a\#}$, where $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ and $D = \Sigma^* \cdot \#$.

4.2 Word Suffix Tree Construction Algorithm

Note that $|I| + |E| = O(k)$, which means that the size of $\mathit{WSTree}_D(w)$ is also $O(k)$. Since $\mathit{WSTree}_D(w)$ is path compressed, the edges of $\mathit{WSTree}_D(w)$ are labeled by substrings of w rather than single characters. By implementing these substring labels with pointers to w, $\mathit{WSTree}_D(w)$ can be finally implemented in $O(k)$ space. The time cost for word suffix tree construction is $\Omega(n)$ due to the need of scanning the whole string w. Thus, the final goal is to construct $\mathit{WSTree}_D(w)$ in $O(n)$ time with $O(k)$ space.

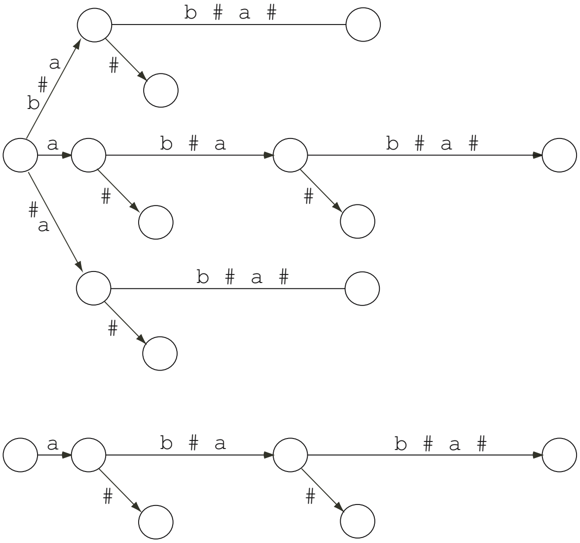


Fig. 5. The normal suffix tree of $w = \mathtt{ab\#ab\#a\#}$ on the upper, and the word suffix tree of w w.r.t. $D = \{\mathtt{a}, \mathtt{b}\}\ \cdot \#$ on the lower.

Suffix Link. The suffix links of $\mathit{WSTree}_D(w)$ are a key to achieve the above goal. Recall that any node s of $\mathit{WSTrie}_D(w)$ is regarded as a unique ordered pair $\langle x, y\rangle$, such that $x \in D^*$ and y is a proper prefix of some string in D. We apply the same notion to the nodes of $\mathit{WSTree}_D(w)$. Also, we use the auxiliary DFA M_D that accepts D in the same way.

Definition 5 (Suffix links of word suffix tree). *Let $D = \Sigma^* \cdot \#$ and M_D be the smallest DFA that accepts D. For each node $s = \langle x, y\rangle$ of $\mathit{WSTree}_D(w)$,*

1. *if $s \in I$ and $x \in D^+$, the suffix link from s goes to node $\langle x', y\rangle$ such that $x' \in D^*$ and $x = hx'$ for some $h \in D$;*
2. *if $s \in I$ and $x = \varepsilon$, the suffix link from s goes to the initial state of M_D;*
3. *otherwise (if $s \in E$), the suffix link from s is undefined.*

The suffix links of those in Group 3 in the above definition remain undefined, as they are never used in our construction algorithm to be shown later. See Fig. 6

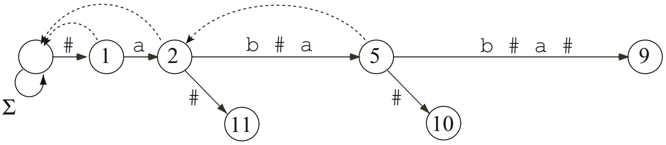


Fig. 6. The word suffix tree with auxiliary DFA M_D and suffix links (broken arrows), where $w = \mathtt{ab\#ab\#a\#}$ and $D = \{\mathtt{a}, \mathtt{b}\}\ \cdot \#$. Note that the suffix links of nodes 9, 10, and 11, which are those in Group 3 of Definition 5, are missing, as they are never used in the construction algorithm.

```
Input:    $w = w[1..n] \in D^+$ and auxiliary DFA $M_D$.
Output:   Word suffix tree of $w[1..n]$ w.r.t. $D$.
{
    $root$ = the final state of $M_D$;  $slink(root)$ = the initial state of $M_D$;
    $(s, k) = (root, 1)$;
    for $(i = 1; i \leq n; i++)$ {
        $oldr$ = nil;
        while ($CheckEndPoint(s, (k, i-1), w[i])$ == false) {
            if $(k \leq i-1)$    $r = SplitEdge(s, (k, i-1))$;
            else    $r = s$;
            $t = CreateNewNode()$;
            create a new edge $r \xrightarrow{(i,\ \ )} t$;
            if ($oldr \neq$ nil)    $slink(oldr) = r$;
            $oldr = r$;
            $(s, k) = Canonize(slink(s), (k, i-1))$;
        }
        if ($oldr \neq$ nil)    $slink(oldr) = s$;
        $(s, k) = Canonize(s, (k, i))$;
    }
}

boolean $CheckEndPoint(s, (k, p), c)$ {
    if $(k \leq p)$ {    /* $(s, (k, p))$ is implicit. */
        let $s \xrightarrow{(k',p')} s$  be the $w[k]$-edge from $s$;
        return $(c == w[k\ \ + p - k + 1])$;
    } else return (there is a $c$-edge from $s$);
}

(node,integer)-pair $Canonize(s, (k, p))$ {
    if $(k > p)$    return $(s, k)$;    /* $(s, (k, p))$ is explicit. */
    find the $w[k]$-edge $s \xrightarrow{(k',p')} s$  from $s$;
    while $(p\ \ - k\ \ \leq p - k)$ {
        $k$+= $p\ \ - k\ \ + 1$; $s = s$ ;
        if $(k \leq p)$    find the $w[k]$-edge $s \xrightarrow{(k',p')} s$  from $s$;
    }
    return $(s, k)$;
}

node $SplitEdge(s, (k, p))$ {
    let $s \xrightarrow{(k',p')} s$  be the $w[k]$-edge from $s$;
    $r = CreateNewNode()$;
    replace this edge by edges $s \xrightarrow{(k',k'+p\ \ \ k)} r$ and $r \xrightarrow{(k'+p\ \ \ k+1,p')} s$ ;
    return $r$;
}
```

Fig. 7. Word suffix tree construction algorithm

for the word suffix tree with the auxiliary DFA M_D and suffix links, using the running example.

Algorithm. A pseudo-code of our algorithm to build word suffix trees is summarized in Fig. 7. It simulates construction of word suffix tries in $O(n)$ time and with $O(k)$ space. Fig. 8 shows a snapshot of on-line construction of $\mathit{WSTree}_D(w)$ with the running example.

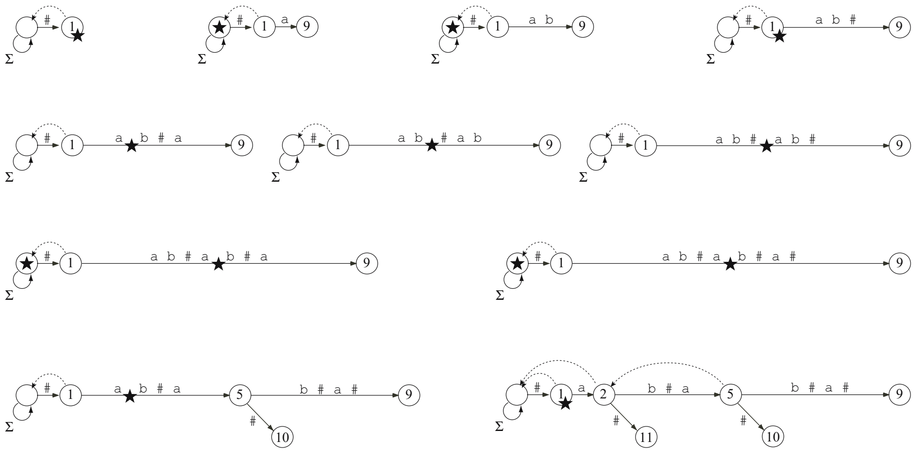


Fig. 8. A snapshot of on-line construction of $\mathit{WSTree}_D(w)$ with $w = \mathtt{ab\#ab\#a\#}$ and $D = \{\mathtt{a}, \mathtt{b}\} \ \cdot \#$. The update with the last # is shown in three steps. The star mark denotes the location represented by $(s, (k, i-1))$ in the algorithm of Fig. 7, from which a new edge is possibly created.

The main result of this paper follows:

Theorem 1. *The algorithm of Fig. 7 builds word suffix trees in linear time (on a fixed alphabet).*

Proof. Since the algorithm is a time and space economical simulation of the word suffix trie construction algorithm of Fig. 3, the correctness follows from Lemma 1.

We now prove the linearity of the algorithm. Consider the location, referred to as $(s, (k, i-1))$, which represents the substring $w[k-\ell..i-1]$ where ℓ is the length of the string represented by the node s. One iteration of the **while** loop in the main routine alters s into $\mathit{slink}(s)$ and therefore decreases the length of the substring by at least one. We note that *Canonize* never alters the substring represented by $(s, (k, p))$ although it might update s and k. On the other hand, the length of the substring is increased by at most one at each iteration of the **for** loop in the main routine. Thus, the total number of iterations of the **while** loop in the main routine is linearly proportional to the input string length. We have only to estimate the total cost of all executions of *Canonize*. We note that the value of variable k changes only by an execution of *Canonize*, and monotonically increases. The cost of one execution of *Canonize* is proportional to the number of iterations of the **while** loop in it plus one, which is linear with respect to the number of times the variable k is increased during the iterations. The total cost of all executions of *Canonize* is therefore proportional to the number of times k is increased in the execution of the algorithm. Since the length of the string $w[k..i-1]$ is increased by at most one at each iteration of the **for** loop in the main routine, the number of times k is increased is linear with respect to the input string length. □

5 Conclusions and Further Discussions

We have presented a new on-line algorithm for constructing word suffix trees. The algorithm is very simple and runs in linear time *even in the worst cases*, whereas the one proposed by Anderson et al. runs in linear time *on the average*.

The simplicity of our algorithm is due to the use of DFA M_D accepting a dictionary D. The idea comes from the synchronization technique introduced in [17] in which similar DFA are embedded onto the Aho-Corasick pattern matching machines [1] so that they process multi-byte character texts in a byte-by-byte manner without extra work for avoiding false matches.

Lastly, our algorithm can be seen as a practical solution to efficient construction of general sparse suffix trees. Let $w \in \Sigma^*$ and Pos be a set of positions of suffixes we want to store in the sparse suffix tree. Let $|w| = n$ and $|Pos| = k$. For any position i in Pos, we insert the special character # at position $i-1$ of the original string w. Note that the length of the modified string w' is at most twice as that of the original string w, and therefore the word suffix tree for w' can be constructed with $O(k)$ space in $O(n)$ time. To search for pattern $p \in \Sigma^*$ of length m, we skip any # in the word suffix tree of w'. This way the matching can be done correctly, and in $O(m)$ time.

References

1. A. V. Aho and M. Corasick. Efficient string matching: An aid to bibliographic search. *Comm. ACM*, 18(6):333–340, 1975.
2. A. Andersson, N. J. Larsson, and K. Swanson. Suffix trees on words. *Algorithmica*, 23(3):246–260, 1999.
3. A. Apostolico. The myriad virtues of subword trees. *Combinatorial Algorithms on Words*, F12:85–96, 1985.
4. R. Baeza-Yates and G. H. Gonnet. Efficient text searching of regular expressions. In *Proc. 16th International Colloquium on Automata, Languages and Programming (ICALP'89)*, volume 372 of *Lecture Notes in Computer Science*, pages 46–62. Springer-Verlag, 1989.
5. H. Bannai, S. Inenaga, A. Shinohara, M. Takeda, and S. Miyano. Efficiently finding regulatory elements using correlation with gene expression. *Journal of Bioinformatics and Computational Biology*, 2(2):273–288, 2004.

6. R. Clifford and M. Sergot. Distributed and paged suffix trees for large genetic databases. In *Proc. 14th Ann. Symp. on Combinatorial Pattern Matching (CPM'03)*, volume 2676 of *Lecture Notes in Computer Science*, pages 70–82. Springer-Verlag, 2003.
7. B. Dorohonceanu and C. G. Nevill-Manning. Accelerating protein classification using suffix trees. In *Proc. 8th International Conference on Intelligent Systems for Molecular Biology (ISMB'00)*, pages 128–133. AAAI Press, 2000.
8. D. Gusfield. *Algorithms on Strings, Trees, and Sequences.* Cambridge University Press, 1997.
9. S. Inenaga, H. Bannai, H. Hyyrö, A. Shinohara, M. Takeda, K. Nakai, and S. Miyano. Finding optimal pairs of cooperative and competing patterns with bounded distance. In *Proc. 7th International Conference on Discovery Science (DS'04)*, volume 3245 of *Lecture Notes in Artificial Intelligence*, pages 32–46. Springer-Verlag, 2004.
10. S. Inenaga, T. Funamoto, M. Takeda, and A. Shinohara. Linear-time off-line text compression by longest-first substitution. In *Proc. 10th International Symp. on String Processing and Information Retrieval (SPIRE'03)*, volume 2857 of *Lecture Notes in Computer Science*, pages 137–152. Springer-Verlag, 2003.
11. S. Inenaga, T. Kivioja, and V. Mäkinen. Finding missing patterns. In *Proc. 4th Workshop on Algorithms in Bioinformatics (WABI'04)*, volume 3240 of *Lecture Notes in Bioinformatics*, pages 463–474. Springer-Verlag, 2004.
12. J. Kärkkänen and E. Ukkonen. Sparse suffix trees. In *Proc. 2nd International Computing and Combinatorics Conference (COCOON'96)*, volume 1090 of *Lecture Notes in Computer Science*, pages 219–230. Springer-Verlag, 1996.
13. N. J. Larsson. Extended application of suffix trees to data compression. In *Proc. Data Compression Conference '96 (DCC'96)*, pages 190–199. IEEE Computer Society, 1996.
14. L. Marsan and M.-F. Sagot. Extracting structured motifs using a suffix tree - algorithms and application to promoter consensus identification. In *Proc. 4th Annual International Conference on Computational Molecular Biology (RECOMB'00)*, pages 210–219. ACM, 2000.
15. E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of ACM*, 23(2):262–272, 1976.
16. J. C. Na, A. Apostolico, C. S. Iliopoulos, and K. Park. Truncated suffix trees and their application to data compression. *Theoretical Computer Science*, 304(1–3):87–101, 2003.
17. M. Takeda, S. Miyamoto, T. Kida, A. Shinohara, S. Fukamachi, T. Shinohara, and S. Arikawa. Processing text files as is: Pattern matching over compressed texts, multi-byte character texts, and semi-structured texts. In *Proc. 9th International Symp. on String Processing and Information Retrieval (SPIRE'02)*, volume 2476 of *Lecture Notes in Computer Science*, pages 170–186. Springer-Verlag, 2002.
18. E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.
19. P. Weiner. Linear pattern-matching algorithms. In *Proc. of 14th IEEE Ann. Symp. on Switching and Automata Theory*, pages 1–11, 1973.

Obtaining Provably Good Performance from Suffix Trees in Secondary Storage[⋆]

Pang Ko[1] and Srinivas Aluru[2]

[1] Department of Electrical and Computer Engineering
[2] Laurence H. Baker Center for Bioinformatics and Biological Statistics
Iowa State University
{kopang, aluru}@iastate.edu

Abstract. Designing external memory data structures for string databases is of significant recent interest due to the proliferation of biological sequence data. The suffix tree is an important indexing structure that provides optimal algorithms for memory bound data. However, string B-trees provide the best known asymptotic performance in external memory for substring search and update operations. Work on external memory variants of suffix trees has largely focused on constructing suffix trees in external memory or layout schemes for suffix trees that preserve link locality. In this paper, we present a new suffix tree layout scheme for secondary storage and present construction, substring search, insertion and deletion algorithms that are competitive with the string B-tree. For a set of strings of total length n, a pattern p and disk blocks of size B, we provide a substring search algorithm that uses $O(|p|/B + \log_B n)$ disk accesses. We present algorithms for insertion and deletion of all suffixes of a string of length m that take $O(m \log_B(n + m))$ and $O(m \log_B n)$ disk accesses, respectively. Our results demonstrate that suffix trees can be directly used as efficient secondary storage data structures for string and sequence data.

1 Introduction

The suffix tree data structure is widely used in text processing, information retrieval, and computational biology. It is especially useful when there are no word or sentence structures, such as in biological sequences, for which the suffix tree is uniquely suited for indexing and querying. With the continued explosion in the size of biological sequence databases, there is growing interest in string indexing schemes in general, and disk-based suffix trees in particular.

The suffix tree of a set of strings is a compacted trie of all suffixes of all the strings. Since the introduction of this data structure by Weiner [15], several linear time algorithms for in-memory construction of suffix trees have been designed: notable ones include McCreight's linear space algorithm [11], Ukkonen's on-line algorithm [13], and Farach's algorithm for integer alphabets [4]. To extend the scale of data that can be handled in-memory, Grossi and Vitter [9] developed

[⋆] Research supported by the National Science Foundation under IIS-0430853.

M. Lewenstein and G. Valiente (Eds.): CPM 2006, LNCS 4009, pp. 72–83, 2006.

compressed suffix trees and suffix arrays. In the last few years, there has been significant research on disk-based storage of suffix trees for exploiting their utility on ever growing sequence databases. Many algorithms and strategies have been proposed to reduce the number of disk accesses during the construction of suffix trees [1, 2, 5, 10, 12]. Of these, only Farach *et al.* provided a construction algorithm for secondary storage that achieves the optimal worst case bound of $\Theta\left(\frac{n}{B}\log_{\frac{M}{B}}\frac{n}{B}\right)$ disk accesses (where M is the size of main memory).

While these algorithms and techniques focused on suffix tree construction in secondary storage, the problems of searching and updating suffix trees (insertion/deletion of all suffixes of a string) on disks have not received as much attention. An interesting solution is provided by Clark and Munro [3] that achieves efficient space utilization. This is used to obtain a bound on disk accesses for substring search as a function of the height of the tree. In the worst case, the height of a suffix tree can be proportional to the length of the text indexed, although it is rarely the case and Clark and Munro's approach provides good experimental performance. To date, algorithms with provably good worst-case performance for substring searches and updates for suffix trees in secondary storage are not known. To overcome these theoretical limitations, Ferragina and Grossi have proposed the string B-tree data structure [6, 7]. String B-trees provide the best known bounds for the worst-case number of disk access required for queries and updates. It is not known if the same performance bounds can be achieved with suffix trees. The unbalanced nature of suffix trees appears to be a major obstacle to designing efficient disk-based algorithms.

In this paper, we propose a new suffix tree layout scheme, and present algorithms with provably good worst-case bounds on disk accesses required for search and update operations, while maintaining our layout. Let n denote the number of leaves in the suffix tree, and B denote the size of a disk block. We provide algorithms that

- search for a pattern p in $O(|p|/B + \log_B n)$ disk accesses,
- insert (all suffixes of) a string of length m in $O(m \log_B(n+m))$ disk accesses, and
- delete (all suffixes of) a string of length m in $O(m \log_B n)$ disk accesses.

Since suffix tree construction can be achieved by starting from an empty tree and inserting strings one after another, the number of disk accesses needed for suffix tree construction is $O(n \log_B n)$. Our results provide the same worst-case performance as string B-trees, thus showing that suffix trees can be stored on disk and searched as efficiently as string B-trees.

The rest of the paper is organized as follows: In Section 2 we present our layout scheme. The scheme partitions the suffix tree such that the number of partitions encountered on any root to leaf path is bounded by $\log_B n$. This is a crucial feature that helps in overcoming problems caused by the unbalanced nature of suffix trees. Section 3 contains our algorithm for substring search using the proposed layout. Algorithms for inserting and deleting a new string are discussed in Section 4. Section 5 contains further discussion and Section 6 concludes the paper.

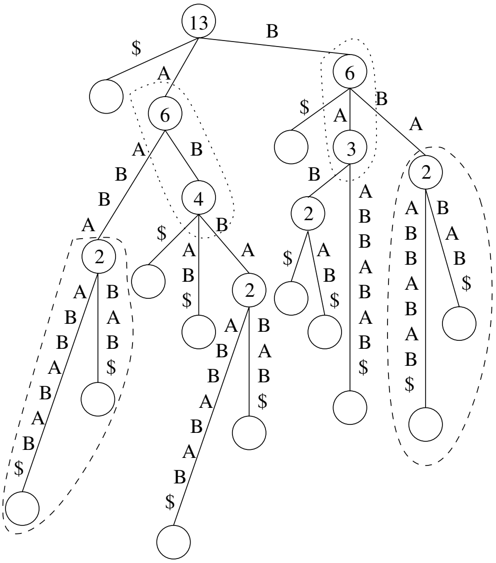


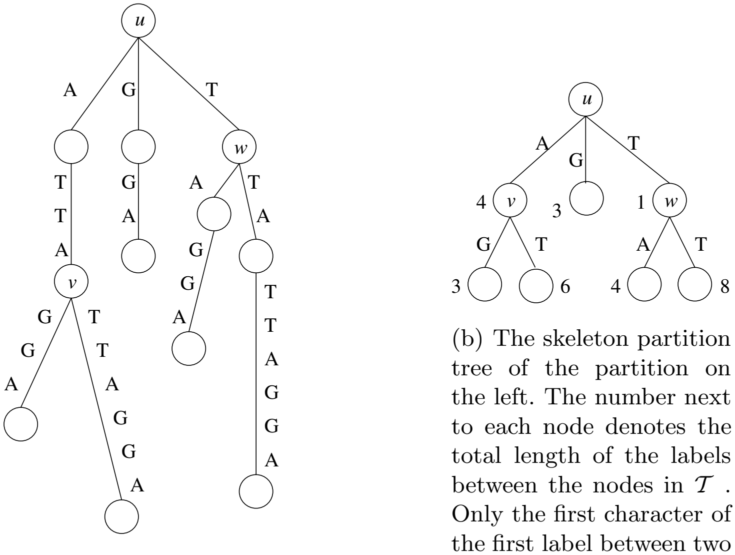
Fig. 1. The suffix tree of the string AABBAABBABAB$. The number in an internal node is the number of leaves in its subtree. A partitioning is shown with $C = 3$. Two example partitions of *rank* zero are circled with dashed lines, and two of *rank* one with dotted lines.

2 Suffix Tree Disk Layout

Consider a set S of strings of total length n and a fixed size alphabet Σ. Without loss of generality, we assume that the last character of each string is a special character $\$ \notin \Sigma$, and the remaining characters are drawn from Σ. Let $s \in S$ be a string of length m. We use $s[i]$ to denote the i-th character of s. Let $s[i..j]$ denote the substring $s[i]s[i+1] \ldots s[j]$. The i-th suffix of s, $s[i..m]$, is denoted by s_i. The suffix tree of the set of strings S, abbreviated ST, is a compacted trie of all suffixes of all strings in S. Another commonly used notation is to use the term *generalized suffix tree* when dealing with multiple strings and reserve the term *suffix tree* when dealing with a single string. For convenience, we use the term *suffix tree* to denote either case.

For a node v in ST, the string depth of v is the total length of all edge labels on the path from the root to v. The number of leaves in the subtree under v is referred to as $size(v)$. If v is a leaf node then $size(v) = 1$. The *rank* of node v, denoted $rank(v)$, is i if and only if $C^i \leq size(v) < C^{i+1}$, for some integer constant C of choice. Nodes u and v belong to the same partition if all nodes on the undirected path between u and v have the same *rank*. It is easy to see that the entire suffix tree is partitioned into disjoint parts. Figure 1 shows an example of a suffix tree, and some of its partitions.

The rank of a partition $\mathcal{P}$ is the same as the rank of the nodes in $\mathcal{P}$, i.e. $rank(\mathcal{P}) = rank(v)$ for any v in $\mathcal{P}$. Node v in $\mathcal{P}$ is a leaf in $\mathcal{P}$ if and only if none



(a) An example partition with all character of the edge labels shown. Nodes u, v, w are branching nodes.

(b) The skeleton partition tree of the partition on the left. The number next to each node denotes the total length of the labels between the nodes in $\mathcal{T}$. Only the first character of the first label between two nodes is used in the labels.

Fig. 2. Illustration of partitions and skeleton partition trees

of v's children in ST belong to $\mathcal{P}$. Node u is termed the root of $\mathcal{P}$ if and only if u's parent is not a node in $\mathcal{P}$. Figure 2(a) shows an example of a partition.

Lemma 1. *There are at most* $C - 1$ *leaves in a partition.*

Proof. Let $\mathcal{P}$ be a partition that has $C' \geq C$ leaves, and node u be its root. Since $size(u) \geq C^i \cdot C' \geq C^i \cdot C = C^{i+1}$, $rank(u) > rank(\mathcal{P})$, a contradiction. □

Node v in $\mathcal{P}$ is a branching node if two or more of its children are in $\mathcal{P}$. All other nodes are referred to as non-branching nodes. From Figure 2(a) we see that a partition $\mathcal{P}$ need not be a compacted trie. For each partition $\mathcal{P}$, a compacted trie is constructed containing the root node of $\mathcal{P}$, all branching nodes and all the leaves. Furthermore, only the first character of each edge label is stored. This resulting compacted trie is referred to as the skeleton partition tree of $\mathcal{P}$, or $\mathcal{T}_{\mathcal{P}}$. Figure 2(b) shows the skeleton partition tree of the partition in Figure 2(a).

Lemma 2. *For a partition* $\mathcal{P}$*, the number of nodes in* $\mathcal{T}_{\mathcal{P}}$ *is at most* $2C - 2$*.*

Proof. By Lemma 1 there are at most $C - 1$ leaf nodes in a skeleton partition tree. Therefore, there can be at most $C - 2$ branching nodes. In addition, the root node may or may not be a branching node. So the total number of nodes in a skeleton partition tree is at most $2C - 2 = O(C)$. □

While Lemma 2 shows that the size of $\mathcal{T}_{\mathcal{P}}$ is bounded by $O(C)$, it gives no bound on the size of $\mathcal{P}$. The worst case number of nodes in partition $\mathcal{P}$ of rank i is $C^{i+1} - C^i$, corresponding to a chain of $C^{i+1} - C^i$ nodes in ST where the bottom node has a subtree with C^i leaves and all other nodes have an additional leaf child each. Note that $\mathcal{T}_{\mathcal{P}}$ in this case has only two nodes, the top and bottom nodes of the chain. So $\mathcal{T}_{\mathcal{P}}$ can be viewed as an additional data structure built on top of $\mathcal{P}$ in order to traverse $\mathcal{P}$ effectively. A node u in ST appearing in a partition $\mathcal{P}$ is described as $u_{\mathcal{P}}$; similarly, its appearance in $\mathcal{T}_{\mathcal{P}}$ is described as $u_{\mathcal{T}}$. The information stored in $u_{\mathcal{T}}$ and $u_{\mathcal{P}}$ are different, because $u_{\mathcal{T}}$'s function is to help navigate $\mathcal{T}_{\mathcal{P}}$ to locate a part of $\mathcal{P}$, while $u_{\mathcal{P}}$ is used to navigate from one partition to another in ST. Any disk block can contain either skeleton partition trees, part of a partition, or some of the input strings, but not a mixture of them.

Let $u_{\mathcal{T}}$ and $v_{\mathcal{T}}$ be nodes in $\mathcal{T}_{\mathcal{P}}$ such that $u_{\mathcal{T}}$ is the parent of $v_{\mathcal{T}}$. All the non-branching nodes between $u_{\mathcal{P}}$ and $v_{\mathcal{P}}$ (including $u_{\mathcal{P}}$, $v_{\mathcal{P}}$) form a linked list. Since $u_{\mathcal{P}}$ is a branching node, it will be the head of multiple such linked lists. Also, $u_{\mathcal{P}}$ will be the tail of another linked list. For storage efficiency, we require each node in $\mathcal{P}$ to be stored in exactly one linked list. So $u_{\mathcal{P}}$ is not stored as the head (first node) of the linked list from $u_{\mathcal{P}}$ to $v_{\mathcal{P}}$, but rather as the tail (last node) in the linked list that ends at $u_{\mathcal{P}}$. Note if $u_{\mathcal{P}}$ is the root of a partition, it is stored by itself. Since the linked list between $u_{\mathcal{P}}$ and $v_{\mathcal{P}}$ does not contain $u_{\mathcal{P}}$ we refer to this linked list as $\mathcal{LL}(u_{\mathcal{P}}, v_{\mathcal{P}}]$.

To summarize, our layout scheme first divides the suffix tree into partitions. All the nodes in a partition are further divided into linked lists. The skeleton partition tree allows us to find any of the linked lists in a partition efficiently. All links in our data structure are bidirectional for navigation. We will now describe the augmenting information needed to efficiently perform the search operation.

The function of a skeleton partition tree $\mathcal{T}_{\mathcal{P}}$ is to allow the identification of a $\mathcal{LL}(u_{\mathcal{P}}, v_{\mathcal{P}}]$ in $\mathcal{P}$, such that one of the nodes in $\mathcal{LL}(u_{\mathcal{P}}, v_{\mathcal{P}}]$ has a child pointer to the next partition we need to load for our search. Let $v_{\mathcal{T}}$ be a child of $u_{\mathcal{T}}$ in $\mathcal{T}_{\mathcal{P}}$.

- Store in $v_{\mathcal{T}}$ the first character of the first edge label on the path from $u_{\mathcal{T}}$ to $v_{\mathcal{T}}$ in ST, and refer to this character as $first\_char(v_{\mathcal{T}})$.
- Store in $v_{\mathcal{T}}$ a pointer $ptr\_\mathcal{LL}(v_{\mathcal{T}})$ to the tail (last node) of $\mathcal{LL}(u_{\mathcal{P}}, v_{\mathcal{P}}]$.
- Store in $u_{\mathcal{T}}$ a pointer $ptr\_\mathcal{LL}(u_v)$ to the head (first node) of $\mathcal{LL}(u_{\mathcal{P}}, v_{\mathcal{P}}]$.
- Store in $u_{\mathcal{T}}$ the string depth of $u_{\mathcal{T}}$ in ST, denoted as $string\_depth(u_{\mathcal{T}})$.
- Store in $u_{\mathcal{T}}$ $(s, rep\_suff(u_{\mathcal{T}}))$, such that $s_{rep\_suff(u_{\mathcal{T}})}$ is a suffix represented by one of the leaves in ST under $u_{\mathcal{T}}$.

For each node $u_{\mathcal{P}}$ we store the following information:

- String depth of $u_{\mathcal{P}}$, also denoted as $string\_depth(u_{\mathcal{P}})$.
- For each child w of u in the suffix tree, store in $u_{\mathcal{P}}$ a pointer to $w_{\mathcal{T}}$. Note that this pointer is stored irrespective of if w is in the same partition as u. Also store the first character of the edge label from $u_{\mathcal{P}}$ to $w_{\mathcal{T}}$ in the suffix tree. We call this character $\mathcal{LL}\_first\_char(u_w)$.

3 Substring Search

Given a pattern p and the suffix tree for a set of strings S, the substring matching problem is to locate a position i and a string $s \in S$ such that $s[i..i+|p|-1] = p$ where $|p|$ is the length of p, or conclude that it is impossible to find such a match. In a suffix tree we match p character by character with edge labels of the suffix tree until we can proceed no longer, or until all p's characters have been exhausted in which case a match is found. To search for a pattern p in a suffix tree with our proposed layout, we will traverse the tree partition by partition. The search begins with the partition containing the root node of ST. Let ℓ be a counter that is initialized to zero. The following steps are performed and repeated for each partition $\mathcal{P}$ we encounter.

1. Load $\mathcal{T}_{\mathcal{P}}$ into the main memory.
2. Start from the root r of $\mathcal{T}_{\mathcal{P}}$ and travel down $\mathcal{T}_{\mathcal{P}}$ as follows. Suppose we are at node $u_{\mathcal{T}}$, and let $v_{\mathcal{T}}$ be a child of $u_{\mathcal{T}}$. If $first\_char(v_{\mathcal{T}})$=$p[string\_depth(u_{\mathcal{T}})+1]$ then travel to $v_{\mathcal{T}}$ and repeat this process. Stop if no such $v_{\mathcal{T}}$ can be found, or when $u_{\mathcal{T}}$ is a leaf node in $\mathcal{T}_{\mathcal{P}}$, or p is exhausted.
3. Suppose the previous step stopped at node $v_{\mathcal{T}}$. Compare the substring $s[rep\_suff(v_{\mathcal{T}}) + \ell..rep\_suff(v_{\mathcal{T}}) + \min\{string\_depth(v_{\mathcal{T}}), |p|\}]$ with the substring $p[\ell..\min\{string\_depth(v_{\mathcal{T}}), |p|\}]$. Let lcp be the number of characters matched, set $\ell = \ell + lcp$.
4. Repeat Steps 2 and 3 until the first node $w_{\mathcal{T}}$ such that $string\_depth(w_{\mathcal{T}}) \geq \ell$ is located. Let $u_{\mathcal{T}}$ be the parent of $w_{\mathcal{T}}$ in $\mathcal{T}_{\mathcal{P}}$, and load $\mathcal{LL}(u_{\mathcal{P}}, w_{\mathcal{P}}]$ by using the pointer at $u_{\mathcal{T}}$. Start from the first node $u'_{\mathcal{P}}$ of $\mathcal{LL}(u_{\mathcal{P}}, w_{\mathcal{P}}]$, locate the first node $u''_{\mathcal{P}}$ in $\mathcal{LL}(u_{\mathcal{P}}, w_{\mathcal{P}}]$ such that $string\_depth(u''_{\mathcal{P}}) \geq \ell$. So far the process is similar to the search of PAT-tree proposed by Gonnet *et al* [8].
5. Suppose we stopped at node $u_{\mathcal{P}}$, there are three cases:
   (a) If $string\_depth(u_{\mathcal{P}}) = \ell$ and $\ell = |p|$, then a match is found at node u and the search is stopped.
   (b) Else if $string\_depth(u_{\mathcal{P}}) = \ell$ but $\ell \neq |p|$, i.e. the mismatch occurred immediately after that $u_{\mathcal{P}}$. Find $\mathcal{LL}\_first\_char(u_w) = p[\ell+1]$ and use the pointer to $w_{\mathcal{T}}$ to find the next partition. If no match is found we terminate the search and report no match for p in s.
   (c) Otherwise if $string\_depth(u_{\mathcal{P}}) > \ell$, then we also terminate the search and report no match for p in S.

Using the two-level memory structure proposed by Vitter and Shriver [14], we assume the size of a disk block is B. Since each node requires constant amount of space, the number of nodes a disk block can hold is $\Theta(B)$.

Lemma 3. *Given a string s of length n, and a pattern p of length $|p|$. The number of disk accesses needed to locate p in the suffix tree of s is $O(|p|/B + \log_B n)$.*

Proof. We choose C such that the skeleton partition tree can fit in one disk block. Since the number of nodes in a skeleton partition tree is at most $2C-2$ by

Lemma 2 and each node requires constant space $C = \Theta(B)$. For each partition $\mathcal{P}$ its $\mathcal{T}_{\mathcal{P}}$ is stored in one disk block, so Steps 1 and 2 can be done with one disk access for each partition. Since the search goes through at most $O(\log_B n)$ partitions, the total disk accesses for these steps is $O(\log_B n)$. Over the course of the entire search process, Step 3 makes at most $O(|p|)$ character comparisons, and requires $O(|p|/B + \log_B n)$ disk accesses because both p and the substring being compared are stored contiguously on disk. By similar reasoning Step 4 requires $O(|p|/B + \log_B n)$ disk accesses because the number of nodes loaded for all linked lists combined is less than $|p|$. Finally all the information needed for Step 5 is stored with the node, and no additional disk access is needed. Therefore, the total number of disk accesses is $O(|p|/B + \log_B n)$. □

Note that one occurrence of the pattern in the given set of strings can be found by identifying the first node $u_{\mathcal{T}}$ at or below the position after matching all characters of p, and retrieving the representative suffix $(s, \mathit{rep\_suff}(u_{\mathcal{T}}))$. The algorithm can be extended to return all occurrences of the pattern p using $O(|p|/B + \log_B n + \frac{occ}{B})$ disk accesses where *occ* denotes the number of occurrences.

4 Updating the Suffix Tree

A dynamic suffix tree must support insertion, deletion, and modification of strings. Since a modification operation can be viewed as a deletion followed by an insertion, we concentrate our discussion on the insertion and deletion operations. During insertion and deletion the *size* of a node may be changed. In order to facilitate the calculation of *size*, we choose $C = 2$. By Lemma 1, if $C = 2$ then each partition only has one leaf, and is now a path. The skeleton partition tree contains only two nodes, the root and the leaf, denoted $\mathcal{R}$ and $\mathcal{L}$, respectively. With $C = 2$, it can be easily verified that all the leaves of ST are in a partition of their own.

Note that the bound of asymptotic number of disk accesses for substring search in Lemma 3 is obtained using $C = \Theta(B)$. This result can be achieved even when $C = 2$ by packing multiple skeletal partition trees into the same disk block as outlined in Lemma 5.

4.1 Insertion

In order to insert a string s into an existing suffix tree with $O(n)$ nodes, the suffixes of s are inserted into the suffix tree one by one. Therefore we first introduce the procedure to insert a suffix of s into the suffix tree. When a suffix is inserted into the suffix tree a leaf is added, and an internal node may also be added. For every node u in the suffix tree on the path from the root of the suffix tree to the newly inserted leaf node v, $size(u)$ is increased by one. Because of this change, $rank(u)$ may also increase by one, which will change the partition $\mathcal{P}$. However, the number of nodes that have to be moved to another partition is limited.

Lemma 4. *The insertion of a new suffix into the suffix tree may increase the rank of a node by* 1 *only if it is an ancestor of the new leaf and is the root node of its partition. The ranks of all other nodes are unaffected.*

Proof. If a node is not an ancestor of the new leaf, its size and hence its rank does not change. The size of each node that is an ancestor of the newly inserted leaf will increase by one. Consider a node v that is an ancestor of the new leaf. Suppose v is not the root of a partition and let r denote the root of the partition containing v. If $rank(v)$ were to increase, then $size(v) = C^i - 1$ just before the insertion. Since r is an ancestor of v, then $size(r) > size(v) \Rightarrow size(r) \geq C^i$, so r could not have been in the same partition as v, a contradiction. □

While Lemma 4 applies for any choice of C, we choose $C = 2$ as described in the beginning of the section. While the *size* of many nodes will change after an insertion, it is not necessary to keep track of the correct *size* of all nodes at all times. It is enough to only have the correct *size* for the root $\mathcal{R}$ of all partitions. Since we have chosen $C = 2$ and as stated before each partition $\mathcal{P}$ is now a path in ST, i.e. $\mathcal{P}$ has no branching nodes. So the linked list between $\mathcal{R}_\mathcal{P}$ and $\mathcal{L}_\mathcal{P}$ can now contain the node $\mathcal{R}_\mathcal{P}$ as its head, without fear of duplication. The linked list under the new definition is referred to as $\mathcal{LL}[\mathcal{R}_\mathcal{P}, \mathcal{L}_\mathcal{P}]$. We alter slightly the information stored with $\mathcal{R}_\mathcal{T}$ and $\mathcal{L}_\mathcal{T}$ to facilitate the insertion operation.

- In $\mathcal{R}_\mathcal{T}$, $size(\mathcal{R}_\mathcal{T})$ contains the number of leaves in the suffix tree under $\mathcal{R}_\mathcal{P}$.
- Since $\mathcal{R}_\mathcal{T}$ has only one child, the pointer to the head of $\mathcal{LL}[\mathcal{R}_\mathcal{T}, \mathcal{L}_\mathcal{T}]$ is now $ptr\_\mathcal{LL}(\mathcal{R}_\mathcal{T})$. The pointer $ptr\_\mathcal{LL}(\mathcal{L}_\mathcal{T})$ still points to the tail of $\mathcal{LL}[\mathcal{R}_\mathcal{T}, \mathcal{L}_\mathcal{T}]$.
- The definition for $string\_depth(v)$, $first\_char(v)$ and $(s, rep\_suff(v))$, where v is either $\mathcal{R}_\mathcal{T}$ or $\mathcal{L}_\mathcal{T}$, remain unchanged from before.

The insertion algorithm is applied iteratively to each partition it encounters. Each iteration is divided into two stages. In the first stage we find the appropriate place to insert the new leaf and add a new internal node if necessary. In the second stage we update the partition if the root needs to be moved to another partition. Assume that the *size* parameters are correct for $\mathcal{R}_\mathcal{T}$ of each partition. Suppose we are at partition $\mathcal{P}$, perform Steps 1 to 4 of the search algorithm described in Section 3. Assume after Step 4, the search algorithm stops at a node $u_\mathcal{P}$. One of the following three scenarios will apply.

1. An internal node w needs to be inserted between $\mathcal{R}$ and its parent v in partition $\mathcal{P}'$, and the new leaf attached to $w_\mathcal{P}$. In this case $size(w) = size(\mathcal{R}_\mathcal{T})+1$ and its *rank* can be calculated accordingly. Based on its *rank* one of the following cases is true.
   (a) The new node w has the same rank as $\mathcal{R}_\mathcal{T}$, so it is the new root of $\mathcal{P}$. Put $w_\mathcal{P}$ as the head of $\mathcal{LL}[w_\mathcal{P}, \mathcal{L}_\mathcal{P}]$, set pointer $ptr\_\mathcal{LL}(w_\mathcal{P})$ to $w_\mathcal{P}$.
   (b) The new node w is in a partition by itself, then a new partition is made containing only w. Pointers in $v_{\mathcal{P}'}$ and $\mathcal{R}_\mathcal{P}$ are updated accordingly.
   (c) The new node $w_\mathcal{P}$ is in the same partition $\mathcal{P}'$ as $v_{\mathcal{P}'}$. Node w is inserted after $v_{\mathcal{P}'}$ in $\mathcal{LL}[\mathcal{R}_{\mathcal{P}'}, \mathcal{L}_{\mathcal{P}'}]$ and update the tail pointer stored in $\mathcal{L}_\mathcal{T}$.

2. Else if $string\_depth(u_{\mathcal{P}}) = l$ but $l \neq |p|$, i.e., the mismatch occurred immediately after that $u_{\mathcal{P}}$. Find $\mathcal{LL}\_first\_char(u_w) = p[l+1]$ and use the pointer to $w_{\mathcal{T}}$ to find the next partition. If no match is found then node $u_{\mathcal{P}}$ is where the new leaf should be attached. In either case increment $size(\mathcal{R}_{\mathcal{P}})$ by one.
3. Otherwise if $string\_depth(u_{\mathcal{P}}) > l$, then a new internal node $w_{\mathcal{P}}$ is inserted between $u_{\mathcal{P}}$ and its parent $v_{\mathcal{P}} \in \mathcal{LL}[\mathcal{R}_{\mathcal{P}}, \mathcal{L}_{\mathcal{P}}]$, the new leaf is attached to $w_{\mathcal{P}}$ and $size(\mathcal{R}_{\mathcal{P}})$ is incremented by one.

It is easy to verify that the *size* parameter is correctly set at the end of this stage. From Lemma 4 we know that for each partition $\mathcal{P}$ encountered during the insertion, only the root may need to be moved. If so first remove the first node of $\mathcal{LL}[\mathcal{R}_{\mathcal{P}}, \mathcal{L}_{\mathcal{P}}]$ by changing $ptr\_\mathcal{LL}(\mathcal{R}_{\mathcal{P}})$ to point at the next node in $\mathcal{LL}[\mathcal{R}_{\mathcal{P}}, \mathcal{L}_{\mathcal{P}}]$. The next node becomes the new root of $\mathcal{P}$, so update the values we stored for $\mathcal{R}_{\mathcal{P}}$. All these values can be found except for the new $size(\mathcal{R}_{\mathcal{P}})$. Let r be the old root of $\mathcal{P}$, then $size(\mathcal{R}_{\mathcal{P}}) = size(r) - \sum_{i=1}^{k} size(v_i)$, where v_i is a child of r. The *sizes* of v_i's are known because they are roots of different partitions. The old root r will either become a partition on its own or become a part of the partition of its parent. In the former case, carry out the procedures in 1b), and the procedure in 1c) should be followed in the latter case.

For a partition $\mathcal{P}$, $\mathcal{LL}[\mathcal{R}_{\mathcal{P}}, \mathcal{L}_{\mathcal{P}}]$ may not be able to fit in one disk block. When a disk block becomes full, a new disk block is opened. The second half of the linked list on the current block is copied to the new block. The nodes of the second half of the linked list could be scattered across the disk block.

Lemma 5. *The number of disk accesses needed for the insertion of a suffix is $O(m/B + \log_B n)$, where m is the length of the suffix.*

Proof. Assume the number of nodes that can be contained in a block is $O(B)$ and $B = 2^k$. Under the new scheme each skeleton partition tree contains only two nodes. For a partition $\mathcal{P}$ with rank $rank(\mathcal{P})$, we calculate a $block\_rank(\mathcal{P}) = \lfloor rank(\mathcal{P})/k \rfloor$. For partitions $\mathcal{P}'$ and $\mathcal{P}''$, without lost of generality assume $\mathcal{R}_{\mathcal{P}''}$ is a child of $v \in \mathcal{P}'$. If $block\_rank(\mathcal{P}') = block\_rank(\mathcal{P}'')$ we put $\mathcal{T}_{\mathcal{P}'}$ and $\mathcal{T}_{\mathcal{P}''}$ on the same disk block. So each time a new disk block containing skeleton partition trees is loaded the *block_rank* decreases by one, so $O(\log_B n)$ disk accesses are sufficient for loading all blocks containing the skeleton partition trees needed by the algorithm.

Let $\mathcal{R}_{\mathcal{P}'}$ be a child of $\mathcal{R}_{\mathcal{P}}$ in ST such that $block\_rank(\mathcal{P}') = block\_rank(\mathcal{P})$. We store in $\mathcal{R}_{\mathcal{T}}$ of $\mathcal{P}$ a pointer to $\mathcal{R}_{\mathcal{T}}$ of partition $\mathcal{P}'$, and the first character of the edge label. Therefore, we can find a partition $\mathcal{P}''$ in the same disk block as $\mathcal{P}$, using Step 2 of the substring search algorithm. Then use *rep_suff* at $\mathcal{P}''$ to decided how to navigate through all the skeleton partition trees on the same disk block. Thus the complexity is the same as search. □

To speed up insertion of all suffixes of a string, suffix links are used. For nodes $u, v \in ST$ there is a suffix link from u to v, denoted $SL(u) = v$, if the concatenation of the edge labels from the root to u and v are $a\beta$ and β, respectively. We first briefly introduce McCreight's suffix tree construction algorithm [11], then show how these ideas can be used in our layout scheme.

To insert a string s of length m into a suffix tree of size n, the suffixes of s are inserted into the suffix tree one by one. Suppose we have just inserted suffix s_i as a leaf, if a new internal node w is created to attach the leaf representing s_i, then go to w's parent and let l be the length of the edge label between w and its parent. Otherwise go to the parent of the leaf and let $l = 0$. Assume we are at node u now, take the suffix link from node u to v. Compare only the first character of each edge label with the appropriate characters of s and skip down, repeat this until l characters have been skipped. If we are inside an edge label, insert a new internal node w', attach the leaf representing s_{i+1} to w' and set $SL(w) = w'$. Otherwise suppose we are at a node w'. Set $SL(w) = w'$ and continue down from w' by comparing appropriate characters of s with the edge labels, until a place to insert s_{i+1} is found.

If we maintain a suffix link for each node $u_{\mathcal{P}}$ to $v_{\mathcal{P}}$, then we can follow the above algorithm to insert suffixes one by one in our layout. To skip the l characters takes at most $O(l/B + \log_B(n+m))$ disk accesses. After the insertion of each suffix, for each partition $\mathcal{P}$ encountered on the path from the root of ST to the new leaf, the size of $\mathcal{R}_{\mathcal{P}}$ is incremented by one, and $\mathcal{R}_{\mathcal{P}}$ is moved to another partition if necessary. Updating all of these partitions takes at most $O(\log_B(n+m))$ number of disk accesses. Therefore the total number of disk accesses required for inserting all suffixes of a string of length m is $O(m \log_B(n+m))$.

4.2 Maintaining Suffix Links

In our string insertion and deletion algorithms, suffix links need to be maintained for each node $u_{\mathcal{P}}$. This can be accomplished by maintaining bidirectional suffix links such that when $u_{\mathcal{P}}$ is moved, all nodes $v_{\mathcal{P}'}$ with $SL(v_{\mathcal{P}'}) = u_{\mathcal{P}}$ can be identified and updated. We note that the string depth of a node in ST never changes. So for nodes $u_{\mathcal{P}}, v_{\mathcal{P}}$ in $\mathcal{LL}[\mathcal{R}_{\mathcal{P}}, \mathcal{L}_{\mathcal{P}}]$ such that $u_{\mathcal{P}}$ is the parent of $v_{\mathcal{P}}$, we can leave enough space between $u_{\mathcal{P}}$ and $v_{\mathcal{P}}$ for future insertions. The amount of space is proportional to the length of the edge label between $u_{\mathcal{P}}$ and $v_{\mathcal{P}}$. This way the suffix links will only change when a node is moved to another partition. The complexity of searching, insertion and deletion is not affected. Ferragina and Grossi [7] have proposed three approaches to maintain their *succ* pointers, which can also be applied to maintain the suffix links in our approach.

4.3 Deletion

The deletion process is analogous to the insertion process. Similar to Lemma 4 only the leaf of a partition may need to be moved to another partition for each of the partitions encountered during the deletion process. In places where $size(u)$ is incremented by one in the insertion process, $size(u)$ should be decremented by one. Since some of the strings will be deleted, the $(s, rep\_suff(u_{\mathcal{T}}))$ entries may no longer be valid for $u_{\mathcal{T}}$ that is an ancestor of a deleted leaf. In this case after the deletion of a leaf we traverse upwards in the tree, and replace the

$(s, \mathit{rep\_suff}(u_{\mathcal{T}}))$ entries with the suffixes represented by a sibling of the deleted leaf, or $(s, \mathit{rep\_suff}(v_{\mathcal{T}}))$ where $v_{\mathcal{T}}$ is a sibling of the deleted leaf.

5 Discussion

In our layout scheme, it is possible that many skeletal partition trees may be small and not occupy a full disk block. Even though a skeletal partition tree can have as many as $C - 2$ nodes, it can also have as few as just two nodes. For example, the suffix tree of the string $A^n\$$ has $\Theta(n)$ number of disk blocks with only one node. Note that this is true even if a large value of C such as $\Theta(B)$ is chosen. Choosing $C = \Theta(B)$ ensures optimal number of disk accesses for substring search, even if many disk blocks are sparsely occupied. This is an interesting contrast with the suffix tree layout of Clark and Munro [3] where the focus is on succinct representation of suffix trees to conserve secondary storage, but this scheme does not provide optimal worst case bound for substring search. Our scheme provides such guarantees despite not trying to minimize the number of disk blocks for suffix tree storage.

While not needed for ensuring asymptotic performance when $C = \Theta(B)$, we can and should pack as many skeletal partition trees into a single disk block for efficient storage. This packing is always beneficial and has no harmful side effects. If C is chosen to be small, such packing is necessary to obtain optimal disk accesses as outlined in the proof of Lemma 5.

The main advantage of our approach is that it provides the same performance guarantees as string B-trees without sacrificing the structure of suffix trees. This will make it easier to co-exist with the large number of applications that already use suffix trees as the underlying data structure. One limitation of our approach is that it is applicable only to constant sized alphabets while string B-trees do not have this limitation. Traversing any root to leaf path in string B-tree incurs the same number of disk accesses. This number is potentially different for each path in our layout with the worst-case asymptotic performance same as for string B-trees. Each disk access in our algorithm increases the number of characters matched with p, and this may not be true for string B-tree. Thus, searching for a pattern p takes $O(\min\{p, p/B + \log_B n\})$ number of disk access, where n is the number of characters indexed.

6 Conclusions

In this paper, we present a new suffix tree layout scheme for secondary storage and provide algorithms with provably good worst-case performance for search and update operations. The performance of our algorithms matches what can be obtained by the use of string B-trees, a data structure specifically designed to efficiently support string operations on secondary storage. Suffix trees are extensively used in biological applications. As our scheme provides how to efficiently store and operate on them in secondary storage that is competitive with the best available alternatives, the research presented provides justification for using suffix trees in secondary storage as well. It is important to compare how the presented algorithms compare in practice with other storage schemes developed so far (those with and without provable bounds on disk accesses), and such work remains to be carried out.

References

1. S.J. Bedathur and J.R. Haritsa. Engineering a fast online persistent suffix tree construction. In *Proc. 20th International Conference on Data Engineering*, pages 720–731, 2004.
2. S.J. Bedathur and J.R. Haritsa. Search-optimized suffix-tree storage for biological applications. In *Proc. 12th IEEE International Conference on High Performance Computing*, pages 29–39, 2005.
3. D.R. Clark and J.I. Munro. Efficient suffix trees on secondary storage. In *Proc. 7th ACM-SIAM Symposium on Discrete Algorithms*, pages 383–391, 1996.
4. M. Farach. Optimal suffix tree construction with large alphabets. In *Proc. 38th Annual Symposium on Foundations of Computer Science*, pages 137–143, 1997.
5. M. Farach-Colton, P. Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *Journal of the ACM*, 47(6):987–1011, 2000.
6. P. Ferragina and R. Grossi. Fast string searching in secondary storage: theoretical developments and experimental results. In *Proc. 7th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 373–382, 1996.
7. P. Ferragina and R. Grossi. The string B-tree: A new data structure for string search in external memory and its applications. *Journal of the ACM*, 46(2):236–280, 1999.
8. G.H. Gonnet, R.A. Baeza-Yates, and T. Snider. *Information Retrieval: Data Structures & Algorithms*, chapter 5: "New indices for text: PAT trees and PAT arrays", pages 66–82. 1992.
9. R. Grossi and J.S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Proc. 32nd Annual ACM Symposium on Theory of Computing*, pages 397–406, 2000.
10. E. Hunt, M.P. Atkinson, and R.W. Irving. Database indexing for large DNA and protein sequence collections. *The VLDB Journal*, 11(3):256–271, 2002.
11. E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23:262–272, 1976.
12. S. Tata, R.A. Hankins, and J.M. Patel. Practical suffix tree construction. In *Proc. 13th International Conference on Very Large Data Bases*, pages 36–47, 2004.
13. E. Ukkonen. On-line construction of suffix-trees. *Algorithmica*, 14:249–260, 1995.
14. J.S. Vitter and E.A.M. Shriver. Algorithms for parallel memory I: Two-level memories. *Algorithmica*, 12(2/3):110–147, 1994.
15. P. Weiner. Linear pattern matching algorithms. In *Proc. 14th Symposium on Switching and Automata Theory*, pages 1–11, 1973.

Geometric Suffix Tree: A New Index Structure for Protein 3-D Structures

Tetsuo Shibuya

Human Genome Center, Institute of Medical Science, University of Tokyo
4-6-1 Shirokanedai, Minato-ku, Tokyo 108-8639, Japan
tshibuya@hgc.jp

Abstract. Protein structure analysis is one of the most important research issues in the post-genomic era, and faster and more accurate query data structures for such 3-D structures are highly desired for research on proteins. This paper proposes a new data structure for indexing protein 3-D structures. For strings, there are many efficient indexing structures such as suffix trees, but it has been considered very difficult to design such sophisticated data structures against 3-D structures like proteins. Our index structure is based on the suffix trees and is called the geometric suffix tree. By using the geometric suffix tree for a set of protein structures, we can search for all of their substructures whose RMSDs (root mean square deviations) or URMSDs (unit-vector root mean square deviations) to a given query 3-D structure are not larger than a given bound. Though there are $O(N^2)$ substructures, our data structure requires only $O(N)$ space where N is the sum of lengths of the set of proteins. We propose an $O(N^2)$ construction algorithm for it, while a naive algorithm would require $O(N^3)$ time to construct it. Moreover we propose an efficient search algorithm. We also show computational experiments to demonstrate the practicality of our data structure. The experiments show that the construction time of the geometric suffix tree is practically almost linear to the size of the database, when applied to a protein structure database.

1 Introduction

Analyzing 3-D structures of proteins is very important in molecular biology and more and more protein structures are solved today with the aid of state-of-the-art technologies such as nuclear magnetic resonance (NMR) techniques, as seen in the increasing number of PDB [4] entries: 35,813 on March 28, 2006. It is said that structurally similar proteins tend to have similar functions even if their amino acid sequences are not similar to each other. Thus it is very important to find proteins with similar structures (even in part) from the growing database to analyze protein functions.

Structure similarity search methods for protein structure databases can be classified into two types. One is by comparing each database entry with the query. There are many comparison algorithms for protein structures [10], and the results could be very accurate, but it will require enormous amount of time

M. Lewenstein and G. Valiente (Eds.): CPM 2006, LNCS 4009, pp. 84–93, 2006.

to apply against very large databases. The other approach is by indexing with some important features of structures [1,3,6,5,8,12]. In ordinary, these methods can search queries more efficiently, but with less accuracy than the pairwise comparison-based methods. The accuracy of comparison of two protein structures is often measured by RMSD (root mean square deviation) [2,9,17] or sometimes by URSMD (unit-vector root mean square deviation) [7,15]; see section 2.1 for more details. But it has been considered too difficult to design indexing structures that strictly consider the RMSD or the URMSD.

In this paper, we propose a new data structure called the geometric suffix tree that succeeds in finding all the substructures whose RMSD or URMSD to a query is not larger than some given bound. As the name implies, our data structure is very similar to the famous suffix tree for character strings: The edges in the ordinary suffix tree represent substrings of texts, while the edges in the geometric suffix tree represent 3-D substructures of protein 3-D structures. The geometric suffix tree can be stored in $O(N)$ space where N is the sum of the lengths of the proteins in the database. We propose an $O(N^2)$ construction algorithm for it, though it takes $O(N^3)$ time if we construct the data structure naively. Furthermore, the experiments will show that the construction time of the geometric suffix tree is almost linear to the size of the database in practice, when applied to a protein structure database. Moreover, we propose an efficient search algorithm for substructure queries. This data structure is also useful for finding structural motifs, clustering substructures, and so on.

Organization of this paper is as follows. In section 2, we explain related work as preliminaries. In section 3, we describe definitions of two data structures: the geometric trie and the geometric suffix tree, where the geometric trie is the basis for the geometric suffix tree. In sections 4 and 5, we explain algorithms for constructing the data structure and algorithms for searching queries. In section 6, we demonstrate experimental results. In section 7, we conclude our results.

2 Related Work

2.1 RMSD and URMSD

A protein is a chain of amino acids. Each amino acid has one unique carbon atom named C_α, and we often use the coordinates of the C_α atom as the representative position of the amino acid. The set of C_α atom positions of all the amino acids in a protein is called the backbone of the protein, and is often used to ease protein structure analysis in previous work. The backbone is topologically linear, but it forms a geometrically very complex structure in the 3-D space. In this paper, we consider the backbone as the target to index.

The most popular and basic measure to determine geometric similarity between two sets of points like the positions of backbone atoms is the RMSD (root mean square deviation) [2,9,17], if we know which atom in one structure corresponds to which atom in the other. The measure describes the similarity of two structures when one of the point sets is rotated and translated reasonably. Let the two sets of points to be compared be $P = \{\boldsymbol{p}_1, \boldsymbol{p}_2, \ldots, \boldsymbol{p}_n\}$ and

$Q = \{\boldsymbol{q}_1, \boldsymbol{q}_2, \ldots, \boldsymbol{q}_n\}$, where $\boldsymbol{p}_i$ and $\boldsymbol{q}_j$ are coordinates in the 3-D space, and we consider $\boldsymbol{p}_i$ corresponds to $\boldsymbol{q}_i$ for each i. The RMSD is the minimum value of $\{(\sum_{i=1}^{n} \|\boldsymbol{p}_i - (R \cdot \boldsymbol{q}_i + \boldsymbol{v})\|^2)/n\}^{1/2}$ over possible rotation matrices R and translation vectors $\boldsymbol{v}$, where $\| \cdot \|$ denotes the norm. Let $\hat{R}(P,Q)$ and $\hat{\boldsymbol{v}}(P,Q)$ be R and $\boldsymbol{v}$ that minimizes the value. We call $\sum_{i=1}^{n} \|\boldsymbol{p}_i - (\hat{R}(P,Q) \cdot \boldsymbol{q}_i + \hat{\boldsymbol{v}}(P,Q))\|^2$ the MSSD (minimum sum squared distance) of P and Q.

It is known that $\hat{\boldsymbol{v}}(P,Q) = \sum_{i=1}^{n} (\boldsymbol{p}_i - \hat{R}(P,Q) \cdot \boldsymbol{q}_i)/n$, *i.e.*, the distance is minimized when the centroids of the two point sets are translated to the same point. Hence, if both of the point sets are translated so that their centroids are located at the origin of the coordinates, the RMSD/MSSD problem is reduced to a problem of finding R that minimizes $f(R) = \sum_{i=1}^{n} \|\boldsymbol{p}_i - R \cdot \boldsymbol{q}_i\|^2$. We can find $\hat{R}(P,Q)$ in linear time by using singular value decomposition (SVD) [2, 17] as follows. Let $H = \sum_{i=1}^{n} \boldsymbol{p}_i \cdot \boldsymbol{q}_i^t$. Then $f(R)$ can be described as $\sum_{i=1}^{n} (\boldsymbol{p}_i^t \boldsymbol{p}_i + \boldsymbol{q}_i^t \boldsymbol{q}_i) - trace(R \cdot H)$, and $trace(RH)$ is maximized when $R = VU^T$, where $U\Lambda V$ is the SVD of H. Hence $\hat{R}(P,Q)$ can be obtained in constant time from H (see [13] for SVD algorithms). Note that there are rare degenerate cases where $det(VU^T) = -1$, which means that VU^T is a reflection matrix. We ignore the degenerate cases in this paper. In this way, we can compute the RMSD/MSSD values in $O(n)$ time.

The URMSD (unit-vector root mean square deviation) [7, 15] is a variation of the RMSD. The RMSD is sometimes influenced badly by very distant pairs of points, and the URMSD is designed to avoid such influence. It is the minimum value of $\{(\sum_{i=1}^{n-1} \|\boldsymbol{p}'_i - R \cdot \boldsymbol{q}'_i\|^2)/(n-1)\}^{1/2}$ over possible rotation matrices R, where $\boldsymbol{p}'_i = (\boldsymbol{p}_{i+1} - \boldsymbol{p}_i)/\|\boldsymbol{p}_{i+1} - \boldsymbol{p}_i\|$ and $\boldsymbol{q}'_i = (\boldsymbol{q}_{i+1} - \boldsymbol{q}_i)/\|\boldsymbol{q}_{i+1} - \boldsymbol{q}_i\|$. Let $\check{R}(P,Q)$ be R that minimizes the value. We call $\sum_{i=1}^{n-1} \|\boldsymbol{p}'_i - \check{R}(P,Q) \cdot \boldsymbol{q}'_i\|^2$ the UMSSD (unit-vector minimum sum squared distance). The URMSD/UMSSD can be computed with the same strategy in $O(n)$ time, *i.e.*, by computing the SVD of $H' = \sum_{i=1}^{n} \boldsymbol{p}'_i \cdot (\boldsymbol{q}'_i)^t$.

2.2 Suffix Trees

The suffix tree [11, 14, 16, 19, 20] of a string $S \in \Sigma^n$ is the compacted trie of all the suffixes of $S^+ = S\$$ where \$ is a character such that $\$ \notin \Sigma$. This data structure can be stored in $O(n)$ space and moreover is known to be buildable in $O(n)$ time. Each leaf represents a suffix of the string S^+, and each node represents some substring. This data structure is very useful for various problems in sequence pattern matching. Using it, we can query a substring of length m in $O(m)$ time, we can find frequently appearing substrings in a given sequence in linear time, we can find a common substring of many sequences in linear time, and so on [14].

Not much work has been done for applying this data structure to biomolecular structures. The PSIST [12] is the only index data structure for protein structures based on the suffix trees as far as we know. It converts local features of the amino acid chain (*i.e.*, some feature vectors computed from only several adjacent atoms) into some alphabets and constructs suffix trees over the converted alphabet sequences, without considering global similarity measures like the RMSD or the

URMSD at all. For RNA (secondary) structures, the s-suffix tree [18], a generalization of the suffix tree, can be used for mining some interesting RNA structures from sequence databases, but it cannot be applied to protein 3-D structures.

3 Geometric Suffix Tree Data Structure

In this section, we describe the definition of the geometric suffix tree. Before defining the geometric suffix tree, we define a data structure called the geometric trie for a set of protein structures.

Consider a set of n protein structures represented by the sequence of their C_α atom coordinates. Let W_i be the i-th structure, where the 3-D coordinates of the j-th C_α atom is denoted as $\boldsymbol{w}_j^{(i)}$, and let ℓ_i be the length of W_i (*i.e.*, number of C_α atoms). Let $W_i[j..k]$ denote $\{\boldsymbol{w}_j^{(i)}, \boldsymbol{w}_{j+1}^{(i)}, \ldots, \boldsymbol{w}_k^{(i)}\}$, which means a structure formed by the $(k-j+1)$ atoms from the j-th atom to the k-th atom in W_i. We call it a substructure of W_i. Moreover, we call $W_i[1..j]$ a prefix substructure of W_i. Conversely, $W_i[j..\ell_i]$ is called a suffix substructure. From now on, we define two versions of the geometric trie: one based on the RMSD/MSSD (which we call the RMSD Geometric trie (RGT)) and the other based on the URMSD/UMSSD (which we call the URMSD geometric trie (UGT)). The geometric trie for the set of protein structures is defined as a rooted tree data structure that has following features:

1. All the internal nodes (nodes other than the leaves) except for the root have more than one child, while the root has only one child. (It corresponds to the fact that a structure with only one atom is always the same structure.) The trie has n leaves, each of which corresponds to one protein structure, and no two leaves correspond to the same structure. Let $leaf(i)$ denote the leaf that corresponds to W_i.
2. All the internal edges (*i.e.*, edges that end at internal nodes) and some external edges (*i.e.*, edges that end at leaves) correspond to a substructure of some protein. If the corresponding substructure of edge e is $P(e) = W_i[j..k]$, we represent it with only three values: i, j, and $length(e) = k - j + 1$. Let $length(e) = 0$ if e is an external edge without a corresponding substructure. We call the value $length(e)$ the edge length of e. Let the $depth(v)$ be the sum of all the edge lengths on the path from the root to v, which we call the depth of v.
3. Add to the three values that represent its corresponding substructure, each edge with a corresponding substructure has information of a rotation matrix $R(e)$ and a translation vector $\boldsymbol{v}(e)$. $R(e)$ and $\boldsymbol{v}(e)$ must satisfy the condition in the items 4 and 5.
4. Let $S(e)$ be a 3-D structure obtained by rotating $P(e)$ with $R(e)$ and translating it with $\boldsymbol{v}(e)$ after that. We call $S(e)$ the 'edge structure' of e. Note that $S(e)$ (not $P(e)$) corresponds to the substring represented by an edge in an ordinary suffix tree for alphabet strings. The 'node structure' $S(x)$ for a node x is defined as a structure that can be obtained by concatenating 'edge

structures' of the edges on the path from the root to the node x. For any leaf $v = leaf(i)$ and its node structure $S(v)$, the MSSD (in case of RGTs, or the UMSSD in case of UGTs) between any prefix substructure of $S(v)$ and the prefix substructure of W_i of the same length must not be larger than some given fixed bound b. (Note that b is unrelated to the RMSD/URMSD bound d used in the next section for searching structures.)

5. For an edge $e = (v, w)$ with some corresponding substructure $P(e)$, the 'branching structure' $str(e)$ is defined as a structure that is obtained by adding the coordinates of the first atom of $S(e)$ (*i.e.*, $S(e)[1]$) after the coordinates sequence $S(v)$. For any internal node v with more than one outgoing edge with corresponding substructures, the MSSD (for RGTs, or the UMSSD for UGTs) between $str(e_1)$ and $str(e_2)$ must be larger than b, where e_1 and e_2 are arbitrary two of the edges.

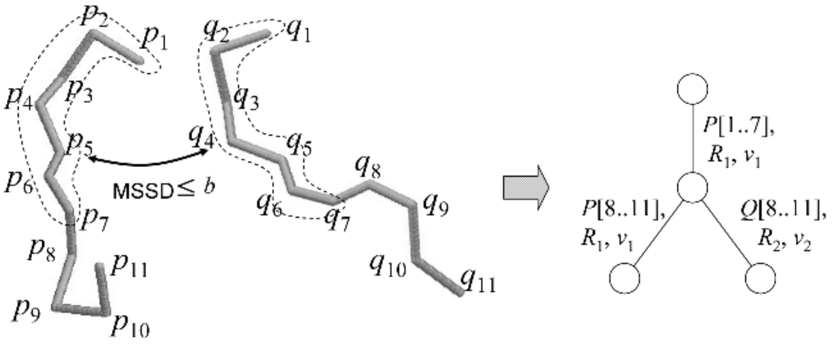


Fig. 1. A geometric trie for two protein 3-D structures

As there are only $O(n)$ nodes/edges in the trie and we need only $O(1)$ memory for each edge/node, the total memory space to store the geometric trie is only $O(n)$. Note that the data structure is not unique for a fixed set of protein structures. Figure 1 shows an example of the geometric trie constructed for two structures P and Q. In the figure, we consider the MSSD of $P[1..7]$ and $Q[1..7]$ is not larger than b, while the MSSD between $P[1..8]$ and $Q[1..8]$ is larger than b.

Now we can define the geometric suffix tree: The geometric suffix tree for a set of proteins is the geometric trie for all the suffix substructures of all the proteins in the set. It is easy to see that we need $O(N)$ space to store the geometric suffix tree, where N is the sum of the lengths of the proteins.

4 Constructing Geometric Suffix Trees

In this section, we describe how to construct the geometric tries and the geometric suffix trees. Given a set of n protein structures W_i and some given MSSD (for RGTs or UMSSD for UGTs) bound b, we can construct the geometric trie by adding structures one by one as follows:

Algorithm 1. At first, construct a tree with only the root node. For each protein structure W_i, set the root node to v and do the following.

1. From among a set of v's outgoing edges with some corresponding substructures, find an edge e such that the MSSD (for RGTs, or the UMSSD for UGTs) between $W_i[1..\mathit{depth}(v)+1]$ and $\mathit{str}(e)$ is smaller than b. If there is more than one such edge, choose an arbitrary one (or preferably the one with the smallest MSSD (or UMSSD)). If no such edge exists, go to step 2. Otherwise go to step 3.
2. Add a new outgoing edge $e' = (v, w)$ to v, and let the new leaf w correspond to W_i. Let $P(e')$ be $W_i[\mathit{depth}(v)+1..\ell_i]$. If v is the root, let $R(e')$ be the identity matrix and let $\boldsymbol{v}(e')$ be a zero vector. Otherwise, in case of RGTs, let $R(e')$ be $\hat{R}(S(v), W_i[1..\mathit{depth}(v)])$, and let $\boldsymbol{v}(e')$ be $\hat{\boldsymbol{v}}(S(v), W_i[1..\mathit{depth}(v)])$. In case of UGTs, let $R(e')$ be $\check{R}(S(v), W_i[1..\mathit{depth}(v)])$, and let $\boldsymbol{v}(e')$ be $(S(v)[\mathit{depth}(v)] - R(e') \cdot W_i[\mathit{depth}(v)])$. Notice that, in both cases, $R(e')$ and $\boldsymbol{v}(e')$ represents alignment between $S(v)$ and $W_i[1..\mathit{depth}(v)]$. Then stop.
3. Let w be the node where the edge e ends. Find the longest prefix substructures of $S(w)$ and W_i whose MSSD (for RGTs, or UMSSD for UGTs) is not larger than b, and let the length be ℓ. If $\ell < \mathit{depth}(w)$ go to step 4. If $\ell = \mathit{depth}(w)$ and $\ell < \ell_i$, set w to v and go to step 1. Otherwise, add a new outgoing edge (w, u) with no corresponding substructure, and let the new leaf u correspond to the structure W_i. Then stop.
4. Insert a new node u between v and w. Let $e_1 = (v, u)$ and let $e_2 = (u, w)$. Let $P(e_1)$ be the prefix substructure of $P(e)$ of length $(\ell - \mathit{depth}(v))$, and $P(e_2)$ be the suffix substructure of $P(e)$ of length $(\mathit{depth}(w) - \ell)$. Let $R(e_1)$ and $R(e_2)$ be the same matrix as $R(e)$, and $\boldsymbol{v}(e_1)$ and $\boldsymbol{v}(e_2)$ be the same vector as $\boldsymbol{v}(e)$. Add a new outgoing edge $e'' = (u, x)$ to u, and let the new leaf x correspond to the structure W_i. If $\ell = \ell_i$, let e'' have no corresponding substructure. Otherwise, let the corresponding substructure $P(e'')$ be $W_i[\ell+1..\ell_i]$. In case of RGTs, let $R(e'')$ be $\hat{R}(S(u), W_i[1..\ell])$ and let $\boldsymbol{v}(e'')$ be $\hat{\boldsymbol{v}}(S(u), W_i[1..\ell])$. In case of UGTs, let $R(e'')$ be $\check{R}(S(u), W_i[1..\ell])$ and let $\boldsymbol{v}(e'')$ be $(S(u)[\ell] - R(e'') \cdot W_i[\ell])$. Then stop.

With the same algorithm, we can construct the geometric suffix tree: Just consider that W_i is the i-th suffix substructure.

Recall that it takes $O(\ell)$ time to compute the MSSD or the UMSSD (and the rotation matrix and the translation vector related to it) between two structures of size ℓ. Thus, if we execute the algorithm naively, we would need $O((\ell_i + n) \cdot \ell_i)$ time to add W_i to the tree, because there are at most n branches on the path from the root node to some leaf. Accordingly, we need $O(\sum_i^n \{(\ell_i + n) \cdot \ell_i\})$ time for constructing the geometric trie. It means that the above algorithm requires $O(N^3)$ time to construct the geometric suffix tree, where N is the sum of all the structure lengths. From now on we present how to reduce it to $O(N^2)$.

We reduce the computation time by proposing an incremental MSSD/UMSSD computation technique. Recall that the MSSD of two protein structures $P[1..j]$ and $Q[1..j]$ can be obtained by computing the SVD of $H =$

$\sum_{i=1}^{j}(\boldsymbol{p}_i - \boldsymbol{c}_P)\cdot(\boldsymbol{q}_i - \boldsymbol{c}_Q)^t$ where $\boldsymbol{c}_P$ and $\boldsymbol{c}_Q$ are the centroids of P and Q. H can be computed in constant time if we are given $f_P(j) = \sum_i^j \boldsymbol{p}_i$, $f_Q(j) = \sum_i^j \boldsymbol{q}_i$, and $g(j) = \sum_i^j \boldsymbol{p}_i \cdot \boldsymbol{q}_i^t$, as $H = g(j) - \{f_P(j)\cdot(f_Q(j))^t\}/j$. Add to these values, we need $h_P(j) = \sum_{i=1}^{j} \boldsymbol{p}_i^t \boldsymbol{p}_i$ and $h_Q(j) = \sum_{i=1}^{j} \boldsymbol{q}_i^t \boldsymbol{q}_i$ to compute the MSSD or RMSD values in constant time. Notice that all of these can be computed incrementally in constant time from $f_P(j-1)$, $f_Q(j-1)$, $g(j-1)$, $h_P(j-1)$, and $h_Q(j-1)$. It means that we can add the structure W_i to the tree in $O(\ell_i + n)$ time, and accordingly we can construct the RGT in $O(N + n^2)$ time. In conclusion, we can construct the RMSD-based geometric suffix tree in $O(N^2)$ time.

Similarly, we can compute the UMSSD of two protein structures $P[1..j]$ and $Q[1..j]$ in constant time if we are given $g'(j) = \sum_{i=1}^{j} \boldsymbol{p}'_i \cdot (\boldsymbol{q}'_i)^t$, $h'_P(j) = \sum_{i=1}^{j} (\boldsymbol{p}'_i)^t \boldsymbol{p}'_i$, and $h'_Q(j) = \sum_{i=1}^{j} (\boldsymbol{q}'_i)^t \boldsymbol{q}'_i$. We can easily see that these can also be computed from $g'(j-1)$, $h'_P(j-1)$ and $h'_Q(j-1)$ in constant time. Therefore we conclude that the UGTs and the URMSD-based geometric suffix trees can be constructed in the same time bound as the RGTs and the RMSD-based geometric suffix trees: We can construct the UGTs in $O(N + n^2)$ time, and the URMSD-based geometric suffix trees in $O(N^2)$ time.

5 Geometric Suffix Tree Applications

There are two important features on the RMSD/MSSD (or URMSD/UMSSD) measures. One is that the MSSD (or UMSSD) of two structures P and Q (of the same length) is always larger than or equal to that of P' and Q', where P' and Q' are any same-length prefix substructures of P and Q. The other is that there is a triangle inequality $c \leq a + b$ where a is the RMSD (URMSD) between P and Q, b is that between Q and R, and c is that between R and P, for any set of three structures P, Q, and R of same lengths.

Using these features, all maximal substructures whose RMSD (or URMSD) to a query $Q[1..m]$ is within some bound d can be computed efficiently as follows. Let 'representative structure' mean any prefix substructure of the 'node structure' of any node in the geometric suffix tree. First, we find all the maximal representative substructures whose RMSD (or URMSD) to the query Q is within $\sqrt{b/m} + d$ by just doing a depth-first or breadth-first search from the root, where b is the MSSD (or UMSSD) bound used for constructing the geometric suffix tree. Let E be the set of edges to which the collected representative substructures correspond. After that, find all the leaves that are descendants of the edges in E. As the suffixes that correspond to the collected leaves are candidates of the answer substructures (and there are no candidates elsewhere), check their RMSDs (or URMSDs) one by one.

Ordinary suffix trees have tremendous number of applications in string pattern matching [14]. Like them, applications of the geometric suffix trees are not limited to the database search. A long representative structure whose corresponding edge has many descendants is a repeated structure in a protein structure, which could have some meaning. By constructing the geometric suffix tree for several functionally-related protein structures, we could find structural motifs. We could further use this fundamental data structure for designing more complicated combinatorial pattern matching algorithms on protein structures, such as structural alignment algorithms, clustering/classification algorithms and functional prediction algorithms.

6 Experimental Results

In this section, we demonstrate the performance of the geometric suffix trees through experiments on a Sun Fire 15K super computer with 288 GB memory and 96 UltraSPARC III Cu CPUs running at 1.2GHz. Note that we used only one CPU for each experiment. As a data for experiments, we used a set of 228 myoglobin or myoglobin-related PDB data files containing 275 protein structures. The total number of amino acids in the protein set is 41,719.

Table 1 shows the computation time for constructing the RMSD-based geometric suffix trees against databases of different sizes, setting 400Å^2 to the MSSD bound. In the experiment (1), we used all the 275 proteins to index. In the experiments (2)-(5), we used different subsets of them. The '#sequence(#a.a.)' column shows the numbers of sequences and amino acids contained in the protein sets. The 'Time' column shows the computation time, while the 'GST Size' column shows the numbers of nodes in the constructed geometric suffix trees. According to the table, the computation time is almost linear to the size of the databases, though the theoretical time bound is $O(N^2)$. It is reasonable as there should be some reasonable upper bound on protein lengths.

Next, we examined the query speed on the RMSD-based geometric suffix trees with different MSSD bounds. Table 2 shows the results, where '$b = \ldots$' denotes the MSSD bound in Å^2. We used two protein substructures of the same length as queries: In experiment (a), we used as a query a substructure from

Table 1. Time for constructing the geometric suffix trees ($b = 400\text{Å}^2$)

Database	#sequence	(#a.a.)	Time (sec)	GST Size
(1) Entire database	275	(41,719)	53.15	57,241
(2) Subset A	198	(30,061)	36.37	41778
(3) Subset B	111	(16,983)	17.68	25,942
(4) Subset C	54	(8,267)	7.91	13,050
(5) Subset D	20	(3,069)	2.89	4,855

Table 2. Query time (sec) on the geometric suffix trees with various MSSD bounds

Queries		$b = 1$	$b = 100$	$b = 400$	$b = 900$	$b = 1600$	$b = 2500$	#found
(a)	$d = 1.0\text{Å}$	1.63	0.56	0.39	0.43	0.60	0.87	19
	$d = 5.0\text{Å}$	11.70	5.08	5.66	6.55	6.63	6.63	217
(b)	$d = 1.0\text{Å}$	1.63	0.73	0.48	0.33	0.19	0.21	0
	$d = 5.0\text{Å}$	16.13	7.83	7.93	8.00	7.58	7.20	0

the 20th amino acid to the 69th amino acid of a myoglobin's structure obtained from the PDB entry named 103M. In experiment (b), we used a protein that is unrelated to myoglobins: A substructure from the 20th amino acid to the 69th amino acid of a rhodopsin's structure obtained from the PDB entry named 1F88. In both experiments, we examined query time by setting two different RMSD bounds: $d = 1.0\text{Å}$ and $d = 5.0\text{Å}$. In the table, the '#found' column shows the numbers of found substructures similar to the query. According to the experiments, the query is very fast when the RMSD bound for the query is small in both experiments. Note that we can observe similar phenomenon on ordinary suffix trees: It is known that the inexact matching on suffix trees is (not) efficient when there is (not) a small edit distance limit.

7 Concluding Remarks

We proposed a new data structure called the geometric suffix tree for indexing the protein 3-D structures. The data structure can be stored in $O(N)$ space where N is the database size, and we presented an $O(N^2)$ construction algorithm for it. Moreover, we showed through experiments that we can build the data structure in quasi-linear time in practice. We also showed that we can search for queries very efficiently with the geometric suffix tree.

It is an open problem whether we can improve the theoretical time bound for building the geometric suffix tree. We are now working on utilizing this data structure for further combinatorial matching problems and machine learning problems on protein structures. We suppose this work is just the beginning.

Acknowledgements

The author would like to thank Prof. Tatsuya Akutsu for fruitful discussions on protein comparison algorithms. All the computational experiments in this research were done on the Super Computer System, Human Genome Center, Institute of Medical Science, University of Tokyo.

References

1. T. Akutsu, K. Onizuka, and M. Ishikawa. New hashing techniques and their application to a protein database system. *Proc. Hawaii Int. Conf. System Sciences (HICSS-28)*, Vol. 5, pp. 197-206, 1995.
2. K. S. Arun, T. S. Huang, and S. D. Blostein. Least-squares fitting of two 3-D point sets. *IEEE Trans Pattern Anal. Machine Intell.*, Vol. 9, pp. 698-700, 1987.
3. Z. Aung, W. Fu and K. Tan. An efficient index-based protein structure database searching method. *Proc. Intl. Conf. on Database Systems for Advanced Applications*, pp. 311-318, 2003.
4. H. M. Berman, J. Westbrook, Z. Feng, et al. The protein data bank. *Nucl. Acids Res.*, Vol. 28, pp. 235-242, 2000.

5. O. Çamoğlu, T. Kahveci and A. Singh. Towards index-based similarity search for protein structure databases. *IEEE Computer Society Bioinformatics Conference*, pp. 148-158, 2003.
6. T. Can and Y. Wang. CTSS: a robust and efficient method for protein structure alignment based on local geometrical and biological features. *IEEE Computer Society Bioinformatics Conference*, pp. 169-179, 2003.
7. L. P. Chew, D. Huttenlocher, K. Kedem and J. Kleinberg. Fast detection of common geometric substructure in proteins. *J. Comput. Biol.*, Vol. 6, No. 3, pp. 313-325, 1999.
8. I. Choi, J. Kwon and S. Kim. Local feature frequency profile: A method to measure structural similarity in proteins. *Proc. Natl. Acad. Sci.*, Vol. 101, No. 11, pp. 3797-3802, 2004.
9. D. W. Eggert, A. Lorusso and R. B. Fisher. Estimating 3-D rigid body transformations: a comparison of four major algorithms. *Machine Vision and Applications*, Vol. 9, pp. 272-290, 1997.
10. I. Eidhammer, I. Jonassen, and W. R. Taylor. Structure Comparison and Structure Patterns. *J. Computational Biology*, Vol. 7, No. 5, pp. 685-716, 2000.
11. M. Farach. Optimal suffix tree construction with large alphabets. *Proc. 38th IEEE Symp. Foundations of Computer Science,* pp. 137-143, 1997.
12. F. Gao and M. J. Zaki. PSIST: Indexing Protein Structures using Suffix Trees. *Proc. IEEE Computational Systems Bioinformatics Conference (CSB)*, pp. 212-222, 2005.
13. G. H. Golub and C. F. Van Loan. *Matrix Computation.* 3rd eds., John Hopkins University Press, 1996.
14. D. Gusfield. *Algorithms on strings, trees, and sequences: computer science and computational biology,* Cambridge University Press, 1997.
15. K. Kedem, P. Chew and R. Elber. Unit-vector RMS (URMS) as a tool to analyze molecular dynamics trajectories. *Proteins: Struct. Funct. Genet.*, Vol. 38, pp. 1-12, 1999.
16. E. M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM.*, Vol. 23, pp. 262-272, 1976.
17. J. T. Schwartz and M. Sharir. Identification of partially obscured objects in two and three dimensions by matching noisy characteristic curves. *Intl. J. of Robotics Res.*, Vol. 6, pp. 29-44, 1987.
18. T. Shibuya. Generalization of a suffix tree for RNA structural pattern matching. *Algorithmica*, Vol. 39, No. 1, pp. 1-19, 2004.
19. E. Ukkonen. On-line construction of suffix-trees. *Algorithmica,* Vol. 14, pp. 249-260, 1995.
20. P. Weiner. Linear pattern matching algorithms. *Proc. 14th Symposium on Switching and Automata Theory*, pp. 1-11, 1973.

New Bounds for Motif Finding in Strong Instances

Broňa Brejová, Daniel G. Brown, Ian M. Harrower, and Tomáš Vinař

David R. Cheriton School of Computer Science, University of Waterloo
{bbrejova, browndg, imharrow, tvinar}@cs.uwaterloo.ca

Abstract. Many algorithms for motif finding that are commonly used in bioinformatics start by sampling r potential motif occurrences from n input sequences. The motif is derived from these samples and evaluated on all sequences. This approach works extremely well in practice, and is implemented by several programs. Li, Ma and Wang have shown that a simple algorithm of this sort is a polynomial-time approximation scheme. However, in 2005, we showed specific instances of the motif finding problem for which the approximation ratio of a slight variation of this scheme converges to one very slowly as a function of the sample size r, which seemingly contradicts the high performance of sample-based algorithms. Here, we account for the difference by showing that, for a variety of different definitions of "strong" binary motifs, the approximation ratio of sample-based algorithms converges to one exponentially fast in r. We also describe "very strong" motifs, for which the simple sample-based approach always identifies the correct motif, even for modest values of r.

1 Introduction

Motif finding is a combinatorial abstraction of the very important problem of regulatory sequence detection in bioinformatics. In motif finding, n discrete input sequences, each of length m, are given, as is a parameter L, called the motif length. The most common goal is to find a contiguous substring of length L in each input sequence, minimizing some function of these substrings (called *the motif occurrences*). One objective function is found in the Consensus-Pattern problem:

Definition 1 (Consensus-Pattern). *Given are n sequences, $s_1, \ldots, s_n$, each of length m, over a finite alphabet Σ, and a parameter L. Find a contiguous subsequence x_i of length L from each sequence, and a consensus sequence x of these subsequences, minimizing $\sum_{i=1\ldots n} d_H(x_i, x)$, where $d_H(x, y)$ is the Hamming distance between two strings.*

While this problem is NP-hard, there is a simple sample-based polynomial-time approximation scheme for it. For a given value of r, the algorithm considers all samples of r substrings of length L from the n sequences. A motif derived from each such sample is then evaluated on all sequences, and the best motif is chosen. This algorithm was shown by Li *et al.* [4] to have approximation ratio of $1 + O(1/\sqrt{r})$ for constant-size alphabets. The algorithm also has $O(L(nm)^{r+1})$ runtime, which is polynomial if r is a constant.

M. Lewenstein and G. Valiente (Eds.): CPM 2006, LNCS 4009, pp. 94–105, 2006.

This bound is not especially useful, since the approximation ratio converges to one only very slowly with increasing r. Yet, sample-based algorithms with small values of r are very successful in practice for both motif finding in the abstract and for regulatory sequence detection [2, 5, 6, 9]. One might imagine that the bounds shown by Li *et al.* are weak, and the simple PTAS actually has a much stronger guarantee. However, in 2005 we showed [1] that this is very likely not the case. For a simple variation of the Li *et al.* PTAS (where the only difference is whether the sampling is without replacement or with replacement), we identified a collection of instances of the problem for which the approximation ratio is $1 + \Theta(1/\sqrt{r})$, suggesting that in order to achieve an approximation ratio of $1 + \varepsilon$, one needs a sample size of $r = \Theta(1/\varepsilon^2)$, which is highly impractical.

Still, the instances of Consensus-Pattern for which we proved our previous bounds are very weak motifs. They are binary instances of Consensus-Pattern; in each position of the motif instances, just over half of the entries are the symbol zero, and just under half are the symbol one. Such motifs are likely uninteresting, as they are no stronger than what we might expect to find if we considered random binary noise.

We might prefer to consider motifs bounded away from uniform noise. For such "strong" motifs, we can do much better: we show here that for various definitions of strong motifs, the approximation ratio of the algorithm approaches one exponentially fast as a function of r. In particular, for strong motifs, the approximation ratio is at most $1 + O(f^{-r})$, for a function f that depends only on the strength of the motif, instead of the approximation ratio of $1 + \Theta(1/\sqrt{r})$ shown for the general case.

Here, we show such theorems for a variety of different definitions of "strong" motifs. First, we consider binary motifs where at least a $\frac{1}{2} + \varepsilon$ fraction of all positions in the motif occurrences matches a given consensus of length L. While occasionally, the sampling PTAS can have bad performance on such an instance, we prove that for randomly chosen instances, the expected approximation ratio converges to one exponentially fast as a function of the sample size r. If we instead require consistently strong binary motifs, where each position of the motif has at least $(\frac{1}{2} + \varepsilon)n$ matches in the motif occurrences, we can prove that the PTAS performs well even in the worst case. In fact, for very strong consistent motifs, the PTAS will always find the correct answer, even for small r.

Our results document that while for arbitrary instances of motif finding, the simple sample-based PTAS may have poor convergence properties, for the kinds of motifs that people care about, the approximation ratio converges exponentially fast to the correct answer.

2 Background

The Consensus-Pattern problem, for an alphabet Σ, can be answered by enumerating all $|\Sigma|^L$ possible choices of the consensus pattern, and finding the best matches to each possible pattern, but such an enumeration is not efficient. Instead, one type of efficient heuristic for this problem first enumerates a polynomial number of candidate consensus patterns, and then finds the best match to each candidate in each of the n sequences, in $O(nmL)$ time per candidate.

One set of candidates is all L-letter substrings of the input strings; there are $(m-L+1)n$ of them, yielding an algorithm with $O(L(nm)^2)$ runtime. Or, we can expand this idea to consider the result of looking at a sample of r substrings of the input. For each such sample, we compute a candidate motif as a consensus of the sample by identifying the most common letter at each position of the motif, breaking ties arbitrarily.

In this paper, we consider two algorithms based on this idea. The first uses samples with replacement, implying that a single substring can occur in the sample multiple times. There are $((m-L+1)n)^r$ such samples; if we try all of them, this yields an algorithm with $O(L(nm)^{r+1})$ runtime. We will refer to this as the PTAS algorithm. Li *et al.* [4] have shown that this simple algorithm is indeed a polynomial-time approximation scheme (a PTAS): the approximation ratio of the algorithm converges to one as the sample size r grows. Unfortunately, the convergence rate they could prove is very slow: they show the approximation ratio is at most $1 + \frac{4|\Sigma|-4}{\sqrt{e}(\sqrt{4r+1}-3)}$.

We will also study a slight modification of the PTAS, in which we consider only samples without replacement. We will refer to it as the SWOR algorithm, for "sampling without replacement". In our previous work [1], we gave specific instances of the problem for which the approximation ratio of the SWOR algorithm is $1 + \Theta(1/\sqrt{r})$ as a function of r. We conjectured that the same lower bound also holds for PTAS, which asymptotically matches the upper bound of Li *et al.*

2.1 Notation and Observations

To simplify our analysis, we will always assume that the input sequences $s_1, \ldots, s_n$ consist solely of the optimal motif occurrences, that is, $m = L$. While Consensus-Pattern is trivial in these cases, since the optimal motif is the consensus string of the input sequences, both PTAS and SWOR are still well-defined and may not always optimize the objective function. In fact, we showed in our earlier work [1] that if one of these algorithms is run on just the motif occurrences themselves, it will do no better than if run on longer sequences. Upper and lower bounds on the approximation ratio that we show for such instances are still applicable to longer sequences.

We will assume that the sequence alphabet Σ is the set $\{0, 1\}$; all of our results here are for binary motifs. If $m = L$, we can always transform the instance of the problem so that the optimal motif is the string 0^L, by relabelling characters in each column that has more ones than zeros. We will use this transformation in some of our results.

Finally, we note that since PTAS always explores more samples than SWOR, its approximation ratio is always at least as good as that of SWOR. Therefore, any upper bound for the approximation ratio of SWOR also applies to PTAS.

2.2 Concentration Bounds

Most of our bounds are obtained by applying the Hoeffding bound [3], which gives concentration bounds on the sum of independent random variables, and an extension of it to certain classes of dependent variables due to Panconesi and Srinivasan [8]. In this section, we summarize the probabilistic bounds we use. We begin with the following variant of the Hoeffding bound from McDiarmid's survey [7, p. 199]; a similar bound can be found in [3, Theorem 1].

Theorem 1 (Hoeffding's bound [7]). *Let $X_1, \ldots, X_n$ be independent random variables, with $0 \leq X_k \leq 1$ for each k. Let $X = \sum X_k$, let $\mu = E[X]$, let $p = \mu/n$ and let $q = 1 - p$. Then for any $0 \leq t < q$,*

$$\Pr[X - \mu \geq nt] \leq \left(\left(\frac{p}{p+t} \right)^{p+t} \left(\frac{q}{q-t} \right)^{q-t} \right)^n .$$

Panconesi and Srinivasan [8] have extended the Hoeffding bound to sums of dependent variables that satisfy certain conditions.

Theorem 2. *Let $X_1, \ldots, X_n$ be (not necessarily independent) binary random variables with $\Pr[X_k = 1] = p$ for each k. If for every subset A of $\{1, \ldots, n\}$ and for every $k \notin A$,*

$$\Pr\left[X_k = 1 \,\middle|\, \bigwedge_{j \in A} (X_j = 1) \right] \leq \Pr[X_k = 1], \tag{1}$$

then Hoeffding's bound from Theorem 1 also holds for $X = \sum X_k$.

Proof. This is an application of Panconesi and Srinivasan's framework [8] for Chernoff-Hoeffding bounds of sums of dependent variables. Binary variables satisfying equation (1) are 1-correlated in the notation of Panconesi and Srinivasan. For such variables, we can apply the Hoeffding bounds directly, as though the variables were independent.

In particular, let $\hat{X}_1, \ldots, \hat{X}_n$ be independent random variables with $\Pr[X_k = 1] = p$. The variables $X = \sum_k X_k$ and $\hat{X} = \sum_k \hat{X}_k$ have the same expectation, $\mu = np$, and equation (1) implies that $\Pr\left[\wedge_{j \in A}(X_j = 1)\right] \leq \prod_{j \in A} \Pr(\hat{X}_i = 1)$. Thus, these random variables satisfy the conditions of Theorem 3.2 in [8], and we obtain

$$\Pr[X - \mu \geq \varepsilon\mu] \leq \frac{E[e^{h\hat{X}}]}{e^{h(1+\varepsilon)\mu}},$$

where ε and h are positive real numbers. As in the proof of Hoeffding's bound in McDiarmid [7, p. 199], we can prove $E[e^{h\hat{X}}] \leq (1 - p + pe^h)^n$. By substituting $\varepsilon = t/p$, we obtain

$$\Pr[X - \mu \geq tn] \leq \left(e^{-h(p+t)} (1 - p + pe^h) \right)^n ;$$

setting e^h to $\frac{(p+t)(1-p)}{p(1-p-t)}$, we obtain the desired result. □

Note that independent variables satisfy equation (1) with equality, so Theorem 1 is a special case of Theorem 2.

We will consider dependent binary random variables that are zero with some probability $p > 0.5$ and we will be interested in the probability that fewer than yn of the random variables are zero for some $0.5 \leq y < p$. Theorem 1 can be easily applied to this case, as is shown in the following lemma.

Lemma 1. *Let $X_1, \ldots, X_n$ be binary random variables with $\Pr[X_k = 0] = p$ for each k, where $p \geq 0.5$. If these variables satisfy the condition of Theorem 2, and $1 - p \leq y < p$ then $\Pr[\sum_k X_k \geq (1-y)n] \leq {\beta_y}^n$, where $\beta_y = \left(\frac{1-p}{1-y}\right)^{1-y} \left(\frac{p}{y}\right)^y$.*

Proof. The expectation of the variable $X = \sum_k X_k$ is $\mu = (1-p)n$. By Theorem 2, we easily obtain desired inequality:

$$\begin{aligned}\Pr[X \geq (1-y)n] &= \Pr[X - \mu \geq (p-y)n] \\ &\leq \left(\left(\frac{1-p}{1-p+(p-y)} \right)^{1-p+(p-y)} \left(\frac{p}{p-(p-y)} \right)^{p-(p-y)} \right)^n \\ &= {\beta_y}^n. \end{aligned}$$

□

Note that in the previous lemma, $\beta_y < 1$ for all p and y such that $0 < y < p < 1$. Therefore the probability that fewer than yn out of n variables are zeroes decreases exponentially as a function of n. For $y = 0.5$ we obtain the following special case.

Lemma 2. *Let $X_1, \ldots, X_n$ be binary random variables with $\Pr[X_k = 0] = p$ for each k, where $p \geq 0.5 + \varepsilon$. If these variables satisfy the condition of Theorem 2 then $\Pr[\sum_k X_k \geq n/2] \leq \alpha^n$, where $\alpha = \sqrt{4p(1-p)}$.*

3 Strong Motifs

We begin our analysis by considering motifs for which we know the number of zeros and ones in the motif instance. We do not necessarily fix the optimal motif to be 0^L.

Definition 2 (Strong motifs of fixed content). *A* strong motif of fixed content *p is a binary motif embedded into n sequences, where the total number of zeros in all n occurrences is pnL.*

Theorem 3. *For any value of r and $p > 0.5$, the worst-case approximation ratio of both PTAS and SWOR on strong motifs of fixed content at least p is the same as on arbitrary motifs.*

Proof. Consider the worst-case motif for a particular algorithm and value of r. Let p' be the number of zeros in this motif. If $p > p'$, we pad such an instance with enough columns, filled entirely with zeros, to make an instance of Consensus-Pattern that has at least pnL zeros. We have simply expanded the value of L.

The overall score of both the motif found by the algorithm and of the optimal motif is exactly the same as if we had not padded the instance with the extra columns. □

We have previously shown [1] that for any value of r, we can produce an instance of CONSENSUS-PATTERN for which SWOR has approximation ratio at least $1 + \Theta(1/\sqrt{r})$. This bound therefore transfers also to strong motifs of fixed content.

Thus, this definition of strong motifs does not give any better upper bound on the approximation ratio of the PTAS for motif finding than we had previously. The reason is that we allow many columns that are intensely weak and many columns that are very strong. In Section 4, we study motifs with more consistency among columns.

In the remainder of this section, we show that despite this negative result there are few bad instances of strong motifs, and if we choose a random strong motif of fixed content, the expected approximation ratio is much lower than in the worst case.

3.1 Randomly Chosen Strong Motifs

A *random motif of fixed content* p is an instance of the problem chosen uniformly from all $\binom{nL}{pnL}$ instances of the problem with exactly pnL zeros and $(1-p)nL$ ones. In such motifs, the zeros and ones may not be distributed uniformly, so some columns may contain more ones than zeroes. We call such columns *bad columns*; all other columns are *good columns.*

To analyze the expected approximation ratio of PTAS or SWOR on such randomly chosen motif, we divide all instances into *bad instances* and *good instances.* Bad instances have more than $L\alpha^{r/2}$ bad columns, where $\alpha = \sqrt{4p(1-p)}$. Lemma 3 shows that such instances are exponentially rare, and do not influence the expected approximation ratio much. Good instances have at most $L\alpha^{r/2}$ bad columns. In Lemma 4, we will show that for such instances, the approximation ratio is low.

Lemma 3. *The probability that a random binary motif of fixed content p is a bad instance is at most $\alpha^{r/2}$.*

Proof. Let $X_{i,j}$ be a binary random variable representing the symbol in row i and column j of the motif instance. For a given column j, let the number of ones be $X_j = \sum_i X_{i,j}$. Column j is bad if X_j is more than $n/2$. Each one in a column reduces the probability of others, so the variables corresponding to this column satisfy the conditions of Lemma 2, so $\Pr[X_j > n/2] \leq \alpha^n$. Since $n \geq r$, this probability is also at most α^r. By linearity of expectation, the expected number of bad columns is at most $L\alpha^r$.

Since a bad motif contains more than $L\alpha^{r/2}$ bad columns, we are bounding the probability that the number of bad columns is more than $\alpha^{-r/2}$ times its mean. This can be no greater than $1/\alpha^{-r/2}$, by the Markov inequality. □

Lemma 4. *The expected cost of a motif returned by PTAS (or SWOR) on a randomly chosen good instance is less than $nL\left(\frac{1-p+2p\alpha^r}{1-\alpha^{r/2}}\right)$.*

Proof. Let X_j be a random variable representing the number of ones in column j. Consider a random sample without replacement of r rows. Let Y_j be the number of ones in column j of this sample. The consensus of the sample is one when $Y_j \geq r/2$. Finally, let A_j be the score of the consensus character of this random sample in column j.

We want to bound $E[A_j|G]$, where G is the event that the motif instance is good. Since A_j is always non-negative, this conditional expectation is at most $E[A_j]/\Pr[G]$. In Lemma 3 we have shown that $\Pr[G] \geq 1 - \alpha^{r/2}$.

Consider the event Z_j that column j is good and the random sample has consensus zero. As for Lemma 3, Lemma 2 gives that $\Pr[X_j > n/2] \leq \alpha^r$, and $\Pr[Y_j \geq r/2] \leq \alpha^r$. Therefore the probability of Z_j is at least $1 - 2\alpha^r$.

In the case of the event Z_j, we are skewed towards having more zeroes than expected, and therefore

$$E[A_j|Z_j] = E[X_j|Z_j] \leq E[X_j] = n(1-p).$$

If we are not in Z_j, the cost of the column is at most n. Therefore, the expected cost of a single column is

$$\begin{aligned} E[A_j|G] &\leq \frac{\Pr[Z_j] \cdot E[A_j|Z_j] + \Pr[\bar{Z}_j] \cdot E[A_j|\bar{Z}_j]}{\Pr[G]} \\ &\leq \frac{(1-2\alpha^r)n(1-p) + 2\alpha^r n}{1-\alpha^{r/2}} = n \cdot \frac{1-p+2p\alpha^r}{1-\alpha^{r/2}} \end{aligned}$$

By linearity of expectation, the expected cost over all columns is at most $L \cdot E[A_j|G]$, and at least one sample in the SWOR algorithm must give us a motif which has at most this cost. □

With this in mind, we can bound the performance of the PTAS for random motifs of fixed content p.

Theorem 4. *When applied to a random motif of fixed content* $p > 0.5 + \varepsilon$*, both PTAS and SWOR have expected approximation ratio at most* $1 + \alpha^{r/2} \cdot \frac{18-16p+2p^2}{1-p^2}$ *for sufficiently large* r*, where* $\alpha = \sqrt{4p(1-p)}$*.*

Proof. For sufficiently large r, $\alpha^{r/2}$ is at most $(1-p)/2$. For good instances, the number of ones in good columns is at least $Ln(1 - p - \alpha^{r/2})$, which gives a non-negative lower bound on the optimal motif cost. Therefore, according to Lemma 4, the approximation ratio for such instances can be bounded by $\frac{1-p+2p\alpha^r}{(1-\alpha^{r/2})(1-p-\alpha^{r/2})}$.

We have previously shown [1] that any sampling algorithm has approximation ratio no more than 2 on all instances. We will use this upper bound for bad instances. This gives an overall bound of no greater than $\frac{1-p+2p\alpha^r}{(1-\alpha^{r/2})(1-p-\alpha^{r/2})} + 2\alpha^{r/2}$. Rearranging, we obtain the upper bound $1 + \alpha^{r/2} \cdot \frac{4-3p-5\alpha^{r/2}+4\alpha^{r/2}p+2\alpha^r}{(1-\alpha^{r/2})(1-p-\alpha^{r/2})}$. For sufficiently large r, $\alpha^{r/2} \leq (1-p)/2$, and we obtain the desired bound. □

This theorem shows that if either PTAS or SWOR is applied to a random strong motif of fixed content $p > 0.5$, the expected approximation ratio is $1 + O(\alpha^{r/2})$. This bound converges exponentially quickly to one as a function of r.

As a function of p, the bound on the approximation ratio is decreasing, as long as $p > 0.5$ and $r \geq 4$. This property will be important in the next section.

3.2 Strong Motifs of Fixed Expected Content

Perhaps more natural as a model of random motifs is the case where each of the L positions in all n sequences is chosen independently of all others, with probability p.

Definition 3 (Strong motif of fixed expected content). *A* strong motif *of fixed expected content p is a random motif where each position is zero with probability p, and one with probability $1 - p$ independently of other positions.*

This stochastic model can generate bad instances of the problem again, but it is very rare that such instances occur, and we can again always bound their approximation ratio by 2, so their contribution to the expected approximation ratio is small.

Theorem 5. *For strong binary motifs of expected content $p > 0.5$, where p is a fixed constant, the expected approximation ratio of the PTAS and SWOR is at most $1 + O(\gamma^r)$, for some constant $\gamma < 1$ that depends on p, but not r.*

Proof. Let $q = 1/4 + p/2$. According to Lemma 1, for a strong motif of expected content $p > 1/2$, the probability that the actual motif generated has fewer than qnL zeros is less than ${\beta_q}^{nL}$, where β_q is less than one. This is certainly less than ${\beta_q}^r$, since $nL \geq r$. For these weak motifs, we use the upper bound of 2 on the approximation ratio.

The remaining instances are strong motifs of content at least q. We can treat the process as first picking the motif content $\pi \geq q$, then picking a random motif of that fixed content. For a fixed content π, we can apply the bound given in Theorem 4. Since this bound decreases with increased strength of the motif, we can use the upper bound obtained with Theorem 4 for content q for all values of π. Therefore the overall approximation ratio of the algorithm is at most $1 + {\alpha_q}^{r/2} \cdot \frac{18-16q+2q^2}{1-q^2} + 2{\beta_q}^{r/2}$, $\alpha_q = \sqrt{4q(1-q)}$, for sufficiently large r, and by setting $\gamma = \sqrt{\max\{\alpha_q, \beta_q\}}$, we obtain the desired bound. □

3.3 Many Motifs Are Weak

We finish this section by noting that for any value of r, we can pick an instance size n for which in fact most motifs are weak, and for which we conjecture that the PTAS has poor convergence. If we let $p = 0.5$, and sample from the distribution of all binary motifs with expected content p, then all motif instances are equiprobable, so theorizing about the common behaviour of the algorithm also applies to common motif instances. A random motif of this content with

$n = r^2$ sequences is expected to have a constant fraction of columns in which the fraction of zeros in the column is between $1/2 + 1/\sqrt{r}$ and $1/2 + 2/\sqrt{r}$; that is, a significant fraction of the columns will be weak to the point where a random sample without replacement of r motif instances has a constant probability of picking the incorrect symbol for that column. Further, the expected cost of a random motif will be of the order of $(1/2 + \Theta(1/\sqrt{r}))nL$.

A random sample (without replacement) will incorrectly assign the symbols in a constant fraction of the motif's columns, giving an expected cost increase on the order of $\Theta(1/\sqrt{r})nL$ over the optimum, and an overall approximation ratio of $1 + \Omega(1/\sqrt{r})$. We conjecture that this bound applies to all samples, and that the overall performance of the best sample is also $1 + \Omega(1/\sqrt{r})$.

4 Consistently Strong Motifs

In the previous section, we were not able to guarantee a good performance of the PTAS in the worst case. This was because some instances of strong motifs may have contained many columns with approximately the same number of zeros and ones. Here, we study the performance of the PTAS on consistently strong motifs, where each motif column has a large number of zeros in it.

Definition 4 (Consistently strong motif). *A* consistently strong motif of content $p > 0.5$ *is a binary motif embedded into n sequences, where each column of the motif has at least pn zeros.*

We first note the performance of the algorithms PTAS and SWOR on a single column of a consistently strong binary motif.

Lemma 5. *Suppose that we choose a random sample of r rows (with or without replacement) from a motif instance in which a particular column has pn zeros and $(1-p)n$ ones, for $p > 0.5$. The expected cost of the consensus character of the sample in this column is at most $n((1-p)(1-\alpha^r)+\alpha^r)$, where $\alpha = \sqrt{4p(1-p)}$.*

Proof. First, we want to bound the probability that the random sample without replacement has fewer zeroes than ones in this column, in which case the consensus character will be one. This situation satisfies conditions of Lemma 2 and the probability that at least half of the sample will be ones is at most α^r.

Note that for any constant $p > 0.5 + \varepsilon$, for some positive ε, this bound on the probability of erring in a single column converges to zero exponentially fast in r. Therefore, such samples will not have much influence on the expected cost, and we can bound their cost from above by n. The cost of a sample with consensus zero is exactly $n(1 - p)$. Therefore, the expected cost is at most $n[(1 - p)(1 - \alpha^r) + \alpha^r]$. □

Theorem 6. *For sufficiently large r, both PTAS and SWOR, applied to a consistently strong motif of content $p > 0.5$, have approximation ratio at most $1 + \alpha^r \cdot \frac{p}{1-p}$, where $\alpha = \sqrt{4p(1-p)}$.*

Proof. Let p_i be the content of zeros of the i-th column of the motif. According to Lemma 5, the expected cost $e(p_i)$ of the i-th column is at most $n((1-p_i)(1-{\alpha_{p_i}}^r)+{\alpha_{p_i}}^r)$, where $\alpha_{p_i} = \sqrt{4p_i(1-p_i)}$. The optimal cost of the same column is $o(p_i) = n(1-p_i)$. From the linearity of expectation, the expected approximation ratio of a consensus of a random sample of r rows over all columns of the motif is $R = \frac{\sum_{i=1}^{L} e(p_i)}{\sum_{i=1}^{L} o(p_i)}$.

Note, that for sufficiently large r (in particular, $r > 2/(2p-1)$), the function $e(p')/o(p')$ is decreasing with increasing value of p' for $p' \geq p$. Therefore, $e(p_i) \leq e(p)o(p_i)/o(p)$, and thus

$$R \leq e(p)/o(p) \leq \frac{n((1-p)(1-\alpha^r)+\alpha^r)}{n(1-p)} = 1 + \alpha^r \cdot \frac{p}{1-p}.$$

At least one sample must achieve this bound, by the first moment principle. Since SWOR examines all samples without replacement, the sample found by SWOR achieves the bound. □

If $p > 0.5 + \varepsilon$ for some constant $\varepsilon > 0$, then this ratio converges exponentially quickly to one.

4.1 Very Strong Consistent Motifs

We finish by noting that some motifs are so strong that the PTAS is guaranteed to find them exactly.

We saw in the proof of Lemma 5 that we can bound the probability of making an error for any column, when we sample r motif instances of that column. If the column has frequency p_i of zeros, the error probability was at most ${\alpha_{p_i}}^r$, where $\alpha_{p_i} = \sqrt{4p_i(1-p_i)}$.

If we have a motif whose columns are strong enough so that the sum of the ${\alpha_{p_i}}^r$ is at most one, the standard union bound gives that the probability that at least one sample column has more ones than zeros is less than 1. Thus, there must exist a sample of r motif instances whose consensus is exactly the correct L-letter-long motif. Since the PTAS is exhaustive, we will examine this sample, and it will be found by the algorithm.

In particular, a motif strong enough that the value of ${\alpha_{p_i}}^r$ is always less than $1/L$ will always be found by the PTAS.

Theorem 7. *The PTAS always finds the correct motif when its input is a consistently strong binary motif of length L with probability $p \geq \frac{1}{2} + \frac{\sqrt{1-L^{-2/r}}}{2}$.*

Proof. This is shown by noting that $\frac{1}{2} + \frac{\sqrt{1-L^{-2/r}}}{2}$ is the root in the range $(0.5, 1]$ of $(4p(1-p))^{r/2} = 1/L$, corresponding to the value where α_p goes below $1/L$. □

This value quickly shrinks for values of r that are not especially large: for a length 10 binary motif, if all columns are at least 80% zeroes, examining all samples of size 11 is certain to find the true motif, while samples of size 5 are all that is needed for motifs of that length where $p = 0.9$. Indeed, for a fixed value of L, if the motif is consistently strong with probability at least $0.5 + f(r)$, where $f(r)$ is a specific function that is only $O(1/\sqrt{r})$, the PTAS will find the optimal motif.

For random motifs, the situation is not as good; obviously, a random motif with probability p might turn out not to be strong. But, for p large enough, the probability of producing a motif that is weak enough that the algorithm has positive probability of failing can easily be estimated, and again converges exponentially rapidly to zero as a function of p or of r, for a fixed motif length L.

5 Conclusion and Open Problems

We have shown a variety of characterizations of "strong" binary instances of Consensus-Pattern for which the simple sampling-based polynomial-time approximation scheme of Li *et al* [4] has an approximation ratio guarantee that converges to one exponentially fast as a function of r, the sample size. This result is in contrast with our previous work, which showed specific instances of Consensus-Pattern for which a variation of the Li *et al.* PTAS can only achieve $1 + \Theta(1/\sqrt{r})$ approximation ratio.

The difference is quite significant; to achieve $1 + \varepsilon$ approximation ratio using the general bound requires samples of size $\Omega(1/\varepsilon^2)$, giving runtimes of $O(L(nm)^{\Omega(1/\varepsilon^2)})$, whereas for strong motifs we show that a sample size of $O(\log(1/\varepsilon))$ is sufficient.

Our bounds apply to random binary motifs of specific strength, or to those for which the probability that any specific position is a zero is fixed to be a constant bounded above 0.5. While it is possible to obtain a difficult-to-solve instance of the problem by chance, such instances are exponentially rare, and as such, do not affect the algorithm's behaviour significantly.

Finally, we show that for strong instances, small samples can guarantee that the motif found is optimal. While the bounds achieved are not practical, this again suggests that motif finding is an easy problem when applied to strong instances, and only hard when applied to irrelevant, weak problem instances.

Open problems. How tight are the bounds for very strong consistent motifs given in Section 4.1? Can we find specific strong instances of Consensus-Pattern for which the sample-based PTAS finds a wrong motif and for which the value of r is close to the one shown in the theorem, or is the bound very loose?

In all our theorems, we have considered only binary alphabet. Our results extend to non-binary alphabets; however, we still require that one of the alphabet symbols has frequency more than 0.5. The problem of regulatory sequence detection is most commonly applied to DNA and protein sequences, and it makes sense to consider instances in which the most common letter in each column is significantly more common than other letters, but still does not achieve frequency more than 0.5. To prove exponential convergence for such instances likely requires a variation on the Chernoff-Hoeffding bounds for multi-outcome variables.

Acknowledgements

All authors are supported by the Natural Science and Engineering Research Council of Canada. We would like to thank Nick Wormald for a helpful conversation and for pointing us to the paper of McDiarmid [7].

References

1. B. Brejova, D.G. Brown, I.M. Harrower, A. Lopez-Ortiz, and T. Vinar. Sharper upper and lower bounds for an approximation scheme for Consensus-Pattern. In A. Apostolico, M. Crochemore, and K. Park, editors, *Combinatorial Pattern Matching, 16th Annual Symposium (CPM 2005)*, pages 1–10, 2005.
2. G.Z. Hertz and G.D. Stormo. Identifying DNA and protein patterns with statistically significant alignments of multiple sequences. *Bioinformatics*, 15(7-8):563–577, 1999.
3. W.J. Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58:713–721, 1963.
4. M. Li, B. Ma, and L. Wang. Finding similar regions in many strings. *Journal of Computer and System Sciences*, 65(1):73–96, 2002.
5. C. Liang. COPIA: a new software for finding consensus patterns in unaligned protein sequences. Master's thesis, University of Waterloo, October 2001.
6. J. Liu. A combinatorial approach for motif discovery in unaligned DNA sequences. Master's thesis, University of Waterloo, March 2004.
7. C. McDiarmid. Concentration. In M. Habib, editor, *Probabilistic methods for algorithmic discrete mathematics*, pages 195–248. Springer, 1998.
8. A. Panconesi and A. Srinivasan. Randomized distributed edge coloring via an extension of the Chernoff-Hoeffding bounds. *SIAM Journal on Computing*, 26:350–368, 1997.
9. P.A. Pevzner and S. Sze. Combinatorial approaches to finding subtle signals in DNA sequences. In *Proceedings of the 8th International Conference on Intelligent Systems for Molecular Biology (ISMB 2000)*, pages 269–278, 2000.

Fingerprint Clustering with Bounded Number of Missing Values

Paola Bonizzoni[1], Gianluca Della Vedova[2],
Riccardo Dondi[3], and Giancarlo Mauri[1]

[1] DISCo, Università degli Studi di Milano-Bicocca, Milano - Italy
[2] Dip. Statistica, Università degli Studi di Milano-Bicocca, Milano - Italy
[3] Dipartimento di Scienze dei Linguaggi, della Comunicazione e degli Studi Culturali,
Università degli Studi di Bergamo, Bergamo - Italy
bonizzoni@disco.unimib.it, gianluca.dellavedova@unimib.it,
riccardo.dondi@unibg.it, mauri@disco.unimib.it

Abstract. The problem of clustering fingerprint vectors with missing values is an interesting problem in Computational Biology that has been proposed in [6]. In this paper we show some improvements in closing the gaps between the known lower bounds and upper bounds on the approximability of variants of the biological problem. Moreover, we have studied two additional variants of the original problem. We prove that all such problems are APX-hard even when each fingerprint contains only two unknown positions and we present a greedy algorithm that has constant approximation factors for these variants. Despite the hardness of these restricted versions of the problem, we show that the general clustering problem on an unbounded number of missing values such that they occur for every fixed position of an input vector in at most one fingerprint is polynomial time solvable.

1 Introduction

High-throughput approaches for the examination of microbial communities are becoming increasingly important, especially after the oligonucleotide fingerprinting strategy has found wide application, allowing the identification of thousands of cDNA clones [3, 4, 5, 8, 9]. After the rDNA clone libraries are constructed, the clones are classified by individual hybridization experiments on DNA microarrays with a series of short DNA oligonucleotides into *clone types* or *operational taxonomic units* (OTUs), where an OTU is a set of DNA clones sharing the same set of oligonucleotides that have successfully hybridized. Once classified, the nucleotide sequence of representative clones from each OTU can then be obtained by DNA sequencing to provide phylogenetic descriptions of the microorganisms. One of the key features of this strategy is that after a comprehensive database correlating hybridization patterns with nucleotide sequence data has been compiled, little additional rDNA clone sequencing will be required, resulting in significant reduction of cost and effort. The effectiveness of this general strategy has been demonstrated in the biotechnology arena, where it is currently being used to screen and identify millions of cDNA clones [3].

M. Lewenstein and G. Valiente (Eds.): CPM 2006, LNCS 4009, pp. 106–116, 2006.

The oligonucleotide fingerprinting method is commonly used to study DNA clone libraries. Such method naturally leads to a combinatorial problem where for each oligonucleotide we are given a fingerprint over the alphabet $\{0, 1, N\}$, where the values 0 or 1 represent respectively that a hybridization has happened or not with a certain clone, while the value N stands for the fact that we are unable to determine if the hybridization has happened or not (typically this is due to the fact that there are two control signals, and the values between those two control signals suggest that both of the two possible outcomes are equally likely to have happened).

Some combinatorial problems naturally arise, most notably the Clustering with Missing Values (CMV) problem. An instance of CMV (and of all problems studied in this paper) is a set F of n vectors with values in $\{0, 1, N\}$, called fingerprint vectors (in short *fingerprint*); in all instances of the problems that we will study, all fingerprints have the same length l, that is they all contain exactly l elements. Two fingerprint vectors $f_1 = \langle f_1[1], f_1[2], \ldots, f_1[l]\rangle$ and $f_2 = \langle f_2[1], \ldots, f_2[l]\rangle$ are *compatible* if for any position i where they differ, one of $f_1[i]$ and $f_2[i]$ is equal to N. A resolved vector $r = \langle r[1], \ldots, r[l]\rangle$ of a fingerprint vector $f = \langle f[1], \ldots, f[l]\rangle$ is a vector over alphabet $\{0, 1\}$ such that for each $1 \leq i \leq l$, if $f[i] \neq N$ then $f[i] = r[i]$. If a resolved vector r and a fingerprint f are compatible, r is said to be a *resolution* of f or to resolve f, this means that r is obtained by replacing each occurrence of N in f with 0 or 1. The expected output is a partition P of F, $P = \{P_1, \ldots, P_k\}$, such that in each set P_i of P there are only pairwise compatible fingerprints.

We will also analyze the effect of a parameter, the maximum number of Ns allowed in a fingerprint, and we will denote by p such parameter. As already stated above, an instance of CMV consists of a set F of fingerprints, and we would like to find a minimum-size partition of F where each pair of fingerprints in a set are compatible. Equivalently we want to find a minimum-size set R of resolved fingerprints such that each input fingerprint is resolved by some fingerprint in R.

Unfortunately the problem is NP-hard [6], therefore it is important to find some restrictions under which the problem becomes tractable. For instance it is possible to restrict the problem to instances where each input fingerprint contains at most p Ns, and we will call such problem CMV(p). It is already known that CMV(2) is NP-hard[7], while CMV(1) can be solved in polynomial-time[6], so for all interesting values of p we have to concentrate on developing approximation algorithms. CMV(p) is known to be approximable within factor 2^p [6] and $\min(1+\ln n, 2+p\ln l)$[7]. In this paper we strengthen the NP-hardness result proving that CMV(2) is APX-hard, that is it cannot be approximated within an arbitrarily small $(1+\epsilon)$-approximation polynomial-time algorithm unless P=NP [2].

Moreover, we will study two related optimization problems introduced in [7], namely: Inside Clustering with p missing values (IEC(p)), where we want to find a partition P maximizing the sum $\sum_{i=1}^{k} \binom{|P_i|}{2}$; Outside Clustering with p missing values (OEC(p)), where we want to find a partition P minimizing the number of compatible pairs of fingerprints belonging to different sets of P.

Notice that, as observed in [7], an exact solution to IEC(p) is also an exact solution to OEC(p) and vice versa. For these two problems, we will present a fixed-parameter approximation algorithm whose running time is $O(2^p n^3 l)$, where p is the maximum number of Ns in a fingerprint and show that our algorithm achieves 2 and $\frac{1}{2}$-approximation factors for IEC and OEC respectively.

On the other side, we prove lower bounds on the approximability of both IEC(p) and OEC(p), showing that such problems are APX-hard. Finally, we show that the restriction of CMV to instances where, for each given position, missing values occur in at most one fingerprint vector, leads to a polynomial time solution.

2 A Fixed-Parameter Approximation Algorithm for IEC and OEC

In this section we present a fixed-parameter approximation algorithm for both IEC and OEC problems, where the parameter is the maximum number p of Ns appearing in a fingerprint. The algorithm we present has a time complexity $O(2^p n^3 l)$. We are able to provide two different analyses, one for each problem, showing that we achieve a 2-approximation ratio for IEC and a $\frac{1}{2}$-approximation ratio for OEC; the analysis for OEC is omitted due to space constraints.

Given a set F of fingerprints, in $O(2^p nl)$ time we are able to compute the set $R = \{r_1, \ldots, r_{|R|}\}$ of all possible resolved vectors that are compatible with at least one fingerprint in F. (Note that $|R| \leq 2^p n$.) Given a resolved vector r, we denote by $s(r, F)$ the set of fingerprints in F that are resolved by r. The *degree* of a resolved vector r, denoted by $d(r)$, is defined as $|s(r, F)|$. Since each resolved fingerprint is compatible with at most n fingerprints in F, computing all such sets $s(r, F)$ can be done in $O(|R|nl) = O(2^p n^2 l)$ time.

The algorithm constructs a partition Π of F greedily as follows: initially let Π be an empty partition and let U be equal to F. At each iteration the algorithm computes the sets R of resolved vectors of U and $s(r_i, U)$ for all $r_i \in R$. Then it finds the resolved vector r of maximum degree, adds $s(r, U)$ as a set of the solution Π and removes all fingerprints in $s(r, U)$ from U. The algorithm iterates until U is empty.

Notice that the algorithm computes a sequence $\langle r_1, \ldots, r_k \rangle$ of maximum degree resolved vectors, one at each iteration. At the i-th iteration the algorithm builds a set S_i of the solution Π containing all fingerprints that are compatible with r_i and that have not been assigned to a set of the solution during one of the previous iterations. For ease of analysis, we will denote by U_i the set of fingerprints that have not been assigned to a set of Π at the beginning of the i-th iteration. Consequently, $U_1 = F$, $U_{i+1} = U_i \setminus S_i$, for $1 \leq i < k$, where k is the number of sets in the final solution. Then, the algorithm computes the partition $\Pi = \{S_1, \ldots, S_k\}$. The optimal partition for both IEC and OEC is denoted by $Opt = \{O_1, \ldots, O_h\}$, where h can be different from k.

The analysis of the time complexity of the algorithm is simple, the running time at each iteration is dominated by the $O(2^p n^2 l)$ time required to compute the sets R and $s(r_i)$ for all $r_i \in R$, moreover at each iteration at least one fingerprint is removed from U, therefore at most n iterations are required, for an overall $O(2^p n^3 l)$ time complexity.

The *value* of the approximate solution Π is the number of pairs of compatible fingerprints co-clustered by Π and is denoted by $V(\Pi)$. More precisely, $V(\Pi) = \sum_{i=1}^{|\Pi|} |P(S_i)|$, where $P(S_i)$ is the set of distinct pairs of fingerprints in S_i. Generalizing such notion, we denote by $P(\Pi)$ the set of all the pairs co-clustered in the partition Π, that is $P(\Pi) = \cup_{i=1}^{|\Pi|} P(S_i)$. Let $W \subseteq U$ be a subset of fingerprints, we denote by $P(\Pi, W)$ the set of pairs (x, y) in $P(\Pi)$ such that at least one of x, y is in W.

By definition, the value of the optimal solution is $|P(Opt)|$; our goal will be to show that $|P(Opt)| \leq 2|P(\Pi)|$. We introduce some sets as follows: $P(Opt, 1) = P(Opt, S_1)$, and $P(Opt, i+1) = P(Opt, S_{i+1}) \setminus \bigcup_{1 \leq j \leq i} P(Opt, j)$ for $1 \leq i < k$. A fundamental property is that $\{P(Opt, i) : 1 \leq i < k\}$ is a partition of $P(Opt)$. Indeed, since $\Pi = \{S_1, \ldots, S_k\}$ is a partition of F, then $P(Opt) = \bigcup P(Opt, S_i)$. Let (x, y) be a pair of $P(Opt)$. W.l.o.g. we can assume that $x \in S_i$, $y \in S_j$, with $i \leq j$. Then $(x, y) \in P(Opt, S_i)$ and (x, y) does not belong to any $P(Opt, z)$ with $z > i$, therefore $(x, y) \in P(Opt, i)$. Consequently the sets $P(Opt, i)$ form a partition of $P(Opt)$, and the value of the optimal solution is equal to $\sum_i |P(Opt, i)|$.

In order to prove that $|P(Opt)| \leq 2|P(\Pi)|$, it suffices to show that $|P(Opt, i)| \leq 2|P(S_i)|$ for $1 \leq i \leq k$. In fact all pairs in $P(Opt, i)$ must belong to $U_i \times U_i$, by definition of $P(Opt, i)$. Each fingerprint x in U_i is in the same set of the optimal solution with at most $|S_i| - 1$ other fingerprints of U_i, otherwise the algorithm would not have chosen S_i at the i-th iteration as a maximum set of compatible fingerprints. By definition of $P(Opt, i)$, there are at most $|S_i|(|S_i| - 1)$ pairs of compatible fingerprints in $P(Opt, i)$, which completes the proof, since in S_i there are exactly $|S_i|(|S_i| - 1)/2$ pairs of compatible fingerprints.

3 APX-Hardness of CMV(2)

In this section we will prove that CMV(2) is APX-hard via an L-reduction from minimum vertex cover on cubic graphs (MVCC) (for details on L-reduction see [2]), which is known to be APX-hard [1]. In particular, we will combine two L-reductions: (1) from minimum vertex cover on a graph G to minimum vertex cover on a graph gadget CG; (2) from minimum vertex cover on a graph gadget CG to CMV(2).

First Reduction. Let $G = (V, E)$ be a cubic graph, the MVCC problem asks for the subset $V' \subseteq V$ of minimum cardinality, such that for each edge $(i, j) \in E$ at least one of i, j belongs to V'. Let $|V| = n$ and $|E| = m$. We reduce MVCC to minimum vertex cover on graph gadgets. Next we define the graph gadget associated with G. For each vertex $v_i \in V$ we define a vertex gadget VG_i consisting of 5 vertices c_{i_1}, c_{i_2}, c_{i_3}, c_{i_4}, c_{i_5} as in Fig. 1. Three vertices, c_{i_1}, c_{i_4}, c_{i_5} are called *docking vertices*. Observe that the minimum vertex cover of a vertex gadget

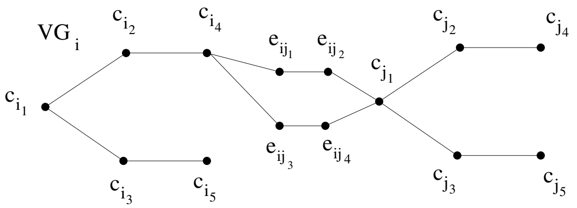


Fig. 1. Vertex gadgets VG_i and VG_j, edge gadget EG_{ij}

consists of 2 vertices, c_{i_2}, c_{i_3}, and we denote this vertex cover as *type 1.* Observe that there is a vertex cover of VG_i consisting of the 3 docking vertices c_{i_1}, c_{i_4}, c_{i_5}, and we denote this cover as *type 2.* For each edge (v_i, v_j) we define an edge gadget $EG_{i,j}$ joining vertex gadgets VG_i, VG_j in two of their docking vertices, such that each docking vertex is associated with exactly one edge of G. An important observation is that if C is a vertex cover of the graph gadget, then we can compute in polynomial time a vertex cover C' that is not larger than C, and such that C' consists only of vertex covers of type 1 and type 2 and more precisely, for each pair of adjacent vertex gadgets at least one has a vertex cover in C' of type 2.

Indeed, if a vertex gadget does not have a cover of type 1, then we can substitute this cover with one of type 2, obtaining a solution not larger than the original one.

Theorem 1. *There is a cover C of G of size k if and only if there is a cover C_G of the graph gadget of size $3k + 2(n - k) + 2m$.*

Second Reduction. Now we reduce minimum vertex cover on graph gadgets to CMV(2). The idea in our reduction is that it is possible to assign a resolved vector to each vertex and a fingerprint to each edge of a graph gadget CG. The instance of CMV(2) consists of the set of fingerprints F_{CG} associated with the graph gadget CG, and all interesting solutions will pick their resolved vectors from those assigned to the vertices. More precisely, we construct the set F_{CG} in such a way that each fingerprint assigned to an edge (x, y) will be resolved by one of the resolved vectors assigned to x or y.

Recall that n denote the number of vertex gadgets. Each fingerprint in F_{CG} consists of n blocks of 7 positions, and each resolved vector associated with a vertex in VG_i consists only of 0s block, except for the i-th block. Given vertex c_x, then r_x denotes the resolved vector associated with c_x while $r_x\langle i\rangle$ denotes the i-th block of r_x. Given the resolved vectors associated with the vertices of VG_i, we define the i-th block of each of such vectors as follows: $r_{i_1}\langle i\rangle = 1110000$, $r_{i_2}\langle i\rangle = 1111100$, $r_{i_3}\langle i\rangle = 1110011$, $r_{i_4}\langle i\rangle = 1001100$, $r_{i_5}\langle i\rangle = 1000011$. For example, the resolved vector r_{i_4} of the i-th vertex gadget c_{i_4} is $0^{7(i-1)}10011000^{7(n-i)}$.

The vertices belonging exclusively to an edge gadget will have two blocks that are not completely made of 0s. More precisely, let VG_i and VG_j be two adjacent vertex gadgets, then only the i-th and the j-th blocks of the resolved vectors $r(e_{i,j,1})$, $r(e_{i,j,2})$, $r(e_{i,j,3})$, $r(e_{i,j,4})$ associated with the vertices of the edge gadgets EG_{ij} do not completely consist of 0s.

Assume that $e_{i,j,1}$, $e_{i,j,3}$ are adjacent to a docking vertex of VG_i, c_{i_x}, and that $e_{i,j,2}$, $e_{i,j,4}$ are adjacent to a docking vertex of VG_j, c_{j_y}. Moreover, observe that each resolved vector associated with a docking vertex has exactly 3 positions set to 1. Let 1, x_1, x_2 be the positions in the i-th block of c_{i_x} set to 1, where $1 < x_1 < x_2 \leq 7$. Let 1, y_1, y_2 be the positions in the j-th block of c_{j_y} set to 1, where $1 < y_1 < y_2 \leq 7$. The i-th and j-th block of $r(e_{i,j,1})$, $r(e_{i,j,2})$, $r(e_{i,j,3})$, $r(e_{i,j,4})$ are defined as follows: $r(e_{i,j,1}\langle i\rangle)$ has value 1 in positions 1 and x_1, while $r(e_{i,j,1}\langle j\rangle)$ has value 1 in position y_1; $r(e_{i,j,3}\langle i\rangle)$ has value 1 in positions 1 and x_2, while $r(e_{i,j,3}\langle j\rangle)$ has value 1 in position y_2; $r(e_{i,j,2}\langle i\rangle)$ has value 1 in position x_1, while $r(e_{i,j,2}\langle j\rangle)$ has value 1 in positions 1 and y_1; $r(e_{i,j,4}\langle i\rangle)$ has value 1 in position x_2, while $r(e_{i,j,4}\langle j\rangle)$ has value 1 in positions 1 and y_2; any other position of i-th and j-th block is set to zero for $r(e_{i,j,1})$, $r(e_{i,j,2})$, $r(e_{i,j,3})$ and $r(e_{i,j,4})$. For any other position not in the i-th or j-th block, the resolved vectors $r(e_{i,j,1})$, $r(e_{i,j,2})$, $r(e_{i,j,3})$ and $r(e_{i,j,4})$, are set to 0. Denote with R the set of resolved vectors associated with vertices of the gadget graph.

Now we define the instance of the problem, that is the set of fingerprints F_{CG} associated with the edges of the graph gadget. Given $e = (x, y)$ an edge of the graph gadget and v_x, v_y the resolved vectors associated with vertices x and y, we associate with e the fingerprint f_e by using the following rule:

$f_e[t] := v_x[t]$ for each position t such that $v_x[t] = v_y[t]$, and $f_e[t] := N$ otherwise.

For example, let r_{i_1}, r_{i_2} be two resolved vectors associated with VG_i and recall that the i-th block of these vectors is $r_{i_1}\langle i\rangle = 1110000$, $r_{i_2}\langle i\rangle = 1111100$. Let e be the edge having c_{i_1} and c_{i_2} as endpoints, it follows that f_e, the fingerprint associated with e, has i-th block equal to $111NN00$, and all other blocks set to 0. Since two resolved vectors associated with an edge of the gadget graph have Hamming distance 2, each fingerprint in F_{CG} has exactly two positions with value N. A fundamental property of F_{CG} is the following:

Lemma 1. *Two fingerprints f_i, f_j in F_{CG} have a common resolution if and only if the edges of the gadget graph associated to such fingerprints share a common vertex v. The resolved vector associated to v is the only common resolution of f_i, f_j.*

Proof. Let us prove the only if part of the Lemma, as the other direction is immediate. Let f_i be a fingerprint encoding edge $e_i = (i_1, i_2)$ and let f_j be a fingerprint encoding edge $e_j = (j_1, j_2)$. There is at least one pair of resolved vectors in R associated with the endpoints of e_i and e_j having Hamming distance at least 4; assume w.l.o.g. those vectors are $r(i_1)$ and $r(j_1)$. Note that none of $r(i_1)$ and $r(j_1)$ can be a common resolution for both f_i and f_j. Any resolution r_i^* of f_i different from $r(i_1)$ and $r(i_2)$, has Hamming distance 1 from $r(i_1)$, hence r_i^* has Hamming distance at least 3 from $r(j_1)$, thus it can not be a resolution of f_j. Similarly, any resolution r_j^* of f_j different from $r(j_1)$ and $r(j_2)$ can not be a resolution of f_i. Hence f_i and f_j have a common resolution only if $r(i_2)$ and $r(j_2)$ are the same vector, that is they encode the same vertex. □

As a consequence of Lemma 1, if C is a vertex cover of the graph gadget, then we can define a solution S of CMV(2) over F_{CG} as the sets s_v of fingerprints resolved by a vector $r(v)$ associated with a vertex v in the cover, that is such a solution is of size $|C|$.

To prove the converse, let us consider a solution S for CMV(2) over instance F_{CG}. If a fingerprint is resolved by a vector v not associated with a vertex of the gadget graph, then this resolution is not shared by any other fingerprint of the instance. Thus, we can replace v with a resolved vector associated with a vertex of the graph, obtaining a solution S' for CMV(2) that has at most the same size of the solution S. Consequently, we can assume that the solution of CMV(2) consists only of sets associated with resolved vectors in R. By Lemma 1, it is immediate that the set of vertices associated to resolved vectors taken in the solution S' of CMV(2) over F_{CG} is a vertex cover of the gadget graph. By the above two observations, it follows that the graph gadget CG has a vertex cover of size k if and only if the instance F_{CG} of CMV(2) has a solution of size k. It is immediate to notice that both reductions in this section are actually L-reductions.

4 APX-Hardness of IEC(2)

In the following section we prove that IEC(2) is APX-hard via an L-reduction from Maximum Independent Set on Cubic Graphs (MIS), which is known to be APX-hard [1]. Let $G = (V, E)$ be a cubic graph, the MIS problem asks for the subset $V' \subseteq V$ of maximum cardinality, such that vertices in V' are not adjacent. Let $G = (V, E)$ be an instance of MIS, the reduction builds an instance F_G of IEC(2) associating with each vertex $v_i \in V$ a set of fingerprints F_i and with each $e = (v_i, v_j) \in E$ a fingerprint $f_{i,j}$.

More precisely, a set F_i of 9 fingerprints is associated with each vertex $v_i \in V$. Such fingerprints are constructed, similarly as in the reduction of Section 3, from a set of resolved vectors. Indeed, we define a set of 8 resolved vectors, $R_i = \{r_{i_j} | 1 \leq j \leq 8\}$, that are the possible resolutions of the fingerprints in F_i. We will show that all interesting solutions of IEC(2) over instance F_i will pick their resolved vectors from R_i. We introduce a graph, called *compatibility graph* and denoted as CG_i (see Fig. 2), such that vertices of CG_i are the resolved vectors in R_i, while each edge (r_{i_u}, r_{i_v}) of CG_i is associated to a fingerprint in F_i, that is compatible with the resolved vectors r_{i_u} and r_{i_v}.

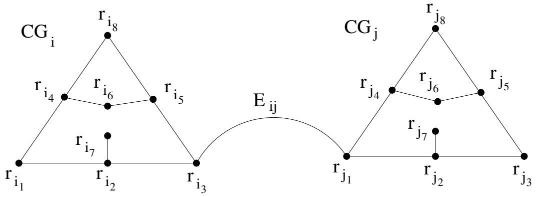


Fig. 2. Compatibility graphs CG_i and CG_j

Three vertices of CG_i, r_{i_1}, r_{i_3} and r_{i_8} are called *docking vertices* and are the resolved vectors that are compatible with fingerprints associated with edges of the instance graph G of MIS. More precisely, given edge (v_i, v_j) of G, a fingerprint denoted by $f_{i,j}$ is associated with (v_i, v_j) and it is represented by an edge $E_{i,j}$ incident on a docking vertex r_{i_x} of CG_i and docking vertex r_{j_y} of CG_j (see Fig. 2), respectively. The fingerprint $f_{i,j}$ is constructed as being compatible with r_{i_x} and r_{j_y}.

The graph consisting of all compatibility graphs CG_i and the edges $E_{i,j}$ is denoted as CG.

Given $G = (V, E)$ the instance of MIS, with $|V| = n$ and $|E| = m$, then set R_i is defined as follows. Each resolved vector consists of n blocks of 5 positions, where the t-th block of resolved vector r_x is denoted by $r_x\langle t\rangle$. Then, $r_{i_1}\langle i\rangle = 11000$, $r_{i_2}\langle i\rangle = 11010$, $r_{i_3}\langle i\rangle = 10010$, $r_{i_4}\langle i\rangle = 11100$, $r_{i_5}\langle i\rangle = 10110$, $r_{i_6}\langle i\rangle = 11110$, $r_{i_7}\langle i\rangle = 11011$, $r_{i_8}\langle i\rangle = 10100$. Assume w.l.o.g. that the vertex v_i is adjacent to v_j, v_h, v_k. Then each of the docking vertices of CG_i, r_{i_1}, r_{i_3} and r_{i_8}, is adjacent to a docking vertex of CG_j, CG_h and CG_k. More precisely, we assume w.l.o.g. that r_{i_1} is adjacent to a docking vertex of CG_j, r_{i_3} is adjacent to a docking vertex of CG_h, r_{i_8} is adjacent to a docking vertex of CG_k. Thus, $r_{i_1}\langle j\rangle = 10000$, $r_{i_3}\langle h\rangle = 10000$, $r_{i_8}\langle k\rangle = 10000$; for any other position not in i-th block, the resolved vectors in R_i have value 0.

Given the set R of all resolved vectors of graph CG, then we construct the set F_G of fingerprints instance of IEC(2) as in the second reduction of Section 3 by applying the same rule. More precisely, given $f_{u,v}$ the fingerprint associated to (r_u, r_v), then for each position t, $f_{u,v}[t] := r_u[t]$ if $r_u[t] = r_v[t]$, and $f_{u,v}[t] := N$ otherwise.

By construction, two resolved vectors associated with adjacent vertices in CG have at most Hamming distance 2, thus each fingerprint in F_G has at most 2 positions with value N. Moreover, observe that fingerprints $f_{i_{2,7}}$, $f_{i_{6,4}}$ and $f_{i_{6,5}}$ associated with edges (r_{i_2}, r_{i_7}), (r_{i_6}, r_{i_4}) and (r_{i_6}, r_{i_5}) respectively, have exactly one position with value N, since the resolved vectors associated with the endpoints of such edges have Hamming distance 1. The set of fingerprints F_G has the following nice properties.

Lemma 2. *Let S be a solution of IEC(2) over instance F_G, then there is a solution S' having at most the same cost and such that each resolved vector of the solution is a resolved vector in R.*

Lemma 3. *Two fingerprints $f_x, f_y \in F_G$ are compatible if and only if they are associated with two edges of graph CG incident on a common vertex v. The resolved vector associated to v is the only common resolution of f_x, f_y in R.*

By the above results, we can restrict to a solution where each set s_v (where v is a vertex of the gadget) contains fingerprints that are resolved by a vector $r_v \in R$ and we say that fingerprints in s_v are *assigned* to r_v. A solution of IEC(2) is computed assigning the fingerprints in F_G to the resolved vectors in R. A fingerprint associated with edge (r_x, r_y) of CG_i is assigned to exactly one of r_x and r_y. Hence a solution of IEC(2) corresponds to assign each edge of CG

(a fingerprint f_x) to one of its endpoints in CG (resolved vectors compatible with f_x). Two edges are co-clustered if they are assigned to the same vertex. Hence the measure of a solution is the number of pairs of co-clustered edges.

In the following, to simplify the notation, we denote each edge of graph CG with the fingerprint associated with it. In particular, note that f_{ij} will denote edge E_{ij}.

By using Lemma 3, we will show that for a solution of IEC(2) over fingerprints F_i associated with a compatibility graph CG_i, we can restrict to two possible cases. In *Solution A* all the edges are assigned to r_{i_2}, r_{i_4} and r_{i_5}. Three edges are assigned to each of these vertices, thus 9 pairs of compatible fingerprints are co-clustered by solution A. In *Solution B* all the edges except for edge $f_{i_{27}}$ (that is edge (r_{i_2}, r_{i_7})) are assigned to r_{i_1}, r_{i_3}, r_{i_6} and r_{i_8}. One pair of edges is assigned to each of these vertices, while $f_{i_{27}}$ is assigned to either r_{i_2} or r_{i_7} and it is not co-clustered with other edges. Thus 4 pairs of compatible fingerprints are co-clustered by solution B.

The following lemma is easily proved using Lemma 3.

Lemma 4. *A solution A of IEC(2) over F_i has the same cost of every solution Z over $F_i \cup f_{i,j} \cup f_{i,h} \cup f_{i,k}$ that extends solution A over F_i.*

Next we show that the optimal solution for $F_i \cup f_{i,j} \cup f_{i,h} \cup f_{i,k}$ corresponds to have solution B for F_i and assign each edge in $\{f_{i,j}, f_{i,h}, f_{i,k}\}$ to a distinct docking vertex of CG_i. For each edge f_x in $\{f_{i,j}, f_{i,h}, f_{i,k}\}$, 2 pairs of co-clustered edges in $F_i \cup f_x$ are gained in solution B assigning the edge to a docking vertex of CG_i. Hence 10 pairs of edges are co-clustered in the extended solution B. We denote such a solution by B^*. By Lemma 4, the following result holds.

Lemma 5. *The optimal solution of IEC(2) over instance $F_i \cup f_{i,j} \cup f_{i,h} \cup f_{i,k}$ is B^*.*

By Lemma 5, it follows that any solution over instance $F_i \cup f_{i,j} \cup f_{i,h} \cup f_{i,k}$ different from B^* is not better than every solution extending solution A. Moreover, since each edge $f_{i,j}$ can be assigned to either r_{i_x} or r_{j_y}, two adjacent compatibility graphs cannot both have a solution B^*. Hence the problem of maximizing the number of co-clustered pairs of fingerprints consists of building an independent set I of compatibility graphs, as stated in the following result.

Theorem 2. *Let G be an instance of MIS. Then, there exists an independent set V' of size k in G if and only if exists a solution S of IEC(2) over instance F_G that co-clusters at least $10k + 9(n - k)$ pairs of compatible fingerprints.*

For each cubic graph $|E| = \frac{3}{2}|V|$ and there exists an independent set of size at least $|V|/4$, hence the above reduction is an L-reduction.

4.1 APX-Hardness of OEC(2)

Observe that the L-reduction described above implies the APX-hardness also of OEC(2). Indeed considering the set of fingerprints associated with a component graph CG_i and with edges $EG_{i,j}$, $EG_{i,h}$, $EG_{i,k}$, we can have 19 compatible pairs of fingerprints. The best solution for this set of fingerprints is solution B^*, which co-clusters 10 pairs of compatible fingerprint vectors and thus it does not co-cluster $19 - 10 = 9$ pairs of compatible fingerprints. Solution A does not co-cluster $19 - 9 = 10$ pairs of compatible fingerprints and no other solution different from solution B^* is better than solution A. Hence the L-reduction for OEC(2) follows directly from the L-reduction for IEC(2).

5 A Polynomial Time Algorithm for Restricted CMV

In this section we will present a polynomial-time algorithm for solving the CMV problem in the case where for each position of a fingerprint vector, there is at most one fingerprint in F with a N symbol in such position (notice that the number of Ns in each fingerprint is unbounded).

Let F be an instance of CMV, in [6] it has been shown that the CMV problem is equivalent to Minimum Cliques partition on a graph $G = (F, E)$ whose vertices are the input fingerprints and where the pair (f_i, f_j) is an edge of G if and only if f_i and f_j are compatible (that is they can be resolved by a same fingerprint). Since in each position there is only one fingerprint with N in such position, we are able to prove some properties of the graph G, namely: (i) any cycle in G induces a clique, (ii) any two maximal cliques share at most one vertex, (iii) given two maximal cliques K_1, K_2 sharing the vertex v_k, all paths connecting a vertex in $K_1 - \{v_k\}$ with a vertex in $K_2 - \{v_k\}$ pass through v_k. Due to space constraints we will prove only (i) since the other two properties follow from (i). Assume to the contrary that $C_t = \{f_{i_1}, f_{i_2}, \ldots f_{i_t}\}$ is a cycle of G that does not induce a clique (notice that $t \geq 4$), w.l.o.g. we can assume that (f_{i_1}, f_{i_u}) (with $2 < u < t$) is not an edge of G. Then f_{i_1} and f_{i_u} are not compatible, that is for a certain position z, $f_{i_1}[z] = 0$ and $f_{i_u}[z] = 1$. Moreover in G there are two vertex-disjoint paths from f_{i_1} to f_{i_u}, where each edge in the paths consisting of pair of compatible fingerprints and thus in both paths there must be a fingerprint with an N in position z. Since the paths are vertex disjoint, there must be two distinct fingerprints with an N in position z, contradicting the assumption that for each position only one fingerprint contains an N in that position.

Exploiting property (i) we are able to prove that there exists a vertex v of G belonging to exactly one maximal clique K of G (let us call such a vertex *private*). Assume to the contrary that such vertex does not exists, then consider two maximal cliques adjacent if they share a common element. Starting from any maximal cliques, visit all maximal cliques of G in a depth-first-like search (picking one of the not visited maximal cliques adjacent to the current one). If the currently visited maximal clique contains a private vertex, the search halts. By hypothesis the procedure visits all maximal cliques in a connected component of G, without finding a private vertex. Let K_{last} be the last maximal clique visited by the procedure, it is immediate to note that the procedure visited at least another maximal clique before K_{last}, let K_{last-1} be the parent of K_{last} in the search tree.

Since K_{last} is maximal, then it contains at least two vertices, and since the procedure halts all vertices in $K_{last} - K_{last-1}$ belong to maximal cliques that have been already visited. This implies that in the depth-first search tree there is a back edge, and consequently there is a cycle of G including vertices of both $K_{last} - K_{last-1}$ and $K_{last-1} - K_{last}$. By property (i), there is a clique including $K_{last} \cup K_{last-1}$, contradicting the maximality of K_{last}, K_{last-1}.

The algorithm simply finds such a maximal clique K containing a private vertex, adds K to the current clique cover, and removes all vertices of K from G updating G. The algorithm iterates until G contains no vertex. The correctness of the algorithm follows from a simple observation: each vertex must be covered in some solution, therefore each private vertex must be covered by one clique. Clearly covering all private vertices of K with K is an optimal choice, therefore let us consider a non-private vertex $w \in K$. If in an optimal solution w is covered by a clique different from K, then w can be covered by K without increasing the total number of cliques in the solution, hence K is an optimal solution.

The polynomial time complexity of the algorithm is a consequence of the following two observations. Two compatible fingerprints have exactly one common resolved vector (which is trivially computable), hence a maximal clique of graph G consists of all fingerprints sharing the same common resolved vector. Secondly, by a simple counting argument there are at most $\binom{n}{2}$ maximal cliques.

References

1. P. Alimonti and V. Kann. Some APX-completeness results for cubic graphs. *Theoretical Computer Science*, 237(1–2):123–134, 2000.
2. G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi. *Complexity and Approximation: Combinatorial optimization problems and their approximability properties*. Springer-Verlag, 1999.
3. R. Drmanac. cDNA screening by array hybridization. *Meth. in Enzym.*, 303:165–178, 1999.
4. S. Drmanac and R. Drmanac. Processing of cDNA and genomic kilobase-size clones for massive screening mapping and sequencing by hybridization. *Biotechn.*, 17:328–336, 1994.
5. S. Drmanac, N. Stavropoulos, I. Labat, J. Vonau, B. Hauser, M. Soares, and R. Drmanac. Gene-representation cDNA clusters defined by hybridization of 57 419 clones from infant brain libraries with short oligonucleotide probes. *Genomics*, 37:29–40, 1996.
6. A. Figueroa, J. Borneman, and T. Jiang. Clustering binary fingerprint vectors with missing values for DNA array data analysis. *Journal of Computational Biology*, 11(5):887–901, 2004.
7. A. Figueroa, A. Goldstein, T. Jiang, M. Kurowski, A. Lingas, and M. Persson. Approximate clustering of fingerprint vectors with missing values. In *Proc. 11th Computing: The Australasian Theory Symposium (CATS)*, volume 41 of *CRPIT*, pages 57–60, 2005.
8. L. Valinsky, G. Della Vedova, T. Jiang, and J. Borneman. Oligonucleotide fingerprinting of rRNA genes for analysis of fungal community composition *Applied and Environmental Microbiology*, 68(12): 5999–6004, 2002.
9. L. Valinsky, G. Della Vedova, A. Scupham, S. Alvey, A. Figueroa, B. Yin, R. Hartin, M. Chrobak, D. Crowley, T. Jiang, and J. Borneman. Analysis of bacterial microbial community composition by oligonucleotide fingerprinting of rRNA genes. *Applied and Environmental Microbiology*, 68(7): 3243–3250, 2002.

Tiling an Interval of the Discrete Line

Olivier Bodini and Eric Rivals

L.I.R.M.M.
Université de Montpellier II, CNRS U.M.R. 5506
161 rue Ada, F-34392 Montpellier Cedex 5, France
{rivals, bodini}@lirmm.fr

Abstract. We consider the problem of tiling a segment $\{0, \ldots, n\}$ of the discrete line. More precisely, we ought to characterize the structure of the patterns that tile a segment and their number. A pattern is a subset of $\mathbb{N}$. A tiling pattern or tile for $\{0, \ldots, n\}$ is a subset $A \in \mathcal{P}(\mathbb{N})$ such that there exists $B \in \mathcal{P}(\mathbb{N})$ and such that the direct sum of A and B equals $\{0, \ldots, n\}$. This problem is related to the difficult question of the decomposition in direct sums of the torus $\mathbb{Z}/n\mathbb{Z}$ (proposed by Minkowski). Using combinatorial and algebraic techniques, we give a new elementary proof of Krasner factorizations. We combinatorially prove that the tiles are direct sums of some arithmetic sequences of specific lengths. Besides, we show there are as many tiles whose smallest tilable segment is $\{0, \ldots, n\}$ as tiles whose smallest tilable segment is $\{0, \ldots, d\}$, for all strict divisors d of n. This enables us to exhibit an optimal linear time algorithm to compute for a given pattern the smallest segment that it tiles if any, as well as a recurrence formula for counting the tiles of a segment.

1 Introduction

Tilings are intriguing in many regards. Their structure, *i.e.*, the way in which the tiles are assembled, may be remarkably complex. As a matter of fact, a theorem from Berger [3] states that, given a set of patterns, determining whether this set tiles the plane by translation is undecidable. This result lets us think there exist sets of tiles that tile the plane only in complex ways. Indeed, Penrose and others [15, 5] demonstrated there exist aperiodic sets of tiles (aperiodic means that it tiles the plane, but that none of its tilings admits an invariant by translation). However, some related questions remain open. The smallest known aperiodic set of tiles contains 13 tiles and it is unknown whether there is one with only one non necessarily connected tile. Over and above that, it is undetermined whether the tiling of the plane with one non connected pattern is decidable. Nevertheless, an interesting result from Beauquier-Nivat [1] states that if the pattern is connected the problem is decidable, and if there exists a tiling, there is also a (doubly)-periodic tiling (*i.e.*, one that is invariant by two non-collinear vectors).

Even when restricted to bounded regions of the plane, tiling problems remain difficult combinatoric questions on which little is known. Numerous articles report on specific cases. Among others, the problem of tiling a connected region

M. Lewenstein and G. Valiente (Eds.): CPM 2006, LNCS 4009, pp. 117–128, 2006.

(respectively, a simply connected region) by dominos is polynomial [19] (resp., linear [20]). But, if generalized to a region that is non necessarily simply connected, the problem of tiling by rectangles of size 1×3 and 3×1 becomes NP-complete [2].

Regarding these difficulties, it is natural to focus on tiling problems for the discrete line $\mathbb{Z}$. These problems are related to additive number theory, which studies the decompositions of sets of numbers in sums of sets of numbers. A major theorem in this field is the decomposition of integers in sums of 4 squares (Lagrange's theorem), which is written $4C = \mathbb{N}$ where $C := \left\{n^2 : n \in \mathbb{N}\right\}$. Let us also mention Golbach's conjecture (in a letter to L. Euler, 1742), which asks whether any even integer is the sum of two primes ($2\mathbb{N} = \mathbb{P} + \mathbb{P}$ where $\mathbb{P}$ designs the set of primes).

Indeed in additive number theory, tiling the discrete line with a tile is equivalent, given a set A (representing the tile), to finding a set B (representing the positions of the tile's translations) such that the function $f(a, b) := a + b$ is one-to-one from $A \times B$ into $\mathbb{Z}$. In this case, we denote it $A \oplus B = \mathbb{Z}$. A classical result [12] states that in this case, there always exists a positions set B that is periodic (*i.e.*, for which there is an integer k such that $B + k = B$). As an immediate corollary, one obtains the decidability of the tiling of the discrete line by a single pattern. Despite that, its algorithmic complexity remains open although a lot of efforts have been dedicated to study bases for the integers [6, 21]. Moreover, the periodicity of the positions set B raises the question of the characterization of sets A and B such that $A \oplus B = \mathbb{Z}/n\mathbb{Z}$. This problem formulated by Minkowski more than hundred years ago is still mainly unsolved despite the last progresses made by Hajòs [9, 10, 17, 18].

In this work, we focus on the characterization of sets A and B satisfying $A \oplus B = [\![n]\!]$, where $[\![n]\!]$ denotes the interval $\{0, ..., n-1\}$. This question has been addressed in the literature as the Krasner factorization [11]. Two different constructions of Krasner factorizations have been described in the literature as special cases of Hajós factorizations and used in code theory [13, 7]. In a first part (Sections 2 and 3), we demonstrate using techniques from word theory that if $A \oplus B = [\![n]\!]$ then either A or B tiles $[\![d]\!]$, for d a proper divisor of n. For any $n \in \mathbb{N}$, let us say a tile is *n-specific* if its smallest tilable segment is $[\![n]\!]$. More precisely, we exhibit a bijection between n-specific tiles and d-specific tiles for all strict divisors d of n. This result yields a simple formula to count the tilings of $[\![n]\!]$. The obtained sequence that for each n gives the number of tilings of $[\![n]\!]$ is described in the Encyclopedia of Integer Sequences [16] (http://www.research.att.com/˜njas/sequences/) by Zumkeller without relationship neither to tilings theory, nor to word combinatorics. Besides, we prove a theorem on the size of the smallest tilable segment in function of the tile's diameter. This solves in a specific case a conjecture of Nivat stating that the smallest torus $\mathbb{Z}/n\mathbb{Z}$ that can be tiled by a pattern of diameter d satisfies $n \leq 2d$. Moreover, we exhibit a linear time algorithm to decide whether a pattern tiles at least one interval of $\mathbb{Z}$ (Section 4).

In a second part (Section 4), using more algebraic techniques, we demonstrate that any tile of $\llbracket n \rrbracket$ can be decomposed in irreducible tiles (*i.e.*, tiles that are not sums of smaller tiles), which we characterize explicitly. This is in fact a new proof of Krasner factorizations. This combinatorial proof may help developping the theory in the general framework. Furthermore, we know for any n how many irreducible tiles there are. Note that to fit in the page limit, all proofs have been removed from this extended abstract.

1.1 Definitions and Notation

Subsets of $\mathbb{N}$ and Polynomials. Let $\mathbb{N}$, resp. $\mathbb{Z}$, be the set of non-negative integers, resp. of integers, and $\mathcal{P}(\mathbb{N})$ the set of finite subsets of $\mathbb{N}$. We denote the set of polynomials with coefficients in $\{0,1\}$ by $\{0,1\}[X]$. We define the mapping ρ that to a finite subset of $\mathbb{N}$ associates a polynomial of $\{0,1\}[X]$ by:

$$\begin{array}{ll} \rho : \mathcal{P}(\mathbb{N}) \rightarrow & \{0,1\}[X] \\ \quad A \qquad \longrightarrow & P_A(X) := \sum_{a \in A} X^a \end{array}$$

Clearly, ρ is one-to-one. For all $A \in \mathcal{P}(\mathbb{N})$, we denote by $c(A)$ its minimal element, by $d(A)$ its maximal element, and by $\#(A)$ its *cardinality*. $d(A)$ is also the degree of P_A.

Let $A, B \in \mathcal{P}(\mathbb{N})$ and $k \in \mathbb{N}$. The following operations on sets have correspondents for polynomials:

union: $P_{A \cup B} = P_A + P_B$ if and only if $A \cap B = \emptyset$,
difference: $P_{A \setminus B} = P_A - P_B$ if and only if $B \subset A$,
translation: if one denotes $A + k := \{a + k : a \in A\}$, then $P_{A+k}(X) = X^k \cdot P_A$.

We introduce a notation for the *direct sum*. Let us denote by $A \uplus B$ the union with repetition for all $b \in B$ of the translates $A + b$. In general, this union is a multi-set on $\mathbb{N}$, *i.e.*, $P_{A \uplus B} := P_{\uplus_{b \in B} A+b} = \sum_{b \in B} P_{A+b}$ is a polynomial with integral coefficients that are eventually strictly greater than 1. If there exists $C \in \mathcal{P}(\mathbb{N})$ such that $C = A \uplus B$, then we denote it by $C = A \oplus B$. In this case, $P_{A \uplus B} := P_{A \oplus B} = P_A P_B$ and it belongs to $\{0,1\}[X]$. In other words, we investigate the case where the sum is stable in $\mathcal{P}(\mathbb{N})$, or where the product of polynomials is stable in $\{0,1\}[X]$. One says that a polynomial is *irreducible* in $\{0,1\}[X]$ if it cannot be factorized in $\{0,1\}[X]$. When transposed to subsets of $\mathbb{N}$, A is *irreducible* means it is impossible to decompose A in a non trivial direct sum (*i.e.*, other than $\{0\} \oplus A$).

Besides, we say A is a *prefix* of B if and only if $A \subset B$ and $\forall i \in B$, $i \leq d(A) \Rightarrow i \in A$ (i.e., $B \cap [[d(A)]] = A$). By convention, one admits that $\emptyset$ is prefix of any other subset of $\mathbb{N}$. We denote by $\llbracket k \rrbracket$ the finite interval of $\mathbb{N}$ of length k whose minimal element is 0, *i.e.*, the interval $[0, k-1]$. We use the word segment as an alternate for interval.

In the sequel, for any finite subset A of $\mathbb{N}$, we assume that $c(A) = 0$ (this is always true up to a translation). We call A a *pattern* or *motif.* For a pattern A, $d(A)$ is also termed *diameter.*

2 Properties of the Direct Sum

In this section, we investigate the properties of the direct sum that are useful to study the tilings of an interval. Note that the propositions hereunder are true for subsets of $\mathbb{N}$, but not necessarily for multi-sets on $\mathbb{N}$.

Proposition 1 (Sums of prefixes). *Let A, B, B', C, C' be subsets of $\mathbb{N}$ such that $A \neq \emptyset$ and C is prefix of C'. Then, together $A \oplus B = C$ and $A \oplus B' = C'$ imply that B is prefix of B'.*

Proposition 2 (Sum of a partition). *Let A, B, D be subsets of $\mathbb{N}$ such that $D \subseteq A$ and $A \oplus B$ be a subset of $\mathbb{N}$. Let us denote by $\complement_A D$ the complement of D in A. Then $(D \oplus B)$ and $\big(\complement_A D \oplus B\big)$ partition $A \oplus B$.*

This proposition is not verified when $A \oplus B$ is multi-set on $\mathbb{N}$ that is not a subset of $\mathbb{N}$. For multi-sets, we have the following property: Let C, D be such that $A = C \biguplus D$, then $(C \oplus B) \biguplus (D \oplus B) = A \oplus B$. In general it is not true that $(C \oplus B) \cap (D \oplus B) = \emptyset$, even if $C \cap D = \emptyset$.

We state two propositions of simplification.

Proposition 3 (Difference of intervals). *Let A, B, C be subsets of $\mathbb{N}$ and $m, n \in \mathbb{N}$. If $A \oplus B = [\![m]\!]$ and $A \oplus C = [\![n]\!]$ with $n \geq m$, then there exists $D \subset \mathbb{N}$ such that $A \oplus D = [\![n - m]\!]$ and $D := \complement_C B - m$.*

Example 1. *Set $A := \{0, 2\}$, $B := \{0, 1, 4, 5\}$ and $B' := \{0, 1, 4, 5, 8, 9\}$. One has $A \oplus B = [\![8]\!]$ and $A \oplus B' = [\![12]\!]$,* i.e., *$m := 8$ and $n := 12$. Let $D := \complement_C B - m = \{8, 9\} - 8 = \{0, 1\}$, one obtains $A \oplus D = [\![4]\!] = [\![n - m]\!]$.*

Proposition 4 (gcd of intervals). *Let A, B be subsets of $\mathbb{N}$ and $m, n \in \mathbb{N}$. If $M \oplus A = [\![n]\!]$ and $M \oplus B = [\![m]\!]$, then there exists $C \in \mathbb{N}$ such that $M \oplus C = [\![\mathrm{gcd}(n, m)]\!]$.*

Proposition 5 (Multiple of an interval). *Let A, B be subsets of $\mathbb{N}$ and $n \in \mathbb{N}$ such that $A \oplus B = [\![n]\!]$. Then, for all $l \in \mathbb{N}$, $A \oplus \big(\oplus_{i=0}^{l-1}(B + in)\big) = [\![ln]\!]$.*

Note that if $\#(A)$ is prime, then A can be decomposed only in the direct sum of the neutral element and itself. We close with an elementary property.

Proposition 6. *For any $A \in \mathcal{P}(\mathbb{N})$, one has $\#(A) \leq d(A) + 1$ and both members are equal if and only if $A = [\![d(A)]\!]$.*

3 Tiling an Interval of the Discrete Line

In this section, let $n \in \mathbb{N}$ be an integer and f be a finite subset of $\mathbb{N}$ such that $d(f) < n$. We use the following notation:

- for any $x < y$, we denote $f \cap [x, y]$ by $f[x, y]$, and $f \cap [x, y[$ by $f[x, y[$;
- for any $0 \leq x \leq d(f)$, let us denote by $f[x]$ the subset $\{i \in f : i < x\}$.

Definition 1 (Tiling, dual). *Let $n \geq 0$ and f be a pattern such that $d\left(f\right) < n$. We say that f* tiles $\llbracket n \rrbracket$ *if and only if there exists $\hat{f}_n$, a subset of $\mathbb{N}$, such that $f \oplus \hat{f}_n = \llbracket n \rrbracket$. We call $\hat{f}_n$ the* dual *of f for n. The element of $\hat{f}_n$ are also called the* translation positions *for f.*

For a given n, the dual is unique. The notion of dual is idempotent: the dual of the dual of f is f itself, and $\hat{f}_n$ also tiles $\llbracket n \rrbracket$. We say that a pattern f that tiles $\llbracket n \rrbracket$ is *trivial* if $f := [0, n-1] = \llbracket n \rrbracket$ or $f := \{0\}$. We define a notion of *self-period* for a pattern. Without loss of generality, we assume that 0 belongs to $f \cap \hat{f}_n$ (which is true up to a translation).

Definition 2 (Self-period of a pattern). *Let $n \in \mathbb{N}$, f be a pattern such that $d\left(f\right) < n$ and p be an integer such that $0 \leq p \leq d\left(f\right)$. We say that p is a* self-period of f for length n *if and only if for any $i \in [0, n-p[$ one has*

$$i \in f \Leftrightarrow (i+p) \in f \ .$$

In other words, f[0,n-p[+ p = f[p,n[. For length n, we denote by $\Pi_n(f)$ the set of self-periods of f, and by $\pi_n(f)$ its smallest non null self-period.

Definition 3 (Completely self-periodic). *We say that a pattern is* completely self-periodic *for length n if and only if it is an arithmetic sequence.* I.e., *if and only if one has $j \in f \Leftrightarrow (\exists i \in \left[0, \lfloor n/c \rfloor\right] : j = ic)$, where c denotes the common difference.*

Note that if a pattern f is completely self-periodic then its common difference is its smallest non-null period, $\pi_n(f)$. We choose the word "self-period" to avoid confusion with the notion of a tiling's period mentioned in the introduction. However, for the sake of simplicity, we use the word period instead of self-period in the sequel, since the context prevents ambiguity. Furthermore, let us point out the connection between the notions of a pattern self-periodicity and of word periodicity.

Example 2. *Consider $n := 12$. The pattern $f := \{0, 1, 4, 5, 8, 9\}$ has periods 0, 4, and 8. So, $\pi_{12}(f) = 4$ and $\Pi_{12}(f) = \{0, 4, 8\}$. It can be decomposed in $\{0, 1, 4, 5, 8, 9\} = \{0, 1\} \oplus \{0, 4, 8\}$. These patterns, $\{0, 1\}$ and $\{0, 4, 8\}$ are completely periodic for lengths 2 and 12 resp., with smallest period 1 and 4 resp. Pattern f tiles $\llbracket 12 \rrbracket$; its dual for $n := 12$ is $\hat{f}_{12} := \{0, 2\}$, it tiles $\llbracket 4 \rrbracket$, $\llbracket 8 \rrbracket$ and $\llbracket 12 \rrbracket$. It is true that $\#\left(f\right) \times \#\left(\hat{f}_{12}\right) = 6 \times 2 = 12$.*

3.1 Properties of Patterns That Tile an Interval

Let f be a pattern. In the sequel, we assume that f tiles $\llbracket n \rrbracket$. First, we list some elementary properties of f.

Proposition 7. *Let f be a pattern that tiles $\llbracket n \rrbracket$. First, $\#\left(f\right) \times \#\left(\hat{f}_n\right) = n$, and second, $d\left(f\right) + d\left(\hat{f}_n\right) = n - 1$. Thus, we have either $d\left(f\right) > d\left(\hat{f}_n\right)$, or $d\left(f\right) < d\left(\hat{f}_n\right)$.*

Now, let us state a simple and useful property. It follows from the positivity of the pattern's elements and from the properties of the direct sum.

Proposition 8. *For any $x \in \llbracket n \rrbracket$, one has $[0, x] \subseteq f[0, x] \oplus \hat{f}_n$.*

In a tile, let us call a *block* a maximal set of consecutive positions. *E.g.*, in $f := \{0, 1, 4, 5, 8, 9\}$ the blocks are $\{0, 1\}$, $\{4, 5\}$, and $\{8, 9\}$. A block contains at least an element and may be a singleton.

Now observe the following simple fact: the gap between the first and second block can only be tiled by translations of the first block (and of course of the whole tile). We show below that this implies first, that all blocks have the same length, and second that the first block tiles periodically the interval between 0 and the start position of the second block.

Proposition 9. *Let f be a pattern that tiles $\llbracket n \rrbracket$. Assume f comprises $k > 1$ blocks; then f is completely specified by the length and starting positions of its k blocks denoted respectively, $(b_i)_{1 \leq i \leq k}$ and $(l_i)_{1 \leq i \leq k}$. W.l.o.g. $b_1 = 0$, and for all i one has $l_i > 0$. Then:*

1. *the block length divides b_2,* i.e., *l_1 divides b_2, and $\hat{f}_n[b_2] = \cup_{j=0}^{b_2/l_1} \{jl_1\}$.*
2. *all blocks have the same length,* i.e., *for all $1 \leq i \leq k$, $l_i = l_1$.*

A corollary of the previous proposition is that the distance between any consecutive block is a multiple of the block length and is larger than b_2. We can now state a theorem showing that a tile f admits a non null smallest self period.

Theorem 1. *A tile f admits a smallest non-null period $\pi_n(f)$.*

Let us show that the smallest non null period of a non trivial tile is smaller than $\lfloor n/2 \rfloor$. Next proposition demonstrates that this period divides n.

Proposition 10. *Let f be a pattern that tiles $\llbracket n \rrbracket$ and such that $d(f) > d\left(\hat{f}_n\right)$. Then: $\pi_n(f) \leq \lfloor n/2 \rfloor$.*

Lemma 1. *Let f be a pattern that tiles $\llbracket n \rrbracket$ and satisfies $d(f) > d\left(\hat{f}_n\right)$. Thus, $\pi_n(f)$ divides n and $f[\pi_n(f)] \oplus \hat{f}_n = \llbracket \pi_n(f) \rrbracket$.*

The next corollary follows from the patterns' properties and from Lemma 1.

Corollary 2. *If f tiles $\llbracket n \rrbracket$ and $d(f) > d\left(\hat{f}_n\right)$ then $d\left(\hat{f}_n\right) < \pi_n(f)$.*

By Proposition 5, we have that any tile of $\llbracket n \rrbracket$ also tiles $\llbracket ln \rrbracket$ for any integer $l > 0$. We deduce the next corollary from Lemma 1 and Proposition 5.

Corollary 3. *Let f be a pattern and d be the smallest integer such that f tiles $\llbracket d \rrbracket$. If $d > 0$, then the $\llbracket ld \rrbracket$, for $l \in \mathbb{N}$, are all the intervals f can tile.*

Theorem 4. *Let n be an integer. Among the patterns f that tile $[\![n]\!]$, it exists a one-to-one mapping that, to any pattern f such that $d(f) \leq n/2$, associates a pattern that tiles $[\![d]\!]$ for d a divisor of n. This bijection associates to such a pattern f its dual $\hat{f}_n$.*

One obtains a canonical decomposition of patterns tiling $[\![n]\!]$ in irreducible patterns. Indeed, Theorem 4 allows us to write any tile f of $[\![n]\!]$ as the direct sum of i/ a completely periodic pattern for length n (with period a divisor strict of n) and ii/ one or more patterns that tiles $[\![d]\!]$, with d a strict divisor of n, and are completely periodic for length d. This decomposition result is also a corollary of Theorem 7 (section 4).

3.2 Numbers of Tiles of an Interval

Let $n \in \mathbb{N}$ such that $n > 0$. We denote by $\varXi_n$ the set of tiles of $[\![n]\!]$. Let $\varDelta_n$ be the subset of patterns in $\varXi_n$ whose diameter is smaller than or equal to $\lfloor n/2 \rfloor$ (*i.e.*, those who tile $[\![d]\!]$ for d a strict divisor of n), and let $\varPsi_n$ be the complement of $\varDelta_n$ in $\varXi_n$ (*i.e.*, those patterns with diameter strictly greater than $\lfloor n/2 \rfloor$). By definition, one has $\varXi_n = \varDelta_n \cup \varPsi_n$. We denote the cardinalities of these sets by ξ_n, δ_n, and ψ_n, respectively.

Theorem 5. *Let $n \in \mathbb{N}$ be an integer such that $n > 1$. One has $\xi_1 = 1$ and*

$$\boxed{\xi_n \quad = \quad 1 + \sum_{d \in \mathbb{N}\ :\ d|n,\ d \neq n} \xi_d\,.} \tag{1}$$

Corollary 6. *If $n > 1$ is prime then $\varDelta_n = \varPsi_1$, $\varPsi_n = \{[\![n]\!]\}$, $\varXi_n = \{\{0\}, [\![n]\!]\}$, $\delta_n = \psi_n = 1$ and $\xi_n = 2$.*

The values of ξ_n for $n > 0$ are those of Sequence entry A067824 in [16], and (1) corresponds to the recurrence relation given for this sequence by Zumkeller.

Let us denote by $\mu(n)$ the Moebius function. This function satisfies $\mu(n) = (-1)^r$ if $n = p_1 p_2 ... p_r$ for distinct primes p_j and $\mu(n) = 0$ whenever n is divisible by a square. The Moebius inversion states that $f(n) = \sum_{d|n} g(d) \Leftrightarrow g(n) = \sum_{d|n} \mu(\frac{n}{d}) f(d)$. Using this property, we can easily obtain the following new induction for ξ_n : $\xi_n = 1 - \sum_{d|n; d \neq n} \mu(\frac{n}{d})(2\xi_d - 1)$.

4 Algebraic Approach

4.1 Polynomials Decomposition

Let us denote by $\mathcal{C}$ the set of *super-composite* integers, *i.e.*, all integers whose prime factorization contains at least two different primes. It is known that $X^n - 1$ admits a unique decomposition (up to the order of its factors) in irreducible elements of $\mathbb{Z}[X]$. Indeed, $\mathbb{Z}[X]$ is a factorial ring (unique factorization domain). This decomposition is $X^n - 1 = \prod_{d|n} \varPhi_d$, where $\varPhi_d$ is the *d-th cyclotomic polynomial* [14]. We use the following properties of cyclotomic polynomials.

Proposition 11

- *The degree of Φ_d is $\varphi(d)$, where φ is Euler's function.*
- $\Phi_d(1) = p$ *if d is a power of a prime p and* $\Phi_d(1) = 1$ *otherwise.*
- *The polynomial Φ_d belongs to $\{0,1\}[X]$ if and only if $d \notin \mathcal{C}$.*

As ρ is a bijection, it induces a one-to-one correspondence between the pairs $(A, B) \in \mathcal{P}(\mathbb{N}) \times \mathcal{P}(\mathbb{N})$ such that $A \oplus B = [\![n]\!]$, and the pairs $(P, Q) \in (\{0,1\}[X] \times \{0,1\}[X])$ such that $P(X)Q(X) = 1 + \cdots + X^{n-1}$. Moreover, as $1 + \cdots + X^{n-1}$ is factorizable in $\prod_{d|n, d \neq 1} \Phi_d(X)$, there exists a partition of $\{d|n, d \neq 1\}$ in D_1 and D_2 such that $P(X) = \prod_{d \in D_1} \Phi_d(X)$ and $Q(X) = \prod_{d \in D_2} \Phi_d(X)$. Finally, We can notice that if P is in $\{0,1\}[X]$ and $P(X)Q(X) = 1 + \cdots + X^{n-1}$ then the polynomial Q is not necessarily in $\{0,1\}[X]$. We have the following counter-example : $(1 + X^2 + X^3 + X^5)(1 + X - X^3 + X^5 + X^6) = 1 + \cdots + X^{11-1}$.

4.2 Results

Lemma 2. *Let $P_1, \ldots, P_k$ be polynomials of $\{0,1\}[X]$ such that $\prod_{i=1}^{k} P_i$ belongs to $\{0,1\}[X]$. For each subsequence $P_{s_1}, \ldots, P_{s_t}$, with $1 \leq s_1, \ldots, s_t \leq k$, one has $\prod_{i=1}^{t} P_{s_i} \in \{0,1\}[X]$.*

For all $n \in \mathbb{N}$, we call *total valuation* of n, denoted by ν_n, the sum of the powers in the prime factorization of n. We call *factorial sequence* of n, a sequence $u_0, u_1, \ldots, u_s$ such that $u_0 := 1$, $u_s := n$, and u_{i+1}/u_i is a prime number. Observe that all factorial sequences of n have $\nu_n + 1$ terms. From a factorial sequence of n, we can build a *sequence of decomposition* $(D_{u_{i-1},u_i})_{1 \leq i \leq s}$ with $D_{u_{i-1},u_i} := \{d | u_i : d \nmid u_{i-1}\}$. For conciseness, for all $D \in \mathcal{P}(\mathbb{N})$ we write $\Phi_D := \prod_{d \in D} \Phi_d$.

Lemma 3. *Let $n, p \in \mathbb{N}$ with p prime. $\Phi_{D_{n,pn}}$ belongs to $\{0,1\}[X]$ and is irreducible in $\{0,1\}[X]$.*

Theorem 7 (Krasner Factorizations). *Each factorization of $1 + \cdots + X^{n-1}$ in irreducible elements in $\{0,1\}[X]$ has the following form $\prod_{1 \leq i \leq s} \Phi_{D_{u_{i-1},u_i}}$ where $(D_{u_{i-1},u_i})_{1 \leq i \leq s}$ is a sequence of decomposition of n, and reciprocally. Moreover, for all $1 \leq i \leq s$, $\Phi_{D_{u_{i-1},u_i}}(1)$ is a prime factor of n.*

Note that the factorization may not be unique.

Example 3. *For $n := 12$, the factorial sequences are: $(1, 2, 4, 12)$, $(1, 2, 6, 12)$, and $(1, 3, 6, 12)$. The associated sequences of decomposition are $(\{2\}, \{4\}, \{3, 6, 12\})$, $(\{2\}, \{3, 6\}, \{4, 12\})$, $(\{3\}, \{2, 6\}, \{4, 12\})$. We obtain that the irreducible factors of $1 + \cdots + X^{n-1}$ in $\{0,1\}[X]$ are Φ_2, Φ_3, Φ_4, $\Phi_3\Phi_6$, $\Phi_2\Phi_6$, $\Phi_3\Phi_6\Phi_{12}$, $\Phi_4\Phi_{12}$.*

Theorem 8. *The number v_n of irreducible factors of $1 + \cdots + X^{n-1}$ in $\{0,1\}[X]$ equals $\sum_{d|n} \#\{$prime factors of $d\}$.*

The pattern associated with the polynomial $\Phi_{D_{d,dp}}$ is the arithmetic sequence starting in 0, of common difference d, and having p terms. This gives the precise structure of all tiles of a segment.

A *reciprocal polynomial* is a polynomial such that $P(X) = X^n P(1/X)$, where n is the degree of P.

Corollary 9. *Let f be a pattern which tiles an interval. The associated polynomial, P_f, is reciprocal.*

Theorem 10. *Let f be a pattern which tiles an interval. Then, the length of the smallest nonempty interval it tiles is smaller than twice the diameter of f,* i.e., *than $2d(f)$.*

Theorem 10 shows that Nivat's conjecture on the upper bound of the tiling periodicity (also mentioned in [4]) is true for the special cases considered here.

Theorem 11. *Let f be a pattern. Algorithm 1 decides in $O(d(f))$ time whether there is $n \in \mathbb{N}$ such that f tiles $[\![n]\!]$ and gives the decomposition of f in completely self-periodic tiles.*

In Algorithm 1, we use a procedure call CompNextBlock (or CNB for short) that computes all information needed about the next block in the tile. This procedure uses a global variable to scan blocks from left to right in the tile. It stores in a block object the following information: length (lg), distance to previous block or 0 for the first block (dist). Note that the distance is the length of the space that separates two successive blocks. At the last block, dist is set to -1.

In fact, the algorithm computes the decomposition of the tile in completely self-periodic tiles. It scans the blocks, checks their length, and deduce the level of periodicity they belong to. The information on each level is stored in a Level object: characteristic distance between the last block of previous level and the first block of this level, period, number of repeats, overall length (this is the product of the number of repeats time the period). Other variables are: Lev: table of levels; ilev: index of the current level; pdist: previous distance between two consecutive blocks; fB: first block; nB: current block;

Two cases arise with the distance. Either the distance between the current consecutive blocks is the same as the previous one, then the periodicity remains the same and we have to scan a new copy (repeat) of the current level. Or the distance increases, then the current blocks marks the beginning of a new level of periodicity. Once a periodicity level has been scanned, the variable *Lev* stores its complete description. A level of periodicity corresponds to a prefix of f. To any given level corresponds a prefix of $\hat{f}_n$ such that it sums with that level tiles the whole level length. Let us denote that by $Lev[i] \oplus p_i(\hat{f}_n)$, where $p_i(\hat{f}_n)$ denotes the prefix of $\hat{f}_n$ that tiles $Lev[i].lg$. The start of a higher level is detected when the inter-block distance increases strictly. The additional space left since the end of the last periodicity level has to be a multiple of $Lev[i].lg$, such that it can be filled with translates of $Lev[i] \oplus p_i(\hat{f}_n)$. The higher level, say $i + 1$ starts necessarily with replicates of all previous periodicity levels. The line "read all previous level" scans these replicates with the descriptions stored in *Lev*.

Algorithm 1. Computes the least interval tiled by f if any and returns -1 otherwise. If f is a tile, the levels of periodicity computed by the algorithm gives the decomposition of f in completely self-periodic tiles.

```
Data: a pattern $f$
Result: the minimal $n \in \mathbb{N}$ such that $f$ tiles $[\![n]\!]$ if it exists, and $-1$ otherwise
if $(d(f)+1 = \#(f))$ then return $\#(f)$;if $(\#(f) = 2)$ then return $2d(f)$;
fB := CNB($f$); // compute the first block;
if $(fB.lg \nmid \#(f))$ then return $(-1)$; // check if 1st block length divides $\#(f)$;
nB := CNB($f$); $ilev := 0$; $pdist := 0$; // compute next Block; init ilev and pdist;
while $(nB.dist \geq 0)$ do
    // check the block length and that the distance increases;
    if $(nB.lg \nmid fB.lg)$ or $(nB.dist < pdist)$ then return $(-1)$;
    // if current distance is larger than previous one: start of a new level;
    if $(nB.dist > pdist)$ then
        $ilev++$;
        if $(ilev = 1)$ then
            // First level;
            // check if the block length divides the distance between blocks;
            if $(fB.lg \nmid nB.dist)$ then return $(-1)$;
            // init current level: period = space length + block length;
            $Lev[ilev].dist$ := nB.$dist$;
            $Lev[ilev].period$ := nB.$dist$ + fB.$lg$;
            $Lev[ilev].nbrep := 1$;
        else
            // Higher level;
            // update the nbrep of prev level and compute its length;
            $Lev[ilev-1].nbrep++$; $Lev[ilev-1].ComputeLg()$;
            // check if previous level period divides the distance difference between the 2
            levels;
            if $(Lev[ilev].period \nmid (nB.dist - Lev[ilev-1].dist))$ then return $(-1)$;
            // init current level: period = previous level's length + distances' difference;
            $Lev[ilev].dist$ := nB.$dist$;
            $Lev[ilev].period := Lev[ilev-1].lg$ + (nB.$dist - Lev[ilev-1].dist$);
            $Lev[ilev].nbrep := 1$;
            $pdist$ := nB.$dist$;
            Read all lower levels until nB.$dist > Lev[ilev-1].dist$; if it fails return $(-1)$;
    else
        // update currently scanned level;
        $Lev[ilev].nbrep++$;
        if $(ilev \neq 1)$ then Read all lower levels until nB.$dist > Lev[ilev-1].dist$; if it
        fails return $(-1)$;
    if $(ilev = 1)$ then
        // only in case of 1st level: compute new block and update variables;
        $pdist$ := nB.$dist$; nB := CNB($f$);
// increment nb of repeats of current level and compute its length;
$Lev[ilev].nbrep++$; $Lev[ilev].ComputeLg()$;
return $Lev[ilev].lg$; // the interval tiled by $f$ is $Lev[ilev].lg$;
```

Example 1. Let $f := [0,6] \cup [21,27] \cup [42,48] \cup [126,132] \cup [147,153] \cup [168,174] \cup [504,510] \cup [525,531] \cup [546,552] \cup [630,636] \cup [651,657] \cup [672,678]$, with $\#\,(f) = 84$ and $d\,(f) = 678$. Then Algorithm 1 will find out that the smallest n such that f tiles $\llbracket n \rrbracket$ is $n = 1008$ and will decompose f as follows:

$$\begin{aligned} f &= \llbracket 7 \rrbracket \oplus \{0, 21, 42, 126, 147, 168, 504, 525, 546, 630, 651, 672\} \\ &= \llbracket 7 \rrbracket \oplus \{0, 21, 42\} \oplus \{0, 126, 504, 630\} \\ &= \llbracket 7 \rrbracket \oplus \{0, 21, 42\} \oplus \{0, 126\} \oplus \{0, 504\}\ . \end{aligned}$$

The algorithm infers 4 levels of periodicity: the first block $\llbracket 7 \rrbracket$, which has period 1 and is not explicitely store in *Lev*, and then the following three levels:

$$\begin{array}{lcllcllcllcl} \text{Lev}[1].\mathit{dist} & = & 14 & \text{Lev}[1].\mathit{period} & = & 21 & \text{Lev}[1].\mathit{nbrep} & = & 3 & \text{Lev}[1].\mathit{lg} & = & 63 \\ \text{Lev}[2].\mathit{dist} & = & 77 & \text{Lev}[2].\mathit{period} & = & 126 & \text{Lev}[2].\mathit{nbrep} & = & 1 & \text{Lev}[2].\mathit{lg} & = & 252 \\ \text{Lev}[3].\mathit{dist} & = & 329 & \text{Lev}[3].\mathit{period} & = & 504 & \text{Lev}[3].\mathit{nbrep} & = & 1 & \text{Lev}[3].\mathit{lg} & = & 1008. \end{array}$$

The dual of f for $n = 1008$ is $\hat{f}_n = \{0, 7, 14\} \oplus \{0, 63\} \oplus \{0, 252\}$, which can also be infered from the length and number of repeats of the periodicity levels.

5 Conclusion

This work characterizes the tilings of an interval as direct sums of arithmetic sequences. Counting results obtained also show that, surprisingly, the number of patterns that tile a segment of length n depends, not on the prime factors of n, but only on the list of their powers. *E.g.*, segments of respective lengths $n_1 := 5 \times 7^2 \times 2^4$ and $n_2 := 13 \times 3^2 \times 11^4$ (n_1 and n_2 have both $(1, 2, 4)$ as list of powers), have the same number of tiles. Moreover, for any positive integer n we show that the number of polynomials p with coefficients in $\{0, 1\}$ that divide $x^n - 1$ and such that $(x^n - 1)/p(x - 1)$ has all coefficients in $\{0, 1\}$ equals the number of tiles of $\llbracket n \rrbracket$. This invalidates the conjecture mentioned in the Encyclopedia of Integer Sequences [16] that these two sequences, A107736 and A067824, are different. Finally, we exhibit a linear time algorithm to recognize a tile and find the smallest n for which it tiles $\llbracket n \rrbracket$. This complexity is otpimal.

The regular structure of the tiles of a segment contrasts sharply with the singular structure of those tiling the torus $\mathbb{Z}/n\mathbb{Z}$. Indeed for this problem, there exists irregular sets A and B such that $A \oplus B = \mathbb{Z}/n\mathbb{Z}$ [8]. However, our results exhibit a relation between tilings, words and polynomials that opens promising directions for the tiling by a single pattern of the discrete plane or of special cases of the torus. Let us mention that Theorem 7 can easily be extended to higher dimensions. As a matter of fact, one can characterize a pattern that tiles a d-dimensional rectangle $n_1 \times \ldots \times n_d$ as the cartesian product of d one-dimensional patterns, each tiling a segment of length n_i respectively (with $1 \leq i \leq d$). This work also shed light on the complementarity of combinatorial and algebraic approaches for tiling problems.

Acknowledgments. We are grateful to O. Gandouet for reading the manuscript and to F. Philippe for constructive comments.

References

1. D. Beauquier and M. Nivat. Tiling the plane with one tile. In *Proc. 6th Annual Symposium on Computational Geometry (SGC'90)*, pages 128–138, Berkeley, CA, 1990. ACM Press.
2. D. Beauquier, M. Nivat, E. Remila, and J.M. Robson. Tiling figures of the plane with two bars. *Computational Geometry: Theory and Applications*, 5, 1996.
3. R. Berger. The undecidability of the domino problem. *Mem. Amer. Math Soc.*, 66:1–72, 1966.
4. E. M. Coven and A. D. Meyerowitz. Tiling the integers with translates of one finite set. *Journal of Algebra*, 212:161–174, 1999.
5. K. Culik II and J. Kari. On Aperiodic Sets of Wang Tiles. *Lecture Notes in Computer Science*, 1337:153–162, 1997.
6. N.G. de Bruijn. On bases for the set of intergers. *Publ. Math. Debrecen*, 1:232–242, 1950.
7. C. De Felice. An application of Hajós factorizations to variable-length codes. *Theoretical Computer Science*, 164(1–2):223–252, 1996.
8. L. Fuchs. *Abelian Groups.* Oxford Univ. Press, 1960.
9. G. Hajós. Sur la factorisation des groupes abéliens. *Cas. Mat. Fys.*, 74(3):157–162, 1950.
10. G. Hajós. Sur le problème de factorisation des groupes cycliques. *Acta Math. Acad. Sci. Hung.*, 1:189–195, 1950.
11. M. Krasner and B. Ranulak. Sur une propriété des polynômes de la division du cercle. *Comptes rendus de l'Académie des Sciences Paris*, 240:397–399, 1937.
12. J.C. Lagarias and Y. Wang. Tiling the line with translates of one tile. *Inventiones Mathematicae*, 124(1-3):341–365, 1996.
13. N. H. Lam. Hajós factorizations and completion of codes. *Theoretical Computer Science*, 182(1–2):245–256, 15 August 1997.
14. S. Lang. *Algebraic Number Theory*, volume 110 of *Graduate Texts in Mathematics.* Addison-Wesley Publishing Company, 2nd edition, 2000.
15. R. Penrose. Pentaplexy. *Bulletin of the Institute of Mathematics and its Applications*, 10:266–271, 1974.
16. N. J. A. Sloane. The On-Line Encyclopedia of Integer Sequences, 2004. Available at http://www.research.att.com/˜njas/sequences/.
17. S.K. Stein and S. Szabo. *Algebra and Tiling: Homomorphisms in the Service of Geometry.* Carus Mathematical Monograph 25, MAA, 1994.
18. S. Szabo. *Topics in factorization of abelian groups.* Birkhauser, 2004.
19. N. Thiant. An $O(n \log n)$-algorithm for finding a domino tiling of a plane picture whose number of holes is bounded. *Theorical Computer Sciences*, 303(2-3):353–374, 2003.
20. W. P. Thurston. Conway's tiling groups. *Am. Math. Monthly*, pages 757–773, October 1990.
21. R. Tijdeman. Decomposition of the integers as a direct sum of two subsets. In S. David, editor, *Number Theory*, pages 261–276. Oxford Univ. Press, 1995.

Common Substrings in Random Strings

Eric Blais* and Mathieu Blanchette**

McGill Centre for Bioinformatics and School of Computer Science
McGill University, Montréal, Québec, Canada
{eblais, blanchem}@mcb.mcgill.ca

Abstract. In computational biology, an important problem is to identify a word of length k present in each of a given set of sequences. Here, we investigate the problem of calculating the probability that such a word exists in a set of r random strings. Existing methods to approximate this probability are either inaccurate when $r > 2$ or are restricted to Bernoulli models. We introduce two new methods for computing this probability under Bernoulli and Markov models. We present generalizations of the methods to compute the probability of finding a word of length k shared among q of r sequences, and to allow mismatches. We show through simulations that our approximations are significantly more accurate than methods previously published.

1 Introduction

Many algorithms in biological sequence analysis are based on the identification of words that are present as substrings of a given set of DNA or protein sequences. Variants on this problem are used for identifying regulatory motifs in a set of co-regulated genes [20, 22], and for selecting seeds for sequence alignment [1, 2, 13]. These applications rely on the ability to estimate the statistical significance of the length of the common substring found: how surprising is it that a set of r strings of length n contain a common substring of length k?

The problem we consider in this paper is the following:

Common Substring in Random Strings (CSRS)
Given: A random process $\mathcal{P}$ generating r independent strings $S_1, \ldots, S_r$ of length n over the alphabet Σ, and a substring length k,
Find: The probability that the r random strings $S_1, \ldots, S_r$ contain a common substring w of length k.

Various authors have studied his problem or its equivalent formulation as the longest common substring problem (see Section 2), but available methods are not accurate for finite length random sequences generated by a non-uniform Bernoulli or Markov process. In this article, we present a new approximation to the Common Substring in Random Strings (CSRS) problem and show that

* Partially supported by the André Courtemanche Fellowship in Bioinformatics.
** Partially supported by the National Science and Eng. Research Council of Canada.

M. Lewenstein and G. Valiente (Eds.): CPM 2006, LNCS 4009, pp. 129–140, 2006.

this new approximation is more accurate than previously published methods for the same problem. Moreover, we generalize our approach to solve the problems where matches are required in only q of the r strings, and where inexact matches are allowed.

The article is organized as follows. We review related work in Section 2. In Section 3, we present a model for solving the CSRS problem approximately using an assumption of independence between words. In Section 4, we present a second simplifying assumption of independence and show how the model obtained by combining both simplifying assumptions can compute approximations to CSRS in polynomial time (relative to k) for strings generated by Bernoulli and Markov processes. In Section 5, we present important generalizations for biological problems. Finally, in Section 6, we show through a set of Monte-Carlo simulations that our approach is quite accurate under all models considered.

2 Related Work

The probability that r strings contain a common *aligned* substring of length k can be determined by characterizing the length of the longest head run in a sequence of biased coin flips. This problem was examined by Feller [8] who provides a method for computing this probability with generating functions. The same problem was later studied by Erdős and Rényi [6] and Erdős and Révész [7] who provide tight asymptotic bounds on the distribution of the longest head run.

Arratia and Waterman [3, 4] generalize the results of Erdős and Rényi to examine the distribution of the longest common (unaligned) substring in two random strings. Karlin and Ost [12] provide further improvements on the bounds for the asymptotic behaviour of the longest common word in multiple random strings for a wide range of random processes.

Naus and Sheng [14] show that while the Karlin and Ost asymptotic equations provide very accurate approximations to CSRS on two random strings, the quality of the same equations deteriorates as the number of strings increases. They also provide refinements to the Karlin and Ost equations for the special case where the random strings are generated by a Bernoulli process, and show that those refinements offer very good approximations to the CSRS problem when the strings are generated by a uniform Bernoulli process. As we will show in Section 6, the quality of those approximations is not as good for strings generated by non-uniform Bernoulli processes, and their methods do not generalize to strings generated by Markov processes.

In a parallel effort, Guibas and Odlyzko [10] and many others since provide approximations and exact results for the distribution of the number of occurrences of a given word in a random text (see for instance [15, 18, 19] and the references therein). Those results allow us to compute the probability that a *given* word is a common substring to random strings. In the next section, we show how to apply these results to the computationi of the probability that a set of random strings contain *any* word as a common substring.

3 The Independent Words Model

Let ξ_w represent the event that the word w occurs in a random string S generated by a Bernoulli or Markov process. The value of $P(\xi_w)$ can be computed with the help of generating functions:

Theorem 1 (Régnier [18]). *The probability $P(\xi_w)$ that a string w of length k is found as a substring of the random string S of length n generated by a Bernoulli process or a stationary Markov process is*

$$P(\xi_w) = [s^n] \left(\frac{1}{1-s} \cdot \frac{P(w)s^k}{P(w)s^k + A_w(s)(1-s)} \right) \ , \tag{1}$$

where $A_w(s)$ is the autocorrelation polynomial of w (see [18]) and $P(w)$ is the stationary probability of observing w at a given position in S.

There are various methods to implement numerical computations of $P(\xi_w)$. Régnier proposes a method to compute the value of $P(\xi_w)$ in $O(\log n)$ time [18]. The partial fractions method of Feller can also compute highly accurate approximations to $P(\xi_w)$ in $O(k)$ time [8].

The computation of $P(\zeta)$, where ζ represents the event that some word of length k occurs in all r random strings $S_1, \ldots, S_r$, is more problematic. Since the strings $S_1, \ldots, S_r$ are generated independently, we have $P(\zeta) = P(\cup_{w \in \Sigma^k} \xi_w^{\ r})$. To compute $P(\zeta)$ exactly, we would need to account for the dependence between all the events ξ_w. However, as we will show in Section 6, we can get a very good approximation to $P(\zeta)$ even when we assume the independence of the events ξ_w. We therefore present the single assumption for the Independent Words Model:

Assumption 1. *For every words $w \neq w'$, we assume that the events ξ_w and $\xi_{w'}$ are independent.*

With this assumption, computing the value of $P(\zeta)$ is now straightforward.

Proposition 1. *When Assumption 1 holds, the probability $P(\zeta)$ of observing a substring of length k in each of the r random strings $S_1, \ldots, S_r$ generated by a Bernoulli or Markov process is*

$$P(\zeta) = 1 - \prod_{w \in \Sigma^k} \left(1 - P(\xi_w)^r\right) \ . \tag{2}$$

Equation (2) can be implemented directly to provide an accurate approximation to $P(\zeta)$. However, since it requires the enumeration of every word w of length k in the alphabet Σ of size σ, its running time is exponential in k, taking $\Omega(\sigma^k)$ time even when a constant-time approximation algorithm is used to compute $P(\xi_w)$. In the next section, we present alternative algorithms for computing the approximation to $P(\zeta)$ in time polynomial in k.

4 The Double Independence Model

To develop efficient approximations to $P(\zeta)$, we want to reduce the number of terms that need to be enumerated during the computation. Already, we may note that some words have the same probability $P(\xi_w)$ of occurring in a random string. In fact, we can significantly augment the number of words that share the same probability $P(\xi_w)$ with a second simplifying assumption.

Assumption 2. *For every position $i \neq j$ in S, we assume that the probability that the events representing occurrences of the word w at position i and j are independent.*

The Assumption 2 gives the following approximation for $P(\xi_w)$:

Proposition 2. *When Assumption 2 is valid, the probability $P(\xi_w)$ of observing a word w of length k in a random string S of length n is*

$$\tilde{P}(\xi_w) = 1 - \big(1 - P(w)\big)^{n-k+1} \ , \tag{3}$$

where $P(w)$ is the stationary probability of observing the word w at a given position in S.

The approximation $\tilde{P}(\xi_w)$ has two advantages: it is the same for many different words, and can be computed in $O(1)$ time. However, we should be aware that it is not an accurate approximation to $P(\xi_w)$ for many words. Specifically, the probability $P(\xi_w)$ for words that have high self-overlap (e.g. `AAAAAA` or `CTCTCT`) is very poorly approximated by Proposition 2 [22]. Nevertheless, we will show in Section 4.3 that it is easy to correct this error in polynomial time.

We refer to the modified CSRS problem in which Assumptions 1 and 2 are valid as the Double Independence Model. In Sections 4.1 and 4.2, we show how the Double Independence Model can be used to obtain polynomial time algorithms for the computation of $P(\zeta)$ when the random string S is generated by a Bernoulli process or a Markov process, respectively.

4.1 Bernoulli Process

When the random string S is generated by a Bernoulli process, each character of S is generated independently and takes the value $x \in \Sigma$ with probability p_x. Under the double independence model, words that share the same composition, as defined below, will have the same probability $\tilde{P}(\xi_w)$ of occurring in S.

Definition 1. *The* Bernoulli composition *of a string w is the multiset γ of characters in w.*

Example 1. The Bernoulli composition of the string $w = \texttt{ACCATA}$ is the multiset $\gamma = \{\texttt{A}, \texttt{A}, \texttt{A}, \texttt{C}, \texttt{C}, \texttt{T}\}$.

We define P_γ to be equal to the probability $\tilde{P}(\xi_w)$ for any word w with composition γ. We also define $N_\gamma(x)$ as the number of copies of the character x in γ. We let $\Omega(\gamma)$ represent the number of words w with composition γ, and we represent the set of all possible Bernoulli compositions for words of length k with $\mathcal{C}_k$. We then obtain the following theorem.

Theorem 2. *Let $S_1, \ldots, S_r$ represent r random strings of length n generated independently by a Bernoulli process over the alphabet Σ. When Assumptions 1 and 2 hold, the probabilty $P(\zeta)$ that the strings $S_1, \ldots, S_r$ share a common substring of length k is defined by*

$$P(\zeta) = 1 - \prod_{\gamma \in \mathcal{C}_k} \left(1 - {P_\gamma}^r\right)^{\Omega(\gamma)} \ , \tag{4}$$

where the probability that a word w with composition γ is found in a random string S_i is $P_\gamma = 1 - (1 - \prod_{\alpha \in \Sigma} {p_\alpha}^{N_\gamma(\alpha)})^{n-k+1}$ and the number of words that have the composition γ is

$$\Omega(\gamma) = \frac{k!}{\prod_{\alpha \in \Sigma} N_\gamma(\alpha)!} \ . \tag{5}$$

Proof. The equation for P_γ follows directly from Proposition 2, by noting that the probability $P(w)$ of observing a word w with composition γ in a given position in S is $P(w) = \prod_{\alpha \in \Sigma} {p_\alpha}^{N_\gamma(\alpha)}$. The number $\Omega(\gamma)$ of words with composition γ is equal to the number of distinct strings that can be formed from the symbols in γ, i.e. the multinomial coefficient in (5) [8]. The final result in (4) follows from Proposition 1 and the observation that every word w of length k is represented in exactly one composition γ in $\mathcal{C}_k$. □

To implement the result from Theorem 2 in an efficient algorithm, we need an efficient method to enumerate all the Bernoulli compositions in $\mathcal{C}_k$. This can be done through a simple recursive algorithm or by using the iterative algorithm of Nijenhuis and Wilf [16]. Either of these approaches uses a constant amount of computation to generate each composition. Assuming a constant alphabet size, the values of P_γ and $\Omega(\gamma)$ can both be computed in $O(1)$ time, and the total running time of a simple algorithm that implements Theorem 2 is $O(|\mathcal{C}_k|)$. Since the number of Bernoulli compositions in $\mathcal{C}_k$ is equal to the number of different compositions of the integer k into a maximum of σ parts, we get $|\mathcal{C}_k| = \binom{k+\sigma-1}{\sigma-1} \in O(k^{\sigma-1})$ and the computation of $P(\zeta)$ can be done in $O(k^{\sigma-1})$ time.

Uniform Bernoulli Process. In the special case where the random strings $S_1, \ldots, S_r$ are generated by a uniform Bernoulli process, the value $\tilde{P}(\xi_w)$ is identical for every word $w \in \Sigma^k$. In this case, the probability $P(\zeta)$ can be computed in constant time with

$$P(\zeta) = 1 - \left(1 - \left(1 - \left(1 - 1/\sigma^k\right)^{n-k+1}\right)^r\right)^{\sigma^k} \ . \tag{6}$$

This result is equivalent to the derivation obtained by Naus and Sheng [14] in their Equation (10) under the same special case of uniform Bernoulli processes.

4.2 Markov Process

Biological sequences rarely follow a Bernoulli model, but have been shown to be accurately modeled by low-order Markov models [20, 22]. To be applicable in a biological context, our probability calculations have to be accurate and tractable under such models. In this section, we present an algorithmic approach to approximate $P(\zeta)$ in polynomial time for 1st order Markov models. With minor changes, the approach described below can also be extended to mth order Markov models, for any fixed $m \geq 1$ [5].

Let S represent a random string generated by a stationary 1st order Markov process. For $x, y \in \Sigma$, the probability that the ith character of S takes the value y is $p_{x \to y}$, where x is the value of the $(i-1)$th character in S. The stationary probability of observing the value y at the position i in S if we do not know the values of any other character in S is $p_{\Lambda \to y}$, where Λ is a special start character. We now define the concept of *Markov composition* of a word to identify words that will share the same probability $\tilde{P}(\xi_w)$ of occurring in S.

Definition 2. *The* 1st order Markov composition *of a string w is the multiset γ of transitions between consecutive characters in w, along with a transition from the start state Λ to the first character in w.*

Example 2. The 1st order Markov composition of the strings `AACAT` and `ACAAT` is $\gamma = \{(\Lambda \to \mathtt{A}), (\mathtt{A} \to \mathtt{A}), (\mathtt{A} \to \mathtt{C}), (\mathtt{A} \to \mathtt{T}), (\mathtt{C} \to \mathtt{A})\}$.

We let P_γ be equal to $\tilde{P}(\xi_w)$ for any word w with the Markov composition γ, we let $N_\gamma(x, y)$ represent the multiplicity of the transition $(x \to y)$ in the multiset γ, and we let $\Omega(\gamma)$ represent the number of words with the composition γ. Defining $\mathcal{C}_k^1$ to be the set of all 1st order Markov compositions for words of length k, we get the following result for $P(\zeta)$.

Theorem 3. *Let $S_1, \ldots, S_r$ represent r random strings of length n generated independently by a 1st order Markov process over the alphabet Σ. When Assumptions 1 and 2 hold, the probabilty $P(\zeta)$ that the strings $S_1, \ldots, S_r$ share a common substring of length k is defined by*

$$P(\zeta) = 1 - \prod_{\gamma \in \mathcal{C}_k^1} \left(1 - P_\gamma{}^r\right)^{\Omega(\gamma)} \ , \tag{7}$$

where the probability that a word w with composition γ is found in a random string S_i is $P_\gamma = 1 - \left(1 - \prod_{(x,y) \in \{\Sigma, \Lambda\} \times \Sigma} p_{x \to y}{}^{N_\gamma(x,y)}\right)^{n-k+1}$ and the number $\Omega(\gamma)$ of words that have the composition γ is defined below in Theorem 4.

Proof. The result again follows directly from Propositions 1 and 2. □

To compute an approximation of $P(\zeta)$ with Theorem 3, we still need to define a method for determining $\Omega(\gamma)$ and for iterating efficiently through the different Markov compositions in $\mathcal{C}_k^1$. We first turn to the problem of evaluating $\Omega(\gamma)$, and introduce a new structure that will help us in this task.

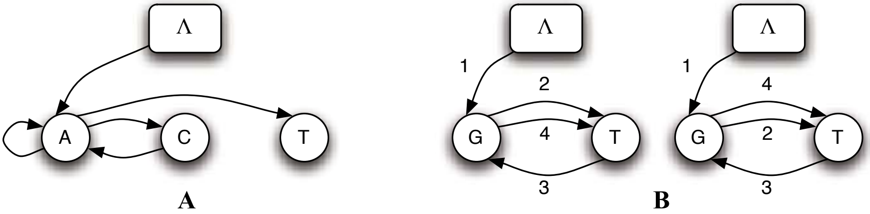


Fig. 1. (**A**) The 1st order Markov graph for the composition of the words AACAT and ACAAT, and (**B**) two distinct Eulerian trails that both represent the word GTGT

Definition 3. *The* 1st order Markov composition graph G_γ *for the Markov composition* γ *is the directed multigraph* $G_\gamma = \{V_\gamma, E_\gamma\}$*, where* V_γ *contains one vertex for every character in* $\Sigma \cup \{\Lambda\}$ *and* E_γ *contains one edge for every transition in* γ*.*

Example 3. The 1st order Markov composition graph for the composition of the strings AACAT and ACAAT is shown in Fig. 1A.

An Eulerian trail on the graph G_γ is a walk through the graph where we traverse each edge in E_γ exactly once. Each Eulerian trail on G_γ corresponds to a word w with composition γ, so we use some results on the enumeration of Eulerian trails on a directed multigraph to compute the number $\Omega(\gamma)$ of words with the composition γ.

Theorem 4. *The number of words with the Markov composition* γ *is*

$$\Omega(\gamma) = \frac{c_\gamma \cdot \prod_{v \in V_\gamma} (d(v) - 1)!}{\prod_{(u,v) \in V_\gamma^{\,2}} M(u,v)!} \ , \tag{8}$$

where $G_\gamma = \{V_\gamma, E_\gamma\}$ *is the Markov composition graph of* γ*,* c_γ *is the cofactor of* G_γ *(see [11]),* $d(v)$ *is the out-degree of the vertex* v*, and* $M(u,v)$ *is the number of edges going from* u *to* v *in* E_γ*.*

Proof. By the BEST theorem [21], the total number of Eulerian trails in the graph G_γ is $c_\gamma \cdot \prod_{v \in V_\gamma} (d(v) - 1)!$ (see [11]). This number overestimates the number of distinct words with composition γ since two trails that represent the same word are counted separately if the edges between two vertices $u, v \in V_\gamma$ are taken in different order (see Fig. 1 for an example). The result in (8) follows from the fact that there are $\prod_{(u,v) \in V_\gamma^{\,2}} M(u,v)!$ different ways to order the edges in a Eulerian trail without affecting the sequence of vertices in the trail. □

We now turn to the problem of efficiently enumerating all the different Markov compositions in $\mathcal{C}_k^1$. To do this, we first present the definition of *canonical* words.

Definition 4. *A word* w *with Markov composition* γ *is* canonical *if and only if no other word with composition* γ *is lexicographically inferior to* w*.*

Example 4. As we saw in Example 2, the words AACAT and ACAAT share the same 1st order Markov composition. Since AACAT $<_{lex}$ ACAAT and there are no other words with the same composition, the word AACAT is canonical while the word ACAAT is not.

We can enumerate the canonical word for each Markov composition using the following proposition.

Proposition 3. *The word w of length k with the* 1*st order Markov composition γ and ending with the character z is canonical if and only if its prefix of length $k-1$ is canonical and all the transitions in γ that go from z to another character are present in lexicographic order in w.*

We can enumerate all the canonical words of length k with a recursive algorithm by enumerating every canonical word of length $k-1$ and using Proposition 3 to determine which characters can be appended to these words. There are at most $|C_k^1|$ canonical words for each length $l \leq k$, and the test for each potential character to append can be accomplished in $O(k)$ time, so the entire recursive enumeration algorithm runs in $O(k^2|C_k^1|)$ time, assuming a constant alphabet size. The complexity of computation for P_γ and $\Omega(\gamma)$ also depends only on the alphabet size, so they can both be computed in $O(1)$ time when the alphabet size is constant. There are $(\sigma+1)\cdot\sigma$ different transitions possible in 1st order Markov models, so the size of $|C_k^1| \in O(k^{\sigma^2})$, and the running time of an algorithm that implements Theorem 3 is $O(k^2|C_k^1|) \in O(k^{\sigma^2+2})$.

4.3 Correcting for Auto-correlation

The *basic period* of a word w is the smallest positive integer i such that the word w overlaps with a copy of itself shifted by i positions to the right. Words with a small basic period are also the ones for which $\tilde{P}(\xi_w)$ gives a poor approximation. Let $\mathcal{W}_{k,c}$ represent the set of all words of length k with a basic period of at most c. A simple method for improving the approximation of $P(\zeta)$ obtained with (4) or (7) is to enumerate all the words w in $\mathcal{W}_{k,c}$ and to replace the inaccurate approximation $\tilde{P}(\xi_w)$ with the better approximations $P(\xi_w)$ for these words. Specifically, we let

$$P(\zeta) = 1 - \prod_{\gamma\in\mathcal{C}_k^m} \left(1 - {P_\gamma}^r\right)^{\Omega(\gamma)} \cdot \prod_{w\in\mathcal{W}_{k,c}} \left(\frac{1-P(\xi_w)^r}{1-\tilde{P}(\xi_w)^r}\right) . \tag{9}$$

There are $O(\sigma^c)$ words over the alphabet Σ of size σ, and the computation of $P(\xi_w)$, as we saw in Section 3, can be accomplished in $O(k)$. Therefore, the correction for the auto-correlation presented in (9) adds a factor of $O(k\sigma^c)$ to the running time of the algorithm. In practice, c does not need to be large to provide significant improvements to the estimates obtained with (4) or (7). In Section 6, we show that even a correction with $c = 1$ is enough to provide a significant improvement in accuracy.

5 Generalizations

The approach for solving the CSRS problem presented in this article can be generalized to handle many variations on the CSRS problem. For instance, we can immediately see that the above framework can be modified to handle random strings $S_1, \ldots, S_r$ that are generated by different random processes and have different lengths $n_1, \ldots, n_r$. We present two other useful generalizations below.

Common Substrings in *q* of *r* Random Strings. A common modification to the CSRS problem is to ask for the probability $P(\zeta_{q,r})$ that q of the r random strings $S_1, \ldots, S_r$ share a common substring. The equation for $P(\zeta_{q,r})$ can be obtained with the following straightforward modification of (2):

$$P(\zeta_{r,s}) = 1 - \prod_{w \in \Sigma^k} \left(1 - \sum_{j=q}^{r} \binom{r}{j} P(\xi_w)^j (1 - P(\xi_w))^{r-j}\right) . \tag{10}$$

A similar modification can also be applied to (4) and (7) for Bernoulli and Markov processes, respectively, with the running time of their corresponding algorithms increasing only by a factor of r.

Allowing Imperfect Matches. In many biologically realistic situation, one may want to allow a small number δ of mismatches in each occurrence of a word w in the random strings $S_1, \ldots, S_r$ [17]. Let $\Delta(w, \delta)$ represent the set of words that have at most δ mismatches to w. The probability that a word w occurs with at most δ mismatches in each of the strings $S_1, \ldots, S_r$ is

$$P(\xi_{w,\delta}) = 1 - \prod_{w' \in \Delta(w,\delta)} \left(1 - P(\xi_{w'})\right) . \tag{11}$$

This result can again be incorporated in (2), (4) and (7). The number of words in $\Delta(w, \delta)$ is $\sum_{i=0 \ldots \delta} \binom{k}{i} (\sigma - 1)^i \in O(k^\delta)$ when the size σ of the alphabet is constant, so the running time of algorithms that incorporate (11) is exponential in k. However, δ is generally quite small in practice so the algorithm remains practical.

6 Results

We tested the accuracy of our approximations for $P(\zeta)$ against a set of hit-or-miss Monte-Carlo simulations. For each configuration of parameters tested, we generated 1,500,000 independent sets of r random strings of length n and counted what fraction of the sets contained a common substring of length k. The number of trials was selected to give a maximum error of the true probability $P(\zeta)$ of ±0.0005, 99 times out of 100, according to the normal error bounds [9].

In Table 1, we compare different approximations to $P(\zeta)$ under a non-uniform Bernoulli model that simulates the distribution of nucleotides in the human genome ($p_A, p_T \approx 0.30$, and $p_C, p_G \approx 0.20$). For the different values of r we selected the string lengths n that give $P(\zeta) \approx 0.05$ and $P(\zeta) \approx 0.01$ to observe the quality of the approximations over the most significant range for statistical

Table 1. Approximations for the probability of observing a common substring of length $k = 6$ in r random strings of length n generated by the Bernoulli model representing the human genome ($p_A = 0.2962$, $p_C = 0.2037$, $p_G = 0.2035$, $p_T = 0.2966$)

r	n	(2)	(4)	(9)	K.O.	N.S.(a)	N.S.(b)	Simulation
2	18	0.0490	0.0491	0.0490	0.0368	0.0367	0.0600	0.0386
4	197	0.0497	0.0500	0.0497	0.0585	0.0322	0.0521	0.0477
6	467	0.0499	0.0508	0.0502	0.1041	0.0164	0.0543	0.0490
8	727	0.0500	0.0514	0.0505	0.2458	0.0075	0.0576	0.0493
10	958	0.0501	0.0519	0.0509	0.6065	0.0034	0.0609	0.0495
2	11	0.0107	0.0107	0.0107	0.0080	0.0079	0.0126	0.0088
4	131	0.0100	0.0101	0.0100	0.0111	0.0062	0.0104	0.0096
6	346	0.0099	0.0101	0.0100	0.0176	0.0029	0.0107	0.0098
8	569	0.0100	0.0103	0.0101	0.0384	0.0012	0.0113	0.0099
10	774	0.0100	0.0104	0.0102	0.1034	0.0005	0.0119	0.0100

Table 2. (**A**) Approximations for $P(\zeta)$ in the 1st order Markov model representing the human genome, when $k = 6$. (**B**) Approximation for $P(\zeta)$ when allowing imperfect matches ($k = 8$, $\delta = 1$) in the human Bernoulli model.

r	n	(2)	(7)	(9)	Simulation
2	17	0.0496	0.0497	0.0495	0.0388
4	175	0.0495	0.0522	0.0493	0.0474
6	410	0.0497	0.0549	0.0493	0.0486
8	633	0.0500	0.0607	0.0494	0.0493
10	828	0.0502	0.0674	0.0494	0.0493
2	10	0.0088	0.0088	0.0088	0.0073
4	117	0.0101	0.0105	0.0101	0.0098
6	304	0.0100	0.0113	0.0099	0.0099
8	494	0.0100	0.0128	0.0099	0.0099
10	666	0.0100	0.0148	0.0098	0.0099

A

r	n	(2), (11)	Simulation
2	9	0.0451	0.0176
4	68	0.0502	0.0417
6	187	0.0502	0.0452
8	315	0.0496	0.0446
10	436	0.0504	0.0443
2	8	0.0115	0.0050
4	47	0.0098	0.0085
6	141	0.0098	0.0091
8	251	0.0099	0.0090
10	358	0.0100	0.0088

B

analysis. As the results in Table 1 show, the approximation (2) obtained with the Independent Words Model is very accurate when $r > 2$ and converges to the true value of $P(\zeta)$ as the number of strings increases. The approximation (4) obtained with the Double Independence Model also offers a good approximation to $P(\zeta)$, although the correction of $P(\xi_w)$ for words with a basic period of 1 provided by (9) improves the approximation significantly. The table also shows that the approximation of Karlin and Ost (K.O.: (2.12) in [12]) and the first approximation of Naus and Sheng (N.S.(a): (8) in [14]) diverge significantly from the true value of $P(\zeta)$ when there are $r > 2$ strings. The alternative approximation of Naus and Sheng that incorporates exact results (N.S.(b): (14) in [14]) diverges more slowly from the value of $P(\zeta)$ as the number of strings increases, but is still not as accurate as our approximations.

We also tested our approximations on a 1st order Markov model that approximates the human genome. As we see in Table 2A, the Independent Model again offers an accurate approximation to $P(\zeta)$. However, in this case the approximation of the Double Independence Model diverges from the true value of $P(\zeta)$ as the number of strings increases and highlights the importance of the correction of Section 4.3. With the correction of $P(\xi_w)$ for words with a basic period of 1, we see a significant improvement in the approximation accuracy.

In Table 2B, we show the results of the approximation for $P(\zeta)$ for words of length $k = 8$ when a mismatch is allowed ($\delta = 1$) for each occurrence of a word in a string, under the Bernoulli model of the human genome described above. We see that the approach described in (11) provides an acceptable approximation to the true value of $P(\zeta)$ in this model, although the values diverge slowly as the number of strings increases.

7 Discussion and Future Work

We propose two new methods for approximating the probability $P(\zeta)$ that r random strings contain a common substring of length k. The approximation obtained under the Independent Words Model offers an accurate estimate for $P(\zeta)$ when $r > 2$ and works well on both the Bernoulli and Markov models. The approximations obtained under the Double Independence Model are also quite accurate when the correction for auto-correlation proposed in Section 4.3 is applied, and can be computed in polynomial time (relative to k). Both methods are shown to be more accurate than previously published approaches on random strings generated by a non-uniform Bernoulli model.

Over all the configurations of parameters tested, we find that our approximations are always slightly conservative. This is to be expected, since the most important dependency that is ignored with Assumption 1 is the positive correlation between the events ξ_w and $\xi_{w'}$ for words w and w' that overlap each other. Importantly, the fact that our estimates are always conservative means that a user who applies our approximation to assess the significance of a common substring will never be mislead into thinking that a result is more significant than it actually is.

The approach presented in this article lends itself to a number of biologically important generalizations, in particular those allowing for the substrings to be found in only a subset of the strings and allowing for imperfect matches.

There are many areas in which the research presented in this article could be further developed. Significantly, theoretical bounds on the error induced by the assumptions in our models would be very useful. Development of new methods that weaken the assumptions presented in this article could also lead to interesting and more accurate approximation algorithms.

Acknowledgments

The authors wish to thank Uri Keich for helpful discussions, and the anonymous referees for many insightful and valuable comments.

References

1. S. F. Altschul, W. Gish, W. Miller, E. W. Myers and D. J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215:403-410, 1990.
2. S. F. Altschul, T. L. Madden, A. A. Schaffer, J. Zhang, Z. Zhang, W. Miller and D. J. Lipman. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic Acids Research*, 25:3389-3402, 1997.
3. R. Arratia and M. S. Waterman. An Erdős-Rényi law with shifts. *Advances in Mathematics*, 55:13-23, 1985.
4. R. Arratia and M. S. Waterman. Critical Phenomena in sequence matching. *The Annals of Probability*, 13:1236-1249, 1985.
5. E. Blais. *Computing Probabilities for Common Substrings in Random Strings.* M. Sc. Thesis, McGill University, 2006.
6. P. Erdős and A. Rényi. On a new law of large numbers. *Journal d'Analyse Mathématique*, 22:103-111, 1970.
7. P. Erdős and P. Révész. On the length of the longest head run. *Topics in Information Theory, Coll. Math. Soc. János Bolyai No. 16*, 219-228, 1975.
8. W. Feller. *An Introduction to Probability Theory and its Applications*, Volume 1 (3rd Edition), John Wiley & Sons, 1968.
9. G. S. Fishman. *Monte Carlo: Concepts, Algorithms, and Apps.*, Springer, 1996.
10. L. J. Guibas and A. M. Odlyzko. String overlaps, pattern matching, and nontransitive games. *Journal of Combinatorial Theory, Series A*, 30:183-208, 1981.
11. F. Harary. *Graphical Enumeration*, Academic Press, 1973.
12. S. Karlin and F. Ost. Maximal length of common words among random letter sequences. *The Annals of Probability*, 16:535-563, 1988.
13. B. Morgenstern, K. Frech, A. Dress and T. Werner. DIALIGN: Finding local similarities by multiple sequence alignment. *Bioinformatics* 14:290-294, 1998.
14. J. Naus and K.-N. Sheng. Matching among multiple random sequences. *Bulletin of Mathematical Biology*, 59:483-496, 1997.
15. P. Nicodème, B. Salvy and P. Flajolet. Motif statistics. *Theoretical Computer Science*, 287:593-617, 2002.
16. A. Nijenhuis and H. Wilf *Combinatorial Algorithms for Computers and Calculators*, Academic Press, 1978.
17. P. A. Pevzner and S. Sze. Combinatorial approaches to finding subtle signals in DNA sequences. *Proc. 8th Inter. Conf. on Int. Sys. for Mol. Biol.*, 269-278, 2000.
18. M. Régnier. A unified approach to word occurrence probabilities. *Discrete Applied Mathematics*, 104:259-280, 2000.
19. M. Régnier and W. Szpankowski. On pattern frequency occurrences in a Markovian sequence. *Algorithmica*, 22:631-649, 1998.
20. S. Sinha and M. Tompa. Discovery of novel transcription factor binding sites by statistical overrepresentation. *Nucleic Acids Research*, 30:5549-5560, 2002.
21. T. van Aardenne-Ehrenfest and N. G. de Bruijn. Circuits and trees in oriented linear graphs. *Simon Stevin*, 28:203-217, 1951.
22. J. van Helden, B. André and J. Collado-Vides. Extracting regulatory sites from the upstream region of yeast genes by computational analysis of oligonucleotide frequencies. *Journal of Molecular Biology*, 281:827-842, 1998.

On the Repeat-Annotated Phylogenetic Tree Reconstruction Problem

Firas Swidan[1,2,⋆], Michal Ziv-Ukelson[1,3,⋆⋆], and Ron Y. Pinter[1]

[1] Department of Computer Science, Technion – Israel Institute of Technology,
Haifa 32000, Israel
{firas, michalz, pinter}@cs.technion.ac.il
[2] Janelia Farm, Howard Hughes Medical Institute, VA 20147, USA
swidanf@jfrc.hhmi.org
[3] School of Computer Science, Tel-Aviv University, Tel-Aviv 69978, Israel
michaluz@post.tau.ac.il

Abstract. A new problem in *phylogenetic inference* is presented, based on recent biological findings indicating a strong association between *reversals* (aka *inversions*) and *repeats*. These biological findings are formalized here in a new mathematical model, called *repeat-annotated phylogenetic trees* (RAPT). We show that, under RAPT, the evolutionary process — including both the tree-topology as well as internal node genome orders — is *uniquely* determined, a property that is of major significance both in theory and in practice. Furthermore, the repeats are employed to provide linear-time algorithms for reconstructing both the genomic orders and the phylogeny, which are NP-hard problems under the classical model of *sorting by reversals* (SBR).

1 Introduction

Phylogenetic inference and ancestral genome order reconstruction are important problems in evolutionary, genetic, and bioinformatic studies. In these problems one seeks to reconstruct the phylogeny of a given set of organisms as well as the genomic order (*i.e.*, the order of the genomic segments) of their ancestors; see for example Figures 1a and 1c. Here, a one-to-one mapping of the orthologous segments of the two strains *Xanthomonas campestris* pathovar *campestris* ATCC 33913 (*X. campestris*) and 8004 (*X. campestris* 8004) is presented schematically. These bacteria cause black rot disease in crucifers such as *Brassica* (cabbage) and *Arabidopsis* (mustard), which results in severe losses in agricultural yield worldwide. This figure suggests that 3 reversals have affected the two bacteria since their divergence. However, during the speciation of which of the two bacteria have these reversals occurred and what is the ancestral genomic order?

Using current methods, reconstructing ancestral genomic order involves solving a multiple *sorting by reversals* (SBR) problem. In SBR, which has been

[⋆] Was supported in part by a fellowship from the Planning and Budgeting Committee of the Council for Higher Education in Israel.

[⋆⋆] Was supported in part by the Aly Kaufman Post Doctoral Fellowship.

M. Lewenstein and G. Valiente (Eds.): CPM 2006, LNCS 4009, pp. 141–152, 2006.

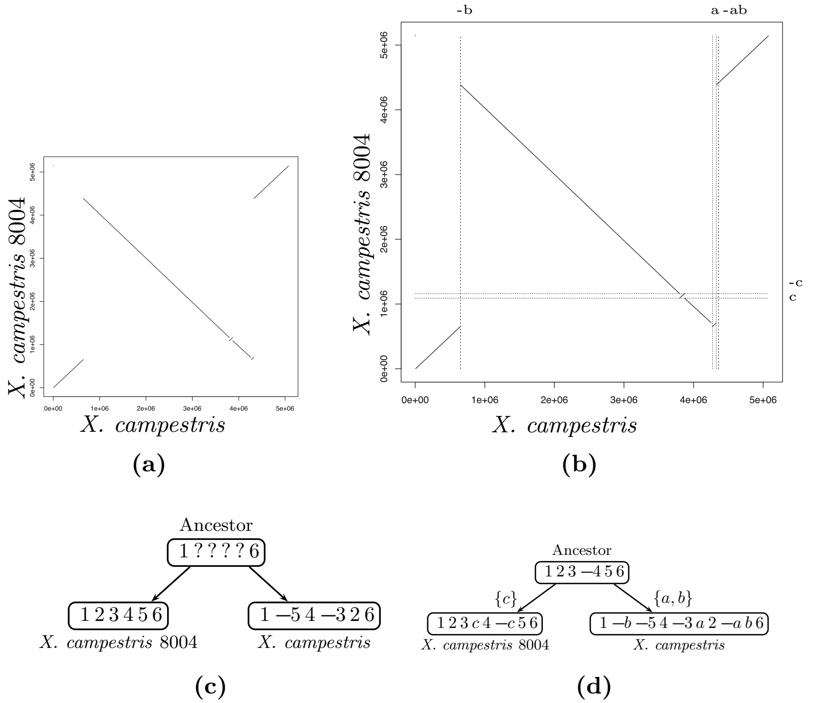


Fig. 1. Inferring the order of genomic segments in the ancestor of the bacteria *X. campestris* and *X. campestris* 8004. (a) The comparative mapping is the result of applying MAGIC [1] to the considered organisms. MAGIC was run with its default parameters, except for discarding segments of length smaller than 10000bp. The lines in the figure represent corresponding rearrangement-free segments (*i.e.*, segments that did not undergo rearrangements) in the two bacteria (since the genomes are circular, the square drawing should be wrapped into a torus). The mapping suggests that 3 reversals have occurred since the divergence of the two bacteria, in agreement with the observations made by Qian *et al.* [2]. (b) Incorporating the repeats that were obtained by applying Repseek [3] to each genome. The repeats are represented by dashed lines and marked by $-a$, a, $-b$, b, and $-c$, c. All the segments $\pm a$, $\pm b$, and $\pm c$ have a high translated sequence similarity (at the amino acid level) to the insertion sequence IS1478. (c) The phylogeny and the permutations corresponding to the mapping in (a). The permutation of *X. campestris* 8004 is chosen to be the identity permutation. One cannot infer the ancestral genomic order or decide during the speciation of which of the bacteria the reversals have occurred. (d) Applying the new approach which consists of including the repeats in the permutations and using them to annotate the edges in the phylogeny.

thoroughly examined over the last two decades [5, 6, 7, 8, 9, 10, 11, 12], one represents the one-to-one orthologous mappings as permutations and the inversion mutations as reversal operations. The goal of the SBR optimization problem is to find a phylogeny and corresponding reversal scenarios along its branches with the minimum number of reversals. In SBR, however, the ancestral genomic order cannot be implied based solely on the comparison of a pair of genomes, as is needed for the bacteria pair in Figure 1a. Moreover, adding one more organism to enable the deduction of the ancestral order makes this task NP-hard [13]. Nonetheless, this problem was addressed by both exhaustive search and heuristic techniques [14, 15].

The ancestral genome order reconstruction problem described above extends to reconstructing large phylogenetic trees over multiple input leaves (including or excluding the recovery of genomic order in internal nodes). Current phylogenetic inference methods, based on different biological evidence ranging from phenotypic morphologies to various genotypic mutations — including point mutations (distance-based, maximum-likelihood, and parsimony approaches), gene insertions and deletions (the Dollo parsimony approach), and genome rearrangements (distance-based and parsimony approaches) — are computationally hard. Moreover, current approaches often yield many alternative solutions; choosing the most sound phylogeny among them has very important biological consequences, but is yet a very challenging task.

In this paper we investigate a new approach to phylogenetic inference and ancestral genome order reconstruction. The new approach is inspired by a recent biological discovery regarding the role of *repeats*, *i.e.*, short genomic sequences that are highly similar to each other, in inducing reversals (or recombinations in general). Several studies indicate a strong association between repeats and recombination events. As reviewed in [4] , repeats cause rearrangements, either by a mechanism of *illegitimate recombination*, or by a mechanism of *homologous recombination*. Moreover, these findings demonstrate that most of the repeats engaged in reversals correspond to *mobile* DNA elements, *i.e.*, regions of DNA that selfishly duplicate and move into new sites. Hence, these repeats are usually found only in the organism affected by the reversals; see, *e.g.*, [2] . This new and important information regarding the repeats became accessible recently with the availability of many sequenced genomes and its automatic generation is made possible by the development of accurate comparative genome mapping methods, see, *e.g.*, [1, 2]. The new data motivates the enhancement of previous phylogenetic models with additional information in the form of repeat "footprints", in order to make their predictions more realistic and increase their potential for producing biologically relevant insights.

Qian *et al.* [2] demonstrate an initial utilization of repeats in the task of ancestral genome order reconstruction of the *Xanthomonas campestris* bacteria. They have identified two identical IS1478-related insertion sequences (corresponding to the repeat pair $-b, b$ in Figure 1b) spanning a putative recombination site. Moreover, they predicted a rearrangement scenario for transforming one genome to the other — see `http://www.genome.org/content/vol0/issue2005/images/data/gr.3378705/DC1/SI_Fig_2.gif` for a detailed (and vivid) animation of their prediction. We continue their analysis by applying our approach to the very same data: in Figures 1b and 1d we incorporate the information regarding the repeats into the mapping. In addition to the repeat pair reported by Qian *et al.*, we identify two more pairs spanning two putative recombination sites.

According to the repeats, one inversion occurred during the speciation of *X. campestris*, while two inversions occurred during the speciation of *X. campestris* 8004. Given this information, deducing the ancestral genomic order is straightforward, as shown in Figure 1d.

The above example demonstrates how repeats can be utilized to uniquely determine the order of the ancestral genomic segments — based solely on pairwise genomic comparisons. Furthermore, it shows that the "repeat footprints" aid in the efficient computation of ancestral genomic orders. To generalize these observations, in Section 2 we formalize the biological assumptions introduced above into a theoretical evolutionary model. In Section 3 we study the pairwise case and present two important results: uniqueness of the solutions and simplification of the computation. In Section 4 we show that these two results scale up to the more general case of multiple genomes. For a formal overview of the combinatorial results of this paper we refer the reader to Section 2.1. *Due to space limitations, this extended abstract contains only a high-level overview of our results. The full version of this manuscript, including all lemmas and their proofs as well as additional figures can be found at* `http://magicmapping.sourceforge.net/download/repeatPhylos.pdf`.

2 A Formal Model Based on Repeats

The model described in this paper is based on the following biological assumptions:

1. Reversals are usually induced by *inverted* repeat pairs, *i.e.*, repeats having opposite orientations [4] .
2. Repeats engaged in reversals — corresponding mostly to mobile DNA elements — are easily identified on the borders of reversed genomic segments, and are present only in the affected organism [2].
3. The information mapping each repeat to its pair-mate is part of the input[1].
4. Each repeat has a very low probability for causing a reversal that remains fixed in the population [1]. Therefore, in our model we assume that each repeat causes up to one reversal.

Note that though the above assumptions may not capture the great variety found in real biological problems, it is easy to check if a given set of input genomes follows these assumptions. Furthermore, as demonstrated by the theoretical results listed in Section 2.1, the assumptions above offer a solid basis for potential future extensions and enhancements.

Based on Assumption 3, the input to our problem comprises sequences, referred to as *repmaps*, of both permutation elements, belonging to a set N, and paired repeats, belonging to a set R. Each permutation element appears exactly once in the repmap, while each repeat appears exactly twice. In addition to the

[1] This information can be obtained using techniques similar to those standardly used for preparing permutations.

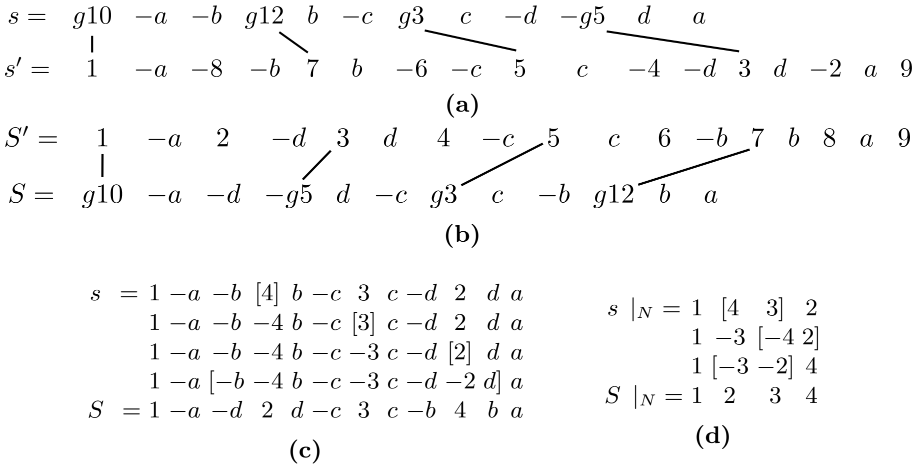

$s =$ $g10$ $-a$ $-b$ $g12$ b $-c$ $g3$ c $-d$ $-g5$ d a
$s' =$ 1 $-a$ -8 $-b$ 7 b -6 $-c$ 5 c -4 $-d$ 3 d -2 a 9
(a)
$S' =$ 1 $-a$ 2 $-d$ 3 d 4 $-c$ 5 c 6 $-b$ 7 b 8 a 9
$S =$ $g10$ $-a$ $-d$ $-g5$ d $-c$ $g3$ c $-b$ $g12$ b a
(b)
s $=1\ -a\ -b\ [4]\ b\ -c\ 3\ c\ -d\ 2\ d\ a$
$1\ -a\ -b\ -4\ b\ -c\ [3]\ c\ -d\ 2\ d\ a$
$1\ -a\ -b\ -4\ b\ -c\ -3\ c\ -d\ [2]\ d\ a$
$1\ -a\ [-b\ -4\ b\ -c\ -3\ c\ -d\ -2\ d]\ a$
S $=1\ -a\ -d\ 2\ d\ -c\ 3\ c\ -b\ 4\ b\ a$
(c)
$s\ |_N = 1\ [4\ 3]\ 2$
$1\ -3\ [-4\ 2]$
$1\ [-3\ -2]\ 4$
$S\ |_N = 1\ 2\ 3\ 4$
(d)

Fig. 2. Calculating the ancestor assignment (a-b) and comparing a legal scenario to an SBR scenario (c-d). (a) Example of transforming a repmap s to a normalized repmap s . The correspondence between the permutation elements of s and those of s is drawn as edges connecting between the respective elements. The permutation elements in s are given over a different alphabet. (b) Determining the ancestor in normalized format S and in input format S from s . In both (c) and (d) we assume that the ancestor S is known. We rename s and S to s and S to enable running SBR on them and we compare a legal scenario (c) to an SBR scenario (d). The reversals are denoted by brackets. Note that the reconstructed scenarios are different, since one of them is guided by fulfilling the constraints imposed by the repeats, while the other aims to minimize the number of reversals. If s is a normalized repmap then a scenario is legal iff it is SBR.

permutation elements (represented by numbers) and based on Assumption 1, the repeats (represented by lowercase characters) are also signed.

Given a repmap s, two repeat elements $s_i, s_j \in R$ are considered a pair if $|s_i| = |s_j|$ (*i.e.*, their absolute values are equal). If they have opposite signs ($s_i = -s_j$) we refer to them as an *inverted repeat pair*; otherwise they are called a *direct repeat pair*. The set of repeats appearing in s is denoted by $R(s) = \{|s_i|, s_i \in R\}$ and referred to as the *repeat set*. We denote the restriction of s to the permutation elements in N by $s|_N$ and refer to it as the *induced permutation*. The restriction of s to the repeat elements is denoted by $s|_R$ and is referred to as the *repeat subsequence*.

Example. Consider the repmap $s = 1\ a\ -4\ -a\ -b\ 3\ 2\ -b\ 5$. Here, $+a, -a$ is an inverted repeat pair, while $-b, -b$ is a direct repeat pair. Moreover, we have $R(s) = \{a, b\}$. The induced permutation is $s|_N = 1\ -4\ \ 3\ \ 2\ 5$, and the repeat subsequence is given by $s|_R = a\ -a\ -b\ -b$.

The next three definitions are intended to formalize the biological assumptions (1-4) into a mathematical model of an evolutionary process. The first definition is based on Assumption 1, as follows.

Definition 1 (Legal Reversal). *Let $s = s_1, \ldots, s_n$ be a repmap and let $\rho = \rho(i, j)$ for $1 < i < j < n$ be a reversal affecting the subsequence $s_i, \ldots, s_j$ in s. The reversal ρ is called* legal *if it is bordered by an inverted repeat pair, i.e., if $s_{i-1} = -s_{j+1}$ (see Figure 2c). We say then that ρ* fulfills *the repeat pair s_{i-1}, s_{j+1}.*

The next two definitions are both based on Assumptions 2 and 4.

Definition 2 (Legal Scenario). *Given a reversal sequence $\varrho = \rho_1, \ldots, \rho_m$ affecting s, we say that ϱ is a* legal scenario relative to a subset of repeat pairs *$\mathscr{R} \subseteq R(s)$ if $\forall i \in \{1, \ldots, m\}, \rho_i$ is a legal reversal when acting on $s \cdot \rho_1 \cdots \rho_{i-1}$ and if ϱ fulfills each repeat in $\mathscr{R}$ exactly once (see Figure 2c). If $\mathscr{R} = R(s)$, we refer to ϱ simply as a* legal scenario*. If $\mathscr{R} \neq R(s)$ is obvious from the context, we refer to ϱ as a* partially legal scenario*.*

Definition 3 (RAPT). *Given a repmap S (ancestor), a* Repeat-Annotated Phylogenetic Tree *originating in S (see Figure 3) is a triplet (T, f, g), where $T = (V, E)$ is a directed tree with root $v_r \in V$ such that all the inner nodes (except perhaps the root) are of degree ≥ 3, $f : V \rightarrow (R \cup N)^*$ maps* assignments *(i.e., repmaps) to the nodes, and $g : E \rightarrow 2^{R(S)}$ maps* labels *to the edges, such that:*

1. *The edge labels are a partition of $R(S)$, i.e., for every two edges $e, e' \in E$: $g(e) \cap g(e') = \emptyset$ and $\bigcup_{e \in E} g(e) = R(S)$.*
2. *The assignments to the nodes fulfill the following two requirements:*
   (a) *The assignment to the root v_r equals S, that is $f(v_r) = S$.*
   (b) *Assuming $u \in V$ is the immediate parent of $v \in V$ and that $e \in E$ is the edge connecting them, we require that $g(e) \subset R(f(u))$ and that there exists a legal scenario $\rho_1, \ldots, \rho_m$ with respect to $g(e)$ such that $(f(u) \cdot \rho_1 \cdots \rho_m)|_N = f(v)|_N$ (Definition 2).*
3. *The repeat set $R(s)$ of a leaf repmap s contains only repeats that engaged in reversals at some point during the history of s, i.e., $R(s) = g(\text{path}(v_r, s))$.*

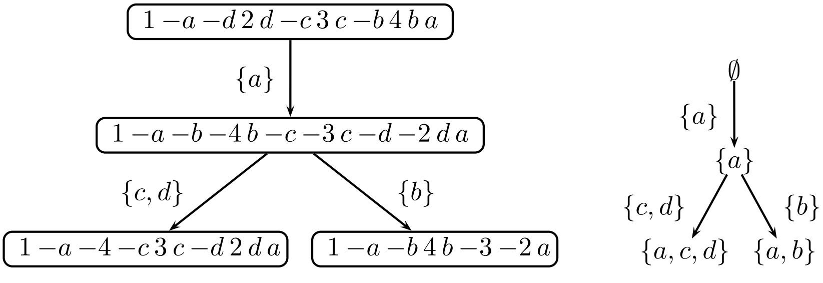
2.1 The Main Results of This Paper

In this paper we study the following problems: Can one reconstruct an unknown RAPT (T, f, g) given, as input, a set L of the corresponding leaf repmaps? More specifically, does L uniquely determine the RAPT? Are the legal scenarios linking the assignments in the RAPT nodes unique? Furthermore, can one efficiently reconstruct the unknown RAPT and the corresponding scenarios? Herein we summarize the answers to these questions.

First, in Section 3, we consider the basic case in which the tree T of the RAPT contains a single leaf and a single ancestor. Since in this case both the tree T and the labels g are trivially determined, our results pertain to both the scenario and the ancestral assignment reconstructions, as follows:

Uniqueness: We show that the ancestral assignment is uniquely determined (Section 3.1). This result is surprising given the ambiguity of the scenarios.

Complexity: We give a linear-time algorithm for reconstructing the ancestor (Section 3.2). Contrary to the classical SBR problem, our algorithm utilizes



(a) **(b)**

Fig. 3. An example of a RAPT (a) and the corresponding set-trie (b). The set-trie is obtained from the RAPT by preserving both the tree topology as well as the edge labels of the RAPT, while discarding the node assignments. The virtual node assignments of the set-trie are determined from the edge labels — see Definition 5.

the constraints introduced by the repeats to calculate the unique ancestor. This algorithm is then employed to solve the problem of reconstructing a plausible legal scenario, by a reduction to SBR (Section 3.2).

Next, in Section 4, the multiple leaf RAPT is studied. Based on the results obtained for the single leaf case, it is straightforward to show that, in the multiple leaf case, the tree topology T, the edge labels g, and the leaf repmaps L both uniquely determine the induced permutations in the inner node assignments and also enable their reconstruction in linear-time. Hence, the complexity and uniqueness issues are reduced to the pair (T, g). To investigate the latter, we introduce a new data structure, which is an abstraction of such (T, g) pairs, called *set-tries* (see Figure 3), which are trie-like structures over sets instead of words (Section 4.1). In terms of this abstraction, our results can be formulated as follows:

Uniqueness: We show that the leaf set collection uniquely determines the underlying set-trie.

Complexity: We give a linear-time algorithm to efficiently reconstruct set-tries from an input leaf set collection.

3 The Single Leaf RAPT

Throughout this section, we assume without loss of generality that the repmaps are given in an easy to handle format, as follows. Consider a repmap (ancestor) S to which a legal scenario ϱ was applied and denote the result by $s = S \cdot \varrho$ (the notation of S denoting the ancestor and s denoting the descendant is used consistently throughout this section). We assume that S (and hence s) starts and ends with a permutation element (otherwise it can be padded). In addition, we assume that S (and hence s) does not contain successive permutation elements (or otherwise they can be united to form a single new permutation element).

$$\begin{tikzcd}
S \arrow[r, "normalize"] \arrow[d, shift right, "\varrho"'] & S' \arrow[d, shift right, "\varrho"'] \\
s \arrow[r, "calculate"] \arrow[u, shift right, "\varrho'"'] & s' \arrow[u, shift right, "\varrho'"']
\end{tikzcd}$$

Fig. 4. Given that $s = S \cdot \rho$, it is convenient to consider S , the *sorted* normalization of S, and $s = S \ \cdot \rho$: Proving that a legal scenario ϱ acting on s is also an SBR scenario implies the uniqueness of the ancestor. Calculating s directly from s enables the reconstruction of S in linear-time. Finally, finding an SBR scenario sorting $s\,|_N$ gives a legal scenario on s.

Whereas uniting successive permutation elements into a single element is straightforward, dealing with successive repeat elements in the input sequence is more challenging. For instance, having successive repeats implies that the corresponding breakpoints may have been reused (the issue of breakpoint reuse has been repeatedly debated in the literature and is currently controversial). From a modeling point of view, however, successive repeats distinguish the RAPT problem from the SBR problem: when they are present in a repmap, its set of legal scenarios (see Definition 2) and the set of SBR scenarios [5] of its induced permutation are not necessarily the same, as demonstrated in Figure 2. This is due to the fact that SBR aims to minimize the number of reversals, whereas RAPT is driven by the objective of fulfilling the constraints imposed by the repeats. Still, for a subset of the repmaps, both sets of legal and SBR scenarios are equal. We refer to these special repmaps as "normalized" and define them below. Normalized repmaps serve as stepping stones for our study; see Figures 2 and 4.

Definition 4 (Normalized Repmap). *A repmap S is a* normalized repmap *if between every two repeats in it there is a permutation element from N.*

3.1 Asserting Uniqueness of Ancestor

In this section we prove that all legal scenarios lead to the same ancestral repmap. This result is surprising, given the richness of the set of all legal scenarios . We first consider the special subclass of normalized repmaps. In this subclass, the proof of uniqueness involves a breakpoint counting argument, showing that all legal scenarios are optimal sorting scenarios (namely SBR scenarios). Next, we extend the uniqueness claim from the subset of normalized repmaps to the general repmap case. The proof here is achieved by transforming any given repmap to a corresponding normalized one and by asserting that this transformation indeed preserves the uniqueness property.

Theorem 1 (Uniqueness). *Let S be a repmap, ϱ a legal scenario, and $s = S \cdot \varrho$. Then, all legal scenarios affecting s result in the same correct ancestor S.*

3.2 Algorithms for Ancestor and Scenarios Reconstruction

Given a repmap $s = S \cdot \varrho$, where S and ϱ are unknown, we present a linear-time algorithm for reconstructing S and a sub-quadratic algorithm for reconstructing a possible legal scenario ϱ', where, by Theorem 1, $S = s \cdot \varrho'$. The reconstruction of the ancestor in linear-time is made possible by utilizing the constraints introduced by the repeats and the strong connection established between the repeats and their surroundings in normalized repmaps. In fact, we first show how to transform s to a normalized repmap s' for which the ancestor S' is sorted[2] based only on the repeat subsequence $s|_R$ (see Figure 4). Then we simply rename the matching elements from S' to obtain S (see Figure 2b). The transformation to a normalized format is done based on the repeats in s and without knowing the ancestor repmap S.

After computing the ancestor S, we solve the problem of finding a legal scenario transforming s to S in sub-quadratic time by a reduction to SBR. In the general case, as exemplified in Figures 2c and 2d, applying SBR to $s|_N$ may yield illegal scenarios. This is due to the fact that SBR aims to minimize the number of reversals, while RAPT is driven by the objective of fulfilling the constraints imposed by the repeats. However, this barrier is overcome here by transforming a repmap to its normalized format, which intuitively uses $O(|s|)$ additional "virtual" permutation elements to simulate the constraints imposed by the repeats (see Figures 2a and 2b). Thus, to reconstruct a legal scenario, we apply SBR algorithms to the permutation elements of s' and S' and show that the resulting scenario is legal on s. Note that whereas reconstructing the ancestor in linear-time is made possible thanks to the constraints introduced separately by each repeat pair, calculating a legal scenario is complicated by the interaction between the constraints introduced by the different repeat pairs.

Reconstructing the Unique Ancestral Repmap S. Reconstructing the ancestral repmap S can be naïvely achieved by applying some of the techniques demonstrated in [6, 12] to the *overlap graph* constructed over the repeat pairs. This approach, however, yields a quadratic-time algorithm for both reconstructing the ancestor and finding a legal scenario.

Here, we present a different approach (with lower complexity) for tackling the problem. Let S be an ancestral repmap, ϱ a legal scenario affecting S, and $s = S \cdot \varrho$. Consider a transformation of S yielding a normalized sorted repmap S', and let $s' = S' \cdot \varrho$. According to the above and since all the transformations are reversible, calculating s' from s can be done by the following series of transformations: $s \longrightarrow S \longrightarrow S' \longrightarrow s'$. However, since S is unknown, this path is intractable. Yet, surprisingly, calculating s' from s can alternatively be done based on the repeat sequence $s|_R$ and without knowing S. Intuitively, this can be explained as follows. Since s' is normalized, the locations of the permutation elements are constrained by the locations of the repeats. Moreover, each permutation element is constrained by the repeat next to it. Thus, the position of a permutation element can be determined from local information (the position of a single repeat) in constant time, and the whole repmap can be reconstructed in linear-time. Note that the above transformation implies that the diagram having S, S', s, and s' as its vertices is commutative (Figure 4).

[2] Note that, unlike the previous section, here we can no longer assume without loss of generality that S is sorted.

Example. Consider the sequence s and the corresponding sequence s' in Figure 2a and suppose that we wish to recover s' based on $s|_R$. The first and second elements in s' are easily fixed, since s' must always start with 1 and the second element in s' always equals to the repeat appearing in the second position of s. Fixing the third element in s' is more challenging. For that, we consider the second element in both s and s', *i.e.*, the repeat $-a$, and its corresponding pair-mate — the repeat a. Since S' is sorted and normalized, once the repeat pair $\{-a, a\}$ got fulfilled while transforming S' to s', the surroundings of both repeats remain contiguous. In particular, the permutation element 2, which appears directly after the repeat $-a$ in S' (see Figure 2b) must appear in the surrounding of the repeat a in s'; its exact position (*i.e.*, before or after the repeat) is determined based on two factors: whether the repeat pair is inverted or direct, and whether the preceding permutation element 1 appears before the repeat $-a$ or after it. In this example, since the repeat pair is inverted and since the preceding permutation element 1 appears before the repeat $-a$, the permutation element 2 must precede the repeat a in s'. The sign of the permutation element 2 is determined via a similar consideration.

Theorem 2 (Time Complexity). *Given a repmap $s = S \cdot \varrho$, one can reconstruct the ancestor S in linear time ($O(|s|)$).*

Reconstructing a Legal Scenario. Unlike the ancestor repmap reconstruction, the scenario reconstruction involves look-ahead to avoid conflicts between repeat pairs. This problem is best demonstrated by an example.

Example. Consider the following repmap: $1\ a\ {-b}\ {-2}\ {-a}\ {-c}\ {-4}\ b\ 3\ c$. Suppose we were first to fulfill the inverted repeat pair b and $-b$. Such a choice would turn the other two repeat pairs (a and c) into direct-repeat pairs. Thus, we reach a deadlock without getting a legal scenario.

As demonstrated in the above example, choosing a legal reversal sequence that avoids deadlocks is a delicate matter. We address this problem by utilizing the fact that we can calculate the normalized repmaps s' and S' in linear-time (Section 3.2). When both repmaps are known, we show that an SBR reversal sequence sorting $s'|_N$ (to $S'|_N$) corresponds to a legal scenario transforming s to S. Currently, the best algorithm for solving SBR works in sub-quadratic time [9]. Hence, we get a sub-quadratic algorithm for reconstructing a legal scenario.

Theorem 3 (Time Complexity). *Given a repmap $s = S \cdot \varrho$, where both the repmap S and the legal scenario ϱ are unknown, one can reconstruct a legal scenario transforming s to S in $O(n\sqrt{n \log n})$ time, where $n = |s|$.*

4 The Multiple Leaf RAPT and Set-Tries

In this section we show that the leaf assignments $L = \{s^1, \ldots, s^q\}$ uniquely determine the underlying RAPT (T, f, g) up to (and not including) repeats in the inner nodes, *i.e.*, they dictate the tree topology, the induced permutations in inner node assignments, and the edge labels. We then describe a linear-time algorithm for reconstructing this information from the given input.

The proof of uniqueness is developed in two stages: first, the RAPT is reduced to a new auxiliary data structure called a *set-trie* (see Section 4.1 and Figure 3), which encodes partial information (tree topology and edge labels). Using this reduction, we show that both the tree topology and the edge labels are uniquely determined and can be reconstructed in linear-time based on the repeat sets $\{R(s) : s \in L\}$ of the leaf assignments. Finally, the application of Theorems 1 and 2 to the above findings leads to the conclusion that the induced permutations in the inner node assignments are uniquely determined and can be reconstructed in linear-time based on the tree topology, the edge labels, and the leaf assignments.

4.1 Set-Tries and Monotonic Collections

Word-tries are well-known data structures, commonly used in text compression and database search [16]. They are used to store the information about the contents of each node in the path from the root to the node rather than in the node itself, thus grouping words with a common prefix along similar paths. Here we introduce a new data structure which, similarly to word-tries, is also based on a tree topology and path-encoding, however, the leaves of the new data structure correspond to sets instead of words (or sequences), as defined below.

Definition 5 (Set-tries). *Let $\mathscr{A} = \{A_1, \dots, A_k\}$ be a collection of finite subsets of* $\mathbb{N}$. *A* set-trie *st over $\mathscr{A}$ is a pair $st = (T, g)$, where $T = (V, E)$ is a directed tree with a root v_r such that all the inner nodes (except perhaps the root) are of degree ≥ 3 and $g : E \rightarrow 2^{\mathbb{N}}$ are* labels *to the edges. In the following discussion we assume that* assignments *to the nodes $f : V \rightarrow 2^{\mathbb{N}}$ are also given. The labels g and the "virtual" assignments f need to fulfill the following requirements:*

1. *$f(v_r) = \emptyset$ and f is $1:1$ from the leafs of T to $\mathscr{A}$. Given that $u \in V$ is an ancestor of $v \in V$, we require that $f(v) = f(u) \cup g(\mathrm{path}(u, v))$. In particular, this requirement implies $\forall v \in V - \{v_r\} : f(v) = g(\mathrm{path}(v_r, v))$.*
2. *$\forall e, e' \in E, e \neq e' : g(e) \cap g(e') = \emptyset$. Thus, the node assignments are determined by the edge labels and vice versa.*

Figure 3 gives an example of a set-trie and its derivation from a RAPT. We observe the following *monotonicity* property of set collections corresponding to leafs of set-tries.

Definition 6 (Monotonic Set Collection). *A set collection $\mathscr{A}$ is* monotonic *if, for any three sets $A, B, C \in \mathscr{A}$, either $A \cap B \subseteq A \cap C$ or $A \cap C \subseteq A \cap B$.*

Theorem 4 (Time Complexity). *Given a monotonic collection $\mathscr{A}$, a set-trie over $\mathscr{A}$ can be constructed in linear-time ($\Theta(|\mathscr{A}|)$).*

References

1. Swidan, F., Rocha, E.P.C., Shmoish, M., Pinter, R.: An integrative method for accurate comparative genome mapping. submitted (2005)
2. Qian, W., Jia, Y., Ren, S.X., He, Y.Q., Feng, J.X., Lu, L.F., Sun, Q., Ying, G., et al.: Comparative and functional genomic analyses of the pathogenicity of phytopathogen *Xanthomonas campestris* pv. *campestris*. Genome Res. **15**(6) (2005) 757–767
3. Achaz, G., Boyer, F., Rocha, E.P.C., Viari, A., Coissac, E.: Extracting approximate repeats from large DNA sequences. (2004)
4. Kowalczykowski, S.C., Dixon, D.A., Eggleston, A.K., Lauder, S.D., Rehrauer, W.M.: Biochemistry of homologous recombination in *Escherichia coli*. Microbiol. Rev. **58** (1994) 401–65
5. Kececioglu, J., Sankoff, D.: Exact and approximation algorithms for the inversion distance between two permutations. In: Proc. of 4th Ann. Symp. on Combinatorial Pattern Matching. (1993) 87–105
6. Kaplan, H., Shamir, R., Tarjan, R.E.: Faster and simpler algorithm for sorting signed permutations by reversals. In: Proc. 8th Ann. Symp. on Discrete Algorithms. (1997) 344–351
7. Hannenhalli, S., Pevzner, P.A.: Transforming cabbage into turnip: Polynomial algorithm for sorting signed permutations by reversals. J. ACM **46** (1999) 1–27
8. Bergeron, A., Mixtacki, J., Stoye, J.: Reversal distance without hurdles and fortresses. In: 15th Ann. Symp. on Combinatorial Pattern Matching. (2004) 388–399
9. Tannier, E., Sagot, M.F.: Sorting by reversals in subquadratic time. In: Proc. of the 15th Ann. Sym. on Combinatorial Pattern Matching. (2004) 1–13
10. Bender, M., Ge, D., He, S., Hu, H., Pinter, R., Skiena, S., Swidan, F.: Improved bounds on sorting with length-weighted reversals. In: Proc. 15th ACM-SIAM Symposium on Discrete Algorithms. (2004) 912–921
11. Swidan, F., Bender, M.A., Ge, D., He, S., Hu, H., Pinter, R.Y.: Sorting by length-weighted reversals: Dealing with signs and circularity. In: Proc. 15th Annual Combinatorial Pattern Matching Symposium. (2004) 32–46
12. Bergeron, A.: A very elementary presentation of the hannenhalli-pevzner theory. Discrete Applied Mathematics **146**(2) (2005) 134–145
13. Caprara, A.: Formulations and hardness of multiple sorting by reversals. In: Proc 3th Ann. Int. Conf. on Computational Molecular Biology, New York, NY, USA, ACM Press (1999) 84–93
14. Moret, B., Wang, L., Warnow, T., Wyman, S.: New approaches for reconstructing phylogenies from gene order data. In: Proc. 9th Int. Conf. Intell. Syst. Mol. Biol. (2001) 165–173
15. Bourque, G., Pevzner, P.A.: Genome-scale evolution: Reconstructing gene orders in the ancestral species. Genome Res. **12**(1) (2002) 26–36
16. Gonnet, G.: Handbook of Algorithms and Data Structures. International Computer Science Services (1983)

Subsequence Combinatorics and Applications to Microarray Production, DNA Sequencing and Chaining Algorithms

Sven Rahmann

Algorithms and Statistics for Systems Biology Group,
Genome Informatics, Faculty of Technology, Bielefeld University,
D-33594 Bielefeld, Germany
Sven.Rahmann@cebitec.uni-bielefeld.de

Abstract. We investigate combinatorial enumeration problems related to subsequences of strings; in contrast to substrings, subsequences need not be contiguous. For a finite alphabet Σ, the following three problems are solved. **(1) Number of distinct subsequences**: Given a sequence $s \in \Sigma^n$ and a nonnegative integer $k \leq n$, how many distinct subsequences of length k does s contain? A previous result by Chase states that this number is maximized by choosing s as a repeated permutation of the alphabet. This has applications in DNA microarray production. **(2) Number of ρ-restricted ρ-generated sequences**: Given $s \in \Sigma^n$ and integers $k \geq 1$ and $\rho \geq 1$, how many distinct sequences in Σ^k contain no single nucleotide repeat longer than ρ and can be written as $s_1^{r_1} \dots s_n^{r_n}$ with $0 \leq r_i \leq \rho$ for all i? For $\rho = \infty$, the question becomes how many length-k sequences match the regular expression $s_1{*}s_2{*}\dots s_n{*}$. These considerations allow a detailed analysis of a new DNA sequencing technology ("454 sequencing"). **(3) Exact length distribution of the longest increasing subsequence**: Given $\Sigma = \{1, \dots, K\}$ and an integer $n \geq 1$, determine the number of sequences in Σ^n whose longest strictly increasing subsequence has length k, where $0 \leq k \leq K$. This has applications to significance computations for chaining algorithms.

1 Introduction

In contrast to substrings, subsequences have received less attention as objects in pattern matching; yet certain aspects of recent technologies emerging in the life sciences, such as short oligonucleotide microarrays or massive short range DNA sequencing by the so-called 454 approach, directly lead to subsequence enumeration problems. The present paper studies a selection of them and presents applications in molecular biology.

A string of length n over a finite alphabet Σ contains $(n+1)n/2 = \Theta(n^2)$ (nonempty) substrings, but 2^n subsequences (including the empty string), making enumerative combinatorics on subsequences potentially more difficult. For a fixed length $1 \leq k \leq n$, there are $n-k+1$ substrings and $\binom{n}{k}$ subsequences of length k. Note that not all of these need to be different.

M. Lewenstein and G. Valiente (Eds.): CPM 2006, LNCS 4009, pp. 153–164, 2006.

In Section 2 we present an algorithm that needs $O(k(n+|\Sigma|))$ arithmetic operations to count the number $C_k(s)$ of distinct length-k subsequences in s. When we compute $C_k(s)$ exactly, the size of these numbers is $O(k \log |\Sigma|)$ bits, so arithmetic operations cannot be assumed to take constant time in the RAM model of computation; however, if we are satisfied with computing them with constant precision, we can make this assumption. Therefore we specify running times in numbers of arithmetic operations. The ability to count the number of distinct subsequences contained in a sequence has applications to DNA microarray production, which we outline also in Section 2.

For the next problem, we generalize the notion of subsequence and say that t is *generated* by s if it consists of a concatenation of runs (repetitions) of selected characters from s. This allows, for example, to determine the number of length-k sequences that match the regular expression $s_1{*}s_2{*}\dots s_n{*}$, where $a{*}$ matches an arbitrary number of (including zero) occurrences of $a \in \Sigma$. If we additionally restrict the run lengths to be bounded by a constant $\rho \geq 1$, the question of determining how many length-k strings are generated by a given string s is of interest for evaluating a new massively parallel DNA sequencing technology ("454 sequencing", [1]). Section 3 presents an efficient counting algorithm and computational results. The results of Sections 2 and 3 can be summarized as

Theorem 1. *The number of distinct subsequences and the number of ρ-restricted ρ-generated length-k sequences from a sequence of length n over an alphabet Σ can be computed with $O(k(n+|\Sigma|))$ arithmetic operations.*

Finally, we are interested in the distribution of the longest (strictly) increasing subsequence of $s \in \Sigma^n$ over an ordered alphabet $\Sigma := \{1, \dots, K\}$. We say that t is an increasing subsequence of s of there exists an integer $1 \leq k \leq n$ and indices $1 \leq j_1 < j_2 < \cdots < j_k \leq n$ such that $t = s_{j_1} s_{j_2} \dots s_{j_k}$ and $s_{j_1} < s_{j_2} < \cdots < s_{j_k}$. Let $I(s)$ be the set of all increasing subsequences of s, and let $LIS(s) := \max_{t \in I(s)} |t|$ be the length of the longest increasing subsequence.

Our goal is to determine the distribution of $L_n := LIS(S)$, where S is a random length-n sequence. Recently, the analogous problem has been completely solved on uniform random permutations; there are exact results for finite n and asymptotic results for $n \to \infty$ provided by the Baik-Deift-Johansson Theorem, e.g., the expected length is $2\sqrt{n} + \Theta(n^{1/6})$, the standard deviation is $\Theta(n^{1/6})$, and the limiting distribution of $(L_n - 2\sqrt{n})/n^{1/6}$ is completely known. A review of these results on permutations and additional results on weakly increasing subsequences on words appears in [2]. So far there seem to exist no exact nor asymptotic results on strictly increasing subsequences in words. Our contribution is a method that needs $O(nK2^K)$ arithmetic operations on $O(n \log K)$-bit numbers to compute the exact distribution (in terms of absolute numbers). We thus have the following fixed-parameter tractability (FPT) result (see [3] for an introduction to the terminology).

Theorem 2. *For given string length n and parameter alphabet size K, the decision problem whether there are at least $T \geq 0$ sequences in $s \in \{1, \dots, K\}^n$ with $L(s) = k$ for any $1 \leq k \leq K$ is FPT.*

2 The Number of Distinct Subsequences

Let Σ be a finite alphabet of size σ; w.l.o.g. we assume $\Sigma = \{1, \ldots, \sigma\}$. Further, let $s \in \Sigma^n$ and an integer $1 \leq k \leq n$ be given. We write $t \lhd s$ to indicate that t is a subsequence of s, i.e., there exist indices $1 \leq i_1 < i_2 < i_{|t|} \leq n$ such that $s_{i_1} s_{i_2} \ldots s_{i_{|t|}} = t$. Our goal is to determine the cardinality $C_k(s)$ of $S_k(s) := \{t \in \Sigma^k : t \lhd s \}$, i.e., the number of distinct length-k subsequences in s. To compute $C_k(s)$ efficiently, we derive a recurrence on the number of distinct subsequences of given length in a given prefix of s that end with a specified character. We drop the dependence on Σ and s in the notation and define

$$S_{m,j} := \{t \in \Sigma^m : t \lhd s_1 \ldots s_j\}, \qquad C_{m,j} := |S_{m,j}|.$$

We refine this definition by conditioning on the last character $a \in \Sigma$:

$$S_{m,j}[a] := \{t \in \Sigma^m : t \lhd s_1 \ldots s_j \text{ and } t_m = a\}, \qquad C_{m,j}[a] := |S_{m,j}[a]|.$$

Note that $S_{0,j} = \{\epsilon\}$ (the set consisting of the empty string) for all j, but $S_{0,j}[a] = \{\}$ for all $a \in \Sigma$, so $1 = C_{0,j} \neq \sum_{a \in \Sigma} C_{0,j}[a] = 0$. However, for $m > 0$, we do have $C_{m,j} = \sum_{a \in \Sigma} C_{m,j}[a]$ for all j.

For two sets S and T of strings over Σ, let $S \circ T := \{st : s \in S,\, t \in T\}$.

The goal is thus to compute $C_k(s) = C_{k,n} = \sum_{a \in \Sigma} C_{k,n}[a]$. The following lemma presents a structural equation for $S_{m,j}[a]$, which leads to a recurrence on $C_{m,j}[a]$ in Lemma 2.

Lemma 1. *Let $1 \leq m \leq j$. Then*

$$S_{m,j}[a] = \begin{cases} S_{m,j-1}[a] & \textit{if } s_j \neq a, \\ S_{m-1,j-1} \circ \{a\} & \textit{if } s_j = a. \end{cases}$$

Proof. Assume first that $s_j \neq a$. The inclusion $S_{m,j-1}[a] \subset S_{m,j}[a]$ is trivial. We prove that $S_{m,j}[a] \subset S_{m,j-1}[a]$: Take $t \in S_{m,j}[a]$. Since $s_j \neq a$, it follows that t is already a subsequence of a shorter prefix of s, i.e., $t \in S_{m,j-1}[a]$.

Now assume that $s_j = a$. By appending an a to each $t \in S_{m-1,j-1}$ (regardless of its last character), we obtain a distinct string $ta \in S_{m,j}[a]$, thus $S_{m-1,j-1} \circ \{a\} \subset S_{m,j}[a]$. Conversely, every string in $S_{m,j}[a]$ can be written as ta with some $t \in S_{m-1,j-1}$. □

Lemma 2. *We have $C_{0,0} = 1$ and $C_{0,0}[a] = 0$ for all $a \in \Sigma$. Further, $C_{m,j} = 0$ if $m > j$. For $1 \leq m \leq j$, we have*

$$C_{m,j}[a] = \begin{cases} C_{m,j-1}[a] & \textit{if } s_j \neq a, \\ C_{m-1,j-1} & \textit{if } s_j = a. \end{cases}$$

Proof. Immediate by taking cardinalities in Lemma 1 and noting that concatenation translates to multiplication of set cardinalities. □

Bernoulli String Model. The fraction of length-k sequences contained in s (or *covered by* s) is thus $C_{k,n}/\sigma^k$. We can generalize Lemma 2 to a Bernoulli or i.i.d. random string model, where the probability or weight of each length-k string is equal to the product of its (possibly unequal) character frequencies. Hence, let $\pi := (\pi_a)_{a\in\Sigma}$ be a non-degenerate probability distribution on Σ, i.e., $\pi_a > 0$ for all $a \in \Sigma$ and $\sum_{a\in\Sigma} \pi_a = 1$. Let $\mathbb{P}_k(t_1 \dots t_k) := \prod_{j=1}^{k} \pi_{t_j}$ be the probability of generating $t_1 \dots t_k$ in k steps. It follows that $\sum_{t\in\Sigma^k} \mathbb{P}_k(t) = 1$ for all $k \geq 1$ and $\mathbb{P}_{k+1}(ta) = \mathbb{P}_k(t) \cdot \pi_a$ for $t \in \Sigma^k$ and $a \in \Sigma$.

Let us define $W_k(s) := \mathbb{P}_k(S_k(s))$ as the weighted fraction of length-k sequence space covered by s. For $m \leq k$ and $j \leq |s| = n$, define

$$W_{m,j} := \mathbb{P}_m(S_{m,j}) = \sum_{t\in S_{m,j}[a]} \mathbb{P}_m(t), \qquad W_{m,j}[a] := \mathbb{P}_m(S_{m,j}[a]).$$

Lemma 3. *$W_{0,0} = 1$ and $W_{0,0}[a] = 0$ for all $a \in \Sigma$. Further, $W_{m,j} = 0$ if $m > j$. For $1 \leq m \leq j$, we have*

$$W_{m,j}[a] = \begin{cases} W_{m,j-1}[a] & \text{if } s_j \neq a, \\ W_{m-1,j-1} \cdot \pi_a & \text{if } s_j = a. \end{cases}$$

Proof. Immediate by applying $\mathbb{P}_m(\cdot)$ resp. $\mathbb{P}_{m-1}(\cdot)$ to Lemma 1. □

A straightforward implementation of the recurrence would need $O(nk|\Sigma|)$ arithmetic operations. It is possible to remove the factor $|\Sigma|$ in a careful implementation: Figure 1 presents an algorithm to compute $W_k(s)$ in $O(k(n + |\Sigma|))$ operations. The memory requirements are $O(k|\Sigma|)$ if only $W_k(s)$ is desired or $O(k(n+|\Sigma|))$ if the whole array $W_m(s_1 \dots s_j)$, $1 \leq m \leq k$, $1 \leq j \leq n$, is desired.

Application to DNA microarray production. DNA oligonucleotide microarrays ("DNA chips") are a tool to monitor the activity level of many genes in cells of living organisms. A DNA chip is a plastic or glass slide containing many *spots*, each consisting of many copies of a known oligomer (a 25-mer for Affymetrix GeneChips®, which we consider here), also called *probe*, attached to the chip. During production, the probes are synthesized on the chip in parallel on a nucleotide-by-nucleotide-basis. In each synthesis step, the same nucleotide is appended to all probes that have been selectively activated to receive it. Activation occurs by exposure to light, enabling the chemical synthesis reaction. Thus each synthesis step is specified by (1) a nucleotide (a character from the DNA alphabet {A,C,G,T}) and (2) a mask, i.e., an index set of the probes to which the nucleotide is appended. The sequence of nucleotides used in the synthesis process is called the *deposition sequence.* Each probe is a subsequence of the deposition sequence, so the deposition sequence is a common supersequence of all probes.

Given a set of probe sequences (in practice up to 10^6 probes can fit on a single chip), one can try to find the shortest deposition sequence, i.e., the shortest common supersequence of all probes (see [4] for bounds on its length and heuristic algorithms). In practice, good deposition sequences can be found but

```
Input: Alphabet $\Sigma$ with probability distribution $\pi$, string $s \in \Sigma^n$, integer $1 \le k \le n$
Output: $W_k(s)$ or the whole array $W_m(s_1 \dots s_j)$ for $m = 1, \dots, k$, $j = 1, \dots, n$
// Initialize arrays W, V and Vsum
W$[m, j] \leftarrow 0$ for $m \leftarrow 1, \dots, k$, $j \leftarrow 1, \dots, n$ // optional: stores $W_{m,j}$
V$[m, a] \leftarrow 0$ for $m \leftarrow 1, \dots, k$, $a \in \Sigma$ // stores $W_{m,j}[a]$ for current value of $j$
Vsum$[m] \leftarrow 0$ for $m \leftarrow 1, \dots, k$ // stores $W_{m,j}$ for current value of $j$

for $j \leftarrow 1, \dots, n$
    $c \leftarrow s_j$ // the current character
    for $m \leftarrow \min\{j, k\}, \dots, 3, 2$
        // Update V and Vsum s.th. V$[m, a] = W_{m,j}[a]$ $(a \in \Sigma)$; Vsum$[m] = W_{m,j}$:
        // (only the $c$-entry needs to be updated, saving a factor of $|\Sigma|$)
        Vsum$[m] \leftarrow$ Vsum$[m] -$ V$[m, c]$
        V$[m, c] \leftarrow$ Vsum$[m - 1] \cdot \pi_c$
        Vsum$[m] \leftarrow$ Vsum$[m] +$ V$[m, c]$
    end for $m$
    // Finally, treat the case $m = 1$ specially:
    if V$[1, c] = 0$ then V$[1, c] \leftarrow \pi_c$; Vsum$[1] \leftarrow$ Vsum$[1] + \pi_c$; end if
    // Invariant: Here Vsum$[m] = W_m(s_1 \dots s_j) = W_{m,j}$ for $m = 1, \dots, k$
    W$[m, j] \leftarrow$ Vsum$[m]$ for $m \leftarrow 1, \dots, k$ // optional: set $j$-th column of W:
end for $j$
return Vsum$[k]$ // optional: return array W
```

Fig. 1. An algorithm with $O(k(n+|\Sigma|))$ operations to compute the π-weighted fraction $W_k(s)$ of length-k strings that are subsequences of s. The array W is not needed when only $W_k(s)$ is required: after step j, the j-th column of W is equal to Vsum.

not proved optimal in a reasonable amount of time. Therefore one can approach the question differently and ask for a deposition sequence that is as "universal" as possible, i.e., that contains the largest number of distinct subsequences. We thus ask for

$$C_k^*(n, |\Sigma|) = \max_{s \in \Sigma^n} C_k(s) \quad \text{and} \quad Best_k^*(n, |\Sigma|) = \{s \in \Sigma^n : C_k(s) = C_k^*(n, \Sigma)\}.$$

A result due to P.J. Chase [5] from 1976 (long before the invention of microarrays) states that precisely the *repeated permutations* of the alphabet form the set $Best_k^*(n, |\Sigma|)$ with the consequence that this set does not depend on k.

Definition 1. *For a finite alphabet Σ of size σ, a string s of length n is called a* repeated permutation *of Σ if there exists a permutation $\pi = \pi_1 \dots \pi_\sigma$ of the characters in Σ such that $s = \pi^c \pi_1 \dots \pi_m$, where the number of full* cycles *is $c := \lfloor n/\sigma \rfloor$ and the number of remaining characters $m := n \bmod \sigma$ satisfies $0 \le m < \sigma$.*

In fact, any sequence that is not a repeated permutation contains strictly fewer subsequences of (some) smaller length. Even though this result appears intuitive, it is nontrivial to prove and apparently does not follow directly from the recurrence in Lemma 2; Chase used induction on the longest sequence prefix that is a repeated permutation to prove optimality.

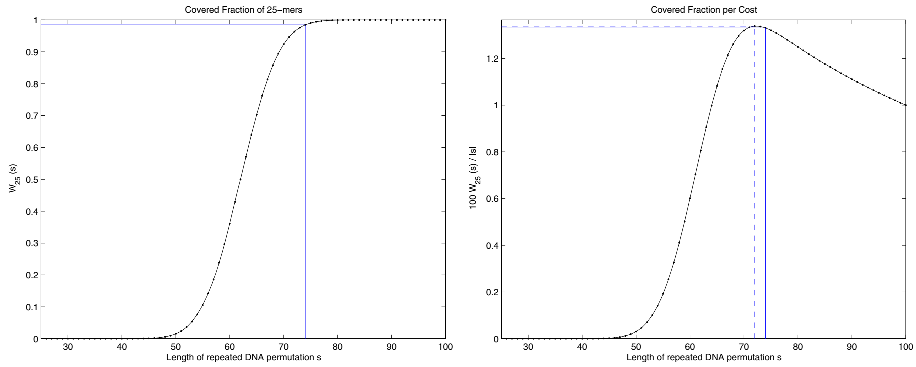


Fig. 2. Left: Fraction of 25-mers covered by a repeated permutation of varying length from 25 to 100: From the deposition sequence s of length 74 used for GeneChip$^{\mathrm{R}}$ production, 98.45% of all 25-mers can be synthesized. Right: Assuming that each synthesis step costs 1/100 (such that using 100 steps implies a cost of 1), the graph shows the covered fraction per cost. The "best value" is obtained for a repeated permutation $s^{\$}$ with 72 steps or 18 full cycles ($W_{25}(s^{\$}) = 96.34\%$, $100\,W_{25}(s^{\$})/72 = 1.338$), but s is almost as cost-effective ($100\,W_{25}(s\ )/74 = 1.3304$) and has higher coverage 98.45%.

The Affymetrix GeneChip® technology uses a repeated permutation of length 74, such as $s^* := (\mathtt{ACGT})^{18}\mathtt{AC}$, to synthesize 25-mers. Figure 2 (left) shows the fraction of 25-mers contained in repeated permutations of increasing length: s^* covers a fraction of 98.45% of all 25-mers. Elongating s^* further quickly results in diminishing returns; for example, adding one additional nucleotide would result in 99.04% of the 25-mers being covered. It is unknown to the author why the length of 74 was chosen, but we offer the following hypotheses: The sequences not covered by s^* have somewhat extremal properties. For example, many of them contain runs of a repeated nucleotide. We may assume that such oligos are rarely used on microarrays because of undesirable thermodynamic properties, so s^* may cover in fact all oligos that are ever chosen to be placed on a chip. For another argument consider Figure 2 (right): In practice, each synthesis step has a certain cost (mask production, chemicals, time, etc.). Assuming that the production cost of a chip is proportional to the number of synthesis steps, we see that using a deposition sequence of length 74 offers both high coverage in absolute terms and close to optimal coverage per money.

3 The Number of ρ-Restricted ρ-Generated Sequences

We consider a variation of the previous problem, where we modify the notion of subsequence: We allow that each character from s, which we call the *generating sequence*, may produce a whole run (up to a specified length ρ) of this character. Thus we write $t \lhd_\rho s$ if there exist n numbers $0 \le r_i \le \rho$ for $i = 1, \ldots, n$, with $|t| = \sum_i r_i$, such that $t = s_1^{r_1} s_2^{r_2} \ldots s_n^{r_n}$. We say that t is *ρ-generated* by s. For $\rho = 1$, we get the usual notion of subsequence. Note that $t \lhd_\rho s$ implies $|t| \le \rho|s|$.

Motivated by the 454 DNA sequencing technology (see below), we are only interested in counting sequences that do not contain a single character run longer than ρ; so we define Σ_ρ^k as the set of all length-k strings over Σ that do not contain $a^{\rho+1}$ as a substring for any $a \in \Sigma$ and call them the *ρ-restricted* strings.

The set of ρ-restricted length-k strings ρ-generated by s is denoted by

$$S_k(s;\rho) := \{t \in \Sigma_\rho^k : t \lhd_\rho s\}.$$

It is important to note that $a^{2\rho} \lhd_\rho aba$, but $a^{2\rho} \notin \Sigma_\rho^{2\rho}$, so $a^{2\rho} \notin S_{2\rho}(aba;\rho)$. Therefore $S_k(s;1)$ is different from $S_k(s)$ as defined in the previous section.

For the generating sequence $s = (s_1, \dots, s_n)$, we may assume that $s_i \neq s_{i+1}$ for all $i = 1, \dots, n-1$, i.e., $s \in \Sigma_1^n$, since repetitions in the generating sequence do not allow to generate additional ρ-restricted sequences.

We set $C_k(s;\rho) := |S_k(s;\rho)|$ and $W_k(s;\rho) := \mathbb{P}_k(S_k(s;\rho))$. Assuming s and ρ fixed, we define for $1 \leq m \leq k$, $0 \leq j \leq n$ and $a \in \Sigma$ the auxiliary quantities

$$\begin{aligned} S_{m,j}[a] &:= \{t \in \Sigma_\rho^m : t \lhd_\rho s_1 \dots s_j \text{ and } t_m = a\}, & S_{m,j}[\overline{a}] &:= \bigcup\nolimits_{b \neq a} S_{m,j}[b], \\ C_{m,j}[a] &:= |S_{m,j}[a]|, & C_{m,j}[\overline{a}] &:= |S_{m,j}[\overline{a}]|, \\ W_{m,j}[a] &:= \mathbb{P}_m(S_{m,j}[a]), & W_{m,j}[\overline{a}] &:= \mathbb{P}_m(S_{m,j}[\overline{a}]), \end{aligned}$$

with the boundary cases $S_{0,j}[a] = \{\}$ and $S_{0,j}[\overline{a}] = \{\epsilon\}$. The structural recurrence for $S_{m,j}[a]$ is slightly more complicated than in the previous section, since we need to express $S_{m,j}[a]$ as a *disjoint* union to determine its cardinality.

Lemma 4. *Let $1 \leq m \leq j$. Then*

$$S_{m,j}[a] = \begin{cases} S_{m,j-1}[a] & \text{if } s_j \neq a, \\ \bigcup_{r=1}^{\min\{\rho,m\}} (S_{m-r,j-1}[\overline{a}] \circ \{a^r\}) & \text{if } s_j = a, \end{cases}$$

where the union is disjoint.

Proof. The case $s_j \neq a$ is proved as in Lemma 1.

For $s_j = a$, appending a^r to any $t \in S_{m-r,j-1}[\overline{a}]$ for any "run length" $1 \leq r \leq \min\{\rho, m\}$ clearly results in a distinct string in $S_{m,j}[a]$. Note that any run length in t is bounded by r by assumption, and in ta^r by construction since t does not end with a. This shows $\cup_{r=1}^{\min\{\rho,m\}} S_{m-r,j-1}[\overline{a}] \circ \{a^r\} \subset S_{m,j}[a]$. Conversely, every string in $S_{m,j}[a]$ can be written uniquely as ta^r, where $r \leq \rho$ and $r \leq m$ and $t \in S_{m-r,j-1}[\overline{a}]$ (possibly the empty string). Because of the uniqueness of the above decomposition, the union is disjoint. □

Lemma 4 immediately allows us to count $C_k(s;\rho)$ and to determine $W_k(s;\rho)$. We only give the Bernoulli string model version for $W_k(s;\rho)$ here.

Lemma 5. *We have $W_{0,j}[\overline{a}] = 1$ and $W_{0,j}[a] = 0$ for all $a \in \Sigma$, $j \geq 0$. For $m \geq 1$ and $j \geq 1$, we have*

$$W_{m,j}[a] = \begin{cases} W_{m,j-1}[a] & \text{if } s_j \neq a, \\ \sum_{r=1}^{\min\{m,\rho\}} W_{m-r,j-1}[\overline{a}] \cdot \pi_a^r & \text{if } s_j = a. \end{cases}$$

The desired result is $W_k(s) = W_{k,n}[\overline{a}] + W_{k,n}[a]$ for any $a \in \Sigma$.

Remarks:

1. The recurrence in Lemma 5 can be implemented to run in $O(k(n+|\Sigma|))$ arithmetic operations by remembering appropriate partial sums.
2. Using $\rho=\infty$ answers the question how many strings of length k match the regular expression $s_1\texttt{*}s_2\texttt{*}\dots s_n\texttt{*}$, where $a\texttt{*}$ matches zero or an arbitrary number of occurrences of $a\in\Sigma$. In Section 2, we effectively determined how many strings of length k match the regular expression $s_1\texttt{?}s_2\texttt{?}\dots s_n\texttt{?}$, where $a\texttt{?}$ matches zero or one occurrence(s) of $a\in\Sigma$.
3. It is reasonable to conjecture that again a repeated permutation s^* maximizes $C_k(s;\rho)$ over all $s\in\Sigma^n$, but this is so far not rigorously proved.
4. Even for arbitrarily large n and optimal $s^*\in\Sigma^n$, we have $C_k(s;\rho)/|\Sigma^k|\le|\Sigma^k_\rho|/|\Sigma^k|\to 0$ as $k\to\infty$, because the probability that a length-k sequence contains a run longer than ρ approaches 1 as $k\to\infty$.

Analysis of 454 Sequencing. Recently, the company “454 Life Sciences” has developed a massively parallel DNA sequencing technology (simply called “454 sequencing”). We refer the reader to [1] and `www.454.com` for more detailed information. Several copies of an organism’s genome are randomly cut into DNA fragments; a part of the sequence of each fragment is determined in parallel, and finally the fragment sequences can assembled to retrieve the whole genomic sequence if each position of the genome is covered by enough fragments. Many copies of one single fragment type are attached to a microscopic bead; each bead is held in place in a different well of the reaction carrier (70 mm × 75 mm). A typical reaction carrier has 1.6 million wells, from which typically 200,000 different high-quality fragment reads can be obtained.

The fragments are sequenced by synthesizing the complementary (`A` ↔ `T`, `C` ↔ `G`) DNA strand to each fragment in several steps. Initially, the complementary strand of each fragment is empty but ready for extension at its starting point. Then, e.g., in an `A`-step, `T`-nucleotides are flooded over the reaction carrier, and `T`s are incorporated into complementary strands in those wells where the next character in the fragment sequence is `A`. Successful elongation of the complementary strand results in a flash of light from the corresponding wells. The light emission pattern is detected with a CCD camera for all wells in parallel. If a fragment contains a consecutive run (homopolymer) of `A`s, all of their counterpart `T`s are incorporated in a single step and the light intensity is proportional to the run length. This works reliably only up to a certain length $\rho=8$, which was the reason for introducing ρ-restricted strings above. Sequences that contain longer homopolymers cannot be reliably sequenced.

Sequencing steps for different nucleotides are repeated in a cyclic pattern for c cycles, e.g., $(\texttt{ACGT})^c$. This process cannot go on forever because the signal/noise ratio deteriorated over time. Public information (as of February 2005) at `www.454.com` states that high-quality sequencing of on average 100-base reads is achieved in 42 cycles of `TACG`. It has also been attempted to use 84 and 168 cycles for high-quality reads of 200 and 400 bases, respectively.

The key issue is that the fraction of length-k DNA sequences that can be reliably sequenced by this technology in n steps is precisely given by $W_k(s;\rho)$,

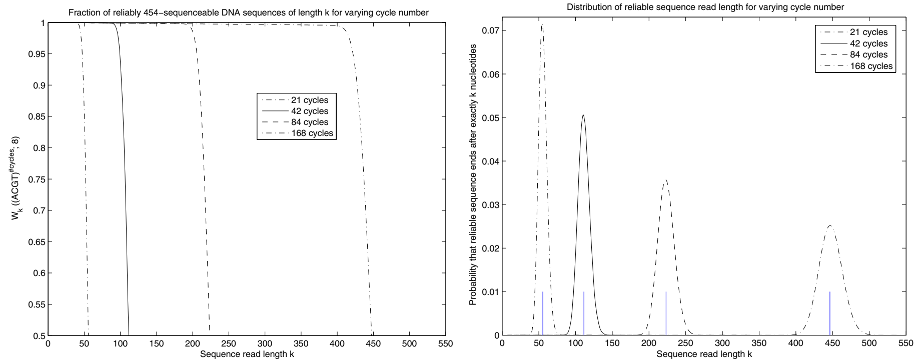


Fig. 3. Left: Fraction W_k of 454-sequenceable length-k DNA sequences by using a repeated permutation of the alphabet for $c \in \{21, 42, 84, 168\}$ cycles, $\rho = 8$. Right: Length distribution of the reliably sequenceable initial fragment of a random DNA sequence, for c and ρ as before. Vertical lines mark the expected lengths.

where $\rho = 8$ and s is a repeated permutation of the DNA alphabet. Assuming a uniform distribution $\pi_a = 1/4$ for each $a \in \Sigma$, we thus determine which fraction $W_k((\texttt{ACGT})^c; 8)$ of length-k DNA sequences for $1 \leq k \leq 550$ can be reliably sequenced in $c \in \{21, 42, 84, 168\}$ full cycles.

The results are visualized in Figure 3 (left). The longest sequence lengths for which the sequenceable fraction exceeds 99% are $k_{\max} = 48$, 101, 209, and 427 for $c = 21$, 42, 84, and 168 cycles. 85.8% of length-50 sequences are sequenceable in 21 cycles, 94.0% of length-100 sequences in 42 cycles, 98.55% of length-200 sequences in 84 cycles, and 99.48% of length-400 sequences in 168 cycles.

A different perspective is shown in Figure 3 (right): If T is any (potentially infinite) random sequence according to the uniform distribution, a certain finite prefix will be reliably sequenced by the generating sequence $s = (\texttt{ACGT})^c$. Let $L_c(T)$ denote the length of this prefix for c cycles. The figure shows the distribution of L_c for $c \in \{21, 42, 84, 168\}$ cycles, which is obtained as follows. The probability that sequencing ends after k steps or later is $W_k \equiv W_k(s; \rho)$. Therefore, the probability that the read ends *exactly* after k steps is $\mathbb{P}(L_c = k) = W_k - W_{k+1}$. The figure also shows that the expected sequence read length $\mathbb{E}[L_c]$ for 21 (42, 84, 168) cycles is 55.4 (111.4, 223.1, 446.3), which exceeds the company-guaranteed values of 50 (100, 200, 400) by more than 10%. To guarantee these expected read lengths, only 19 (37.75, 75.5, 150.75) cycles, i.e., 76 (151, 302, 603) steps would in fact be necessary on random sequences.

4 Longest Increasing Subsequence Length Distribution

We consider an ordered alphabet $\Sigma := \{1, \ldots, K\}$ and a string $s \in \Sigma^n$, and equip Σ^n with a Bernoulli probability measure $\mathbb{P}_n$ given by a probability vector $\pi = (\pi_1, \ldots, \pi_K)$, such that $\mathbb{P}_n(s) = \prod_{j=1}^n \pi_{s_j}$. Several algorithms (e.g., [6, 2]) compute the length $LIS(s)$ of the longest increasing subsequence in s.

Our counting method is based on the *patience sorting* algorithm, which scans s from left to right and keeps track of a subset $\kappa \subset [K] := \{1, \dots, K\}$ whose cardinality after j steps is equal to $LIS(s_1 \dots s_j)$. We write $2^{[K]}$ for the power set of $\{1, \dots, K\}$. Initially, we set $\kappa_0 = \{\}$ and in step $j = 1, \dots, n$, κ_j is computed in $O(\log K)$ operations as $\kappa_j := u(\kappa_{j-1}, s_j)$ from the *update function* $u : 2^{[K]} \times [K] \to 2^{[K]}$; $(\kappa, c) \mapsto \kappa^+$, defined as follows:

- If $c \in \kappa$, do nothing, i.e., set $\kappa^+ := \kappa$.
- If $c \notin \kappa$ and κ contains no element $> c$, add c, i.e., set $\kappa^+ := \kappa \cup \{c\}$.
- If $c \notin \kappa$ and there exists $k \in \kappa$ with $k > c$, find the smallest such k and decrease it to c, i.e., set $\kappa^+ := \kappa \setminus \{k\} \cup \{c\}$.

A proof that $|\kappa_j| = LIS(s_1, \dots, s_j)$ and an explanation in terms of stacks of cards is found in [2]. The running time is seen to be $O(n \log K)$. To avoid running patience sorting for all K^n sequences separately, we condition on κ: Let $\kappa_j(t)$ be the final set κ_j in patience sorting when it is applied to $t \in \Sigma^j$. We set

$$S_j(\kappa) := \{t \in \Sigma^j : \kappa_j(t) = \kappa\}, \qquad C_j(\kappa) := |S_j(\kappa)|, \qquad W_j(\kappa) := \mathbb{P}_j(S_j(\kappa)).$$

It follows that for $0 \le k \le K$,

$$S_n(k) := \bigcup_{\substack{\kappa \subset [K],\\ |\kappa| = k}} S_j(\kappa), \qquad C_n(k) := \sum_{\substack{\kappa \subset [K],\\ |\kappa| = k}} C_j(\kappa), \qquad W_j(k) := \sum_{\substack{\kappa \subset [K],\\ |\kappa| = k}} W_j(\kappa)$$

are the set, number, and weighted fraction of length-n sequences with $LIS = k$, respectively. The following lemma presents a structural equation between $S_j(\kappa)$ and $S_{j-1}(\kappa')$, where κ' is an update-preimage under u.

Lemma 6. *For $j = 0$, we have $S_0(\{\}) = \{\epsilon\}$, $C_0(\{\}) = 1$, $W_0(\{\}) = 1$, and for $\kappa \in 2^{[K]}$, $\kappa \neq \{\}$, we have $S_0(\kappa) = \{\}$, $C_0(\kappa) = 0$, $W_0(\kappa) = 0$. For $1 \le j \le n$ and $\kappa \in 2^{[K]}$,*

$$S_j(\kappa) = \bigcup_{(\kappa', c) \in u^{-1}(\kappa)} S_{j-1}(\kappa') \circ \{c\},$$

$$C_j(\kappa) = \textstyle\sum_{(\kappa', c) \in u^{-1}(\kappa)} C_{j-1}(\kappa'), \quad \textit{and} \quad W_j(\kappa) = \textstyle\sum_{(\kappa', c) \in u^{-1}(\kappa)} W_{j-1}(\kappa') \cdot \pi_c.$$

Proof. The equations for C_j and W_j follow immediately from the one for S_j (obviously the union is disjoint), which in turn is a trivial consequence of the correctness of the patience sorting algorithm (i.e., of the update function). □

Lemma 6 implies a "pull"-type dynamic programming algorithm for computing $W_n(k)$, which has the disadvantage that the update rules must be read "backwards", i.e., for given κ, we need to determine the pairs (κ', c) with $\kappa = u(\kappa', c)$. It is easier to implement a "push"-type algorithm that pushes the information for all (κ, c) forward to the corresponding $\kappa^+ = u(\kappa, c)$. This is shown in Figure 4.

Application: Significance Computations for Chaining Algorithms. In biological sequence analysis, the following problem arises in several situations (e.g., when attempting to classify proteins or to detect cis-regulatory modules): Certain biological sequences (the *family members*) are characterized by the appearance

```
**Input:** Alphabet size $K$, distribution $\pi = (\pi_1, \ldots, \pi_K)$, sequence length $n$
**Output:** $W_n(k)$ for $0 \le k \le K$ as array w[0..$K$]

W'[{}] $\leftarrow$ 1 and W'[$\kappa$] $\leftarrow$ 0 for $\kappa \in 2^{[K]}$ with $|\kappa| \ge 1$ // Initialize array W'[$\kappa$] to $W_0(\kappa)$
for $j \leftarrow 1, \ldots, n$
    W[$\kappa$] $\leftarrow$ 0 for $\kappa \in 2^{[K]}$    // reset array W to zero
    // Invariant here: W'[$\kappa$] = $W_{j\;1}(\kappa)$ and W[$\kappa$] $\equiv$ 0
    for $\kappa \in 2^{[K]}$; for $c \in \Sigma$
        $\kappa^+ \leftarrow u(\kappa, c)$
        W[$\kappa^+$] $\leftarrow$ W[$\kappa^+$] + W'[$\kappa$] $\cdot \pi_c$
    end for $c$; end for $\kappa$
    W' $\leftarrow$ W    // Invariant: W[$\kappa$] = W'[$\kappa$] = $W_j(\kappa)$
end for $j$
w[$k$] $\leftarrow$ 0 for $k \leftarrow 0, \ldots, K$
w[$|\kappa|$] $\leftarrow$ w[$|\kappa|$] + W[$\kappa$] for all $\kappa \in 2^{[K]}$
return w
```

Fig. 4. Push-type dynamic programming algorithm to compute the length distribution of the longest increasing subsequence for alphabet size K with character distribution $\pi = (\pi_1, \ldots, \pi_K)$ and sequence length n. Subsets κ can be encoded as bit-vectors and represented as integers in the range from 0 to $2^K - 1$.

of sequence motifs (e.g., substrings, regular expressions, or sequence profiles) in a certain order. Let there be K distinct motifs and assume that true family members usually contain all of them in the correct order $1, \ldots, K$. However, in some family members some motifs may not be present or detected. To decide whether a sequence should be classified as a family member, in a first step, all motif occurrences are tabulated. Then the best chain of motifs is found in a second "chaining" step. We assume that the quality of a chain is its length, so we classify a sequence as a family member if the longest increasing sequence of motif indices reaches a threshold t. To find a statistically significant value of t, we determine the frequency p_t of length-t chains in random sequences.

We assume that the motifs are chosen in such a way that each one occurs with low frequency $0 < f_k \ll 1$ in random sequences. If also $f := \sum_{k=1}^{K} f_i \ll 1$, motif occurrences can be treated as a Poisson process along a random sequence of length m: If N is the total number of motif occurrences, then $\mathbb{E}[N] = \lambda := m \cdot f$, and the distribution of N can be well approximated as Poisson(λ) with $\mathbb{P}(N = n) = \exp(-\lambda) \cdot \lambda^n / n!$. Given that a motif occurs at some position, it is motif k with probability $\pi_k := f_k / f$.

It follows that the probability of observing an increasing motif sequence of length k in such a random sequence is given by $W_{\text{Poisson}(\lambda)}(k) := \sum_{n=0}^{\infty} \exp(-\lambda) \cdot \lambda^n / n! \cdot W_n(k)$. The p-value associated to a threshold length t is then $p_t = \sum_{k=t}^{K} W_{\text{Poisson}(\lambda)}(k)$. Now t can be chosen such that p_t is reasonably small.

For example, for $K = 6$ distinct motifs that each appear once in 100 positions on average and sequence length $m = 100$, we have $\lambda = 6$ motif occurrences on average. The Poisson mixture distribution $W_{\text{Poisson}(\lambda)}(k) := \sum_{n=0}^{\infty} \exp(-\lambda) \lambda^n / n! \cdot W_n(k)$ is shown on the left side of Figure 5, the p_t-values on the right side: Thresholds of 5 and 6 imply $p_5 = 0.0165$ and $p_6 = 0.0006$, respectively.

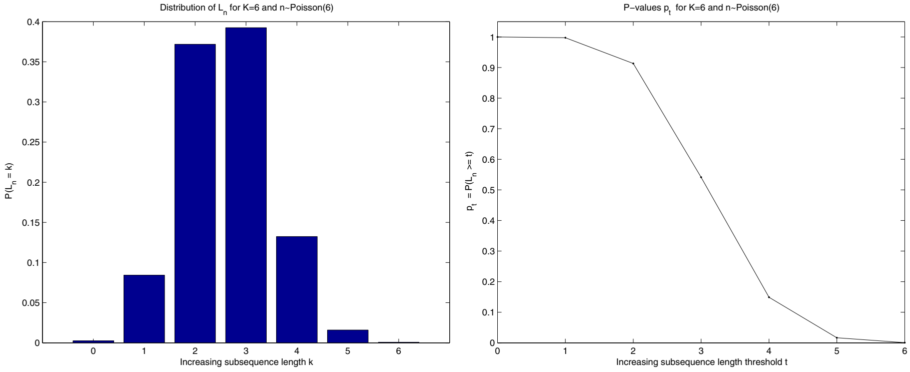


Fig. 5. Left: Length distribution of longest increasing subsequences for alphabet size $K = 6$ and random sequence length $N \sim$ Poisson(6). Right: Associated p-values.

Concluding Remarks. There is considerable literature about subsequence combinatorics (exact and asymptotic counting) on permutations, but there are few results on words, despite the fact that these have interesting practical consequences, as we have shown. Subsequence combinatorics contains a number of interesting problems., e.g., it remains open to prove that indeed the repeated permutations maximize the number of distinct ρ-restricted ρ-generated sequences.

Acknowledgments. I thank Marc Rehmsmeier, Sergio A. de Carvalho Jr., Michael Beckstette, Robert Homann, Jens Stoye, and Dirk Evers for stimulating discussions, and especially Lea Sasaki for her support.

Terms marked ® are registered trademarks of their respective owners. The author is not affiliated with Affymetrix or 454 Life Sciences and has no financial interests competing with this research.

References

1. Margulies, M., et al.: Genome sequencing in microfabricated high-density picolitre reactors. Nature **437**(7057) (2005) 376380 / Corrigendum in Nature **439**(7075) (2006) p.502.
2. Aldous, D., Diaconis, P.: Longest increasing subsequences: From patience sorting to the Baik-Deift-Johansson theorem. Bulletin of the American Mathematical Society **36**(4) (1999) 413432
3. Niedermeier, R.: Invitation to Fixed Parameter Algorithms. Oxford University Press (2006)
4. Rahmann, S.: The shortest common supersequence problem in a microarray production setting. In: Proceedings of the 2nd European Conference in Computational Biology (ECCB 2003). Volume 19 Suppl. 2 of Bioinformatics. (2003) ii156ii161
5. Chase, P.: Subsequence numbers and logarithmic concavity. Discrete Math. **16** (1976) 123140
6. Skiena, S.S.: The Algorithm Design Manual. Springer (1997)

Solving the Maximum Agreement SubTree and the Maximum Compatible Tree Problems on Many Bounded Degree Trees

Sylvain Guillemot and François Nicolas

LIRMM – 161, rue Ada – 34392 Montpellier Cedex 5 – France
{sguillem, nicolas}@lirmm.fr

Abstract. Given a set of leaf-labeled trees with identical leaf sets, the well-known Maximum Agreement SubTree problem (MAST) consists of finding a subtree homeomorphically included in all input trees and with the largest number of leaves. Its variant called Maximum Compatible Tree (MCT) is less stringent, as it allows the input trees to be refined. Both problems are of particular interest in computational biology, where trees encountered have often small degrees.

In this paper, we study the parameterized complexity of MAST and MCT with respect to the maximum degree, denoted D, of the input trees. While MAST is polynomial for bounded D [1, 6, 3], we show that MAST is W[1]-hard with respect to parameter D. Moreover, relying on recent advances in parameterized complexity we obtain a tight lower bound: while MAST can be solved in $O(N^{O(D)})$ time where N denotes the input length, we show that an $O(N^{o(D)})$ bound is not achievable, unless SNP $\subseteq$ SE. We also show that MCT is W[1]-hard with respect to D, and that MCT cannot be solved in $O\big(N^{o(2^{D/2})}\big)$ time, unless SNP $\subseteq$ SE.

1 Introduction

Throughout this paper, $\mathbb{N}$ denotes the set of non-negative integers and, for all $n \in \mathbb{N}$, the set $\{1, 2, \ldots, n\}$ is denoted $[1, n]$.

1.1 Agreement Subtree and Compatible Tree

Trees. All trees considered in this paper are *rooted evolutionary trees*, i.e. trees representing the evolutionary history of a set of species. Such trees are unordered, bijectively leaf-labeled and their internal nodes have at least two children each. Labels are species under study and the branching pattern of the tree describes the way in which speciation events lead from ancestral species to more recent ones.

Leaf labels. For convenience, we will identify the leaves with their labels when the tree is understood. Let T be a (rooted evolutionary) tree. The leaf label set of T is denoted $L(T)$. We say that T is a tree *on* $L(T)$. The *size* of a tree is the cardinality of its leaf set.

M. Lewenstein and G. Valiente (Eds.): CPM 2006, LNCS 4009, pp. 165–176, 2006.

Degree. The *(outer) degree* of a node in T is the number of its children. The *maximum degree* of T, denoted $\Delta(T)$, is the largest degree over all nodes of T.

Parenthetical notation. Parenthetical notation is a convenient way to represent evolutionary trees. Given d non-empty trees T_1, T_2, ..., T_d with pairwise disjoint leaf sets, $\langle T_1, T_2, \ldots, T_d \rangle$ denotes the tree whose root has degree d and admits as child subtrees T_1, T_2, ..., T_d.

Restriction. For each subset $X \subseteq L(T)$, the *(topological) restriction* of T to X is denoted $T|X$. Informally, $T|X$ is the tree on X displaying the branching information of T relevant to X.

Restriction is formally defined by induction as follows. On the one hand, for each leaf tree ℓ, $\ell|\{\ell\} = \ell$ and $\ell|\emptyset$ is the empty tree. On the other hand, a tree T of size at least two can be written as $T = \langle T_1, T_2, \ldots, T_d \rangle$ with $d \geq 2$: if X is a subset of $L(T_i)$ for some $i \in [1, d]$ then $T|X = T_i|X$; otherwise, $T|X$ is the tree on X whose root admits as child subtrees all non empty trees of the form $T_i|(L(T_i) \cap X)$ with $i \in [1, d]$.

MAST and MCT. Let $\mathcal{T}$ be a collection of trees on a common leaf set.

Agreement subtree. An *agreement subtree* of $\mathcal{T}$ is a tree T such that, $\forall T_i \in \mathcal{T}, T = T_i|L(T)$. The Maximum Agreement SubTree problem (MAST) consists of finding an agreement subtree of $\mathcal{T}$ of largest size. In phylogenetics, the maximum size of an agreement subtree of $\mathcal{T}$ is a useful measure of the similarity of the trees in $\mathcal{T}$ [7]. From the point of view of the MAST problem, a node ν of degree d in an input evolutionary tree represents the simultaneous creation of d descendant from the ancestral species represented by ν. As such events are rare if $d > 2$, the trees that people want to calculate maximum agreement subtree for have usually small maximum degrees.

Compatible tree. Let T and T' be two trees on a common leaf set. We say that T *refines* T' if T' can be obtained by collapsing a selection of edges of T. A tree *compatible* with $\mathcal{T}$ is a tree T such that, $\forall T_i \in \mathcal{T}$, T refines $T_i|L(T)$. Obviously, agreement implies compatibility. The converse is usually false for collections including at least a non-binary tree. The Maximum Compatible Tree problem (MCT) consists of finding a tree of largest size compatible with $\mathcal{T}$. The MCT problem is more relevant than the MAST problem when comparing reconstructed evolutionary trees [9, 8]. From the point of view of MCT, a non-binary node is usually interpreted as a lack of decision with respect to the relative grouping of its children rather than as a multi-speciation event. As data sequences are getting longer and phylogenetic methods more accurate, the maximum degree of indecision in reconstructed trees is expected to decrease to a small constant.

Previous Results. MAST is polynomial on two trees (see [12] for the latest algorithm) but becomes NP-hard on three input trees [1]. MCT is NP-hard on two trees even if one of them is of maximum degree three [10] (see also [9]).

Consider now the general setting of an arbitrary number, denoted k, of input trees. Let $\mathcal{T} = \{T_1, T_2, \dots T_k\}$ be the input collection. Let n be the cardinality of the common leaf set of the T_i's, let $d := \min_{i=1}^{k} \Delta(T_i)$ and let $D := \max_{i=1}^{k} \Delta(T_i)$. Above, we argued about the relevance of solving MAST and MCT on bounded maximum degree trees. Three different algorithms were proposed to solve MAST in polynomial time for bounded d [1, 6, 3]. The fastest of these algorithms [6, 3] run in $O(n^d + kn^3)$ time.

Besides, MCT can be solved in $O(4^{kD}n^k)$ time [8]. Hence, for bounded k, MCT is FPT in D. The same result holds for MAST. Let p be a bound on the number of leaves to be removed from the input set of leaves so that the input trees agree, resp. are compatible. Then MAST, resp. MCT, can be solved in $O(\min\{3^p kn, \alpha^p + kn^3\})$ time, where α a constant less than 3 [2]. Thus, both problems are FPT with respect to p.

Our Contribution. We prove that both MAST and MCT are W[1]-hard with respect to D. Furthermore, let $\varphi : \mathbb{N} \rightarrow \mathbb{N}$ be an arbitrary recursive function. Note that the input $\mathcal{T}$ is of size $\widetilde{O}(kn)$. We prove the following.

($R1$). MAST cannot be solved in $\varphi(D)(kn)^{o(D)}$ time, unless SNP $\subseteq$ SE.

($R2$). MCT cannot be solved in $\varphi(D)(kn)^{o(2^{D/2})}$ time, unless SNP $\subseteq$ SE.

Recall that SE [11] is the class of problems solvable in subexponential time and that SNP [13] contains many NP-hard problems. Hence, the inclusion SNP $\subseteq$ SE is unlikely. According to result ($R1$), the $O(n^d + kn^3)$ time algorithms for MAST [6, 3] are somehow optimum. Results ($R1$) and ($R2$) are proved in sections 2 and 3 respectively.

1.2 Parameterized Complexity

In order to clearly prove our intractability results, we recall the main concepts of parameterized complexity [5], together with some recent results. We also introduce the notions of linear FPT-reduction and weak fixed-parameter tractability.

Let Σ be a finite alphabet. The set of all finite words over Σ is denoted $\Sigma^\star$ and, for each word $x \in \Sigma^\star$, $|x|$ denotes the *length* of x. A *parameterized (decision) problem* is a subset $P \subseteq \mathbb{N} \times \Sigma^\star$. For each *instance* $(k, x) \in \mathbb{N} \times \Sigma^\star$, k represents the *parameter*. A *yes-instance* of P is an element of P and a *no-instance* of P is an element of $(\mathbb{N} \times \Sigma^\star) - P$.

Fixed-Parameter Tractability and Weak Fixed-Parameter Tractability. An algorithm A *solves* the parameterized problem P if, for each input $(k, x) \in \mathbb{N} \times \Sigma^\star$, A can decide whether (k, x) is a yes-instance of P. The parameterized problem P is *fixed-parameter tractable* (FPT) if there exists an algorithm solving P, and whose running time is bounded by $\varphi(k)|x|^{O(1)}$ on each input $(k, x) \in \mathbb{N} \times \Sigma^\star$, where $\varphi : \mathbb{N} \rightarrow \mathbb{N}$ is recursive. The parameterized problem P is *weakly fixed-parameter tractable* (WFPT) if there exists an algorithm solving P, and whose running time is bounded by $\varphi(k)|x|^{o(k)}$ on each input $(k, x) \in \mathbb{N} \times \Sigma^\star$, where $\varphi : \mathbb{N} \rightarrow \mathbb{N}$ is recursive.

FPT-Reduction and Linear FPT-Reduction. Let P, $Q \subseteq \mathbb{N} \times \Sigma^\star$ be two parameterized problems and let $f : \mathbb{N} \times \Sigma^\star \to \mathbb{N} \times \Sigma^\star$.

We say that f is a (many-to-one, strongly uniform) *FPT-reduction* from P to Q if there exist recursive functions $g : \mathbb{N} \times \Sigma^\star \to \Sigma^\star$ and φ, $\gamma : \mathbb{N} \to \mathbb{N}$ satisfying, for all $(k, x) \in \mathbb{N} \times \Sigma^\star$:

1. $f(k, x)$ is computable in $\varphi(k)|x|^{O(1)}$ time,
2. $f(k, x) \in Q$ iff $(k, x) \in P$, and
3. $f(k, x) = (\gamma(k), g(k, x))$.

Moreover, if γ is at most linearly increasing (i.e. if $\gamma(k) = O(k)$ as $k \to \infty$) then we say that f is a *linear FPT-reduction* from P to Q.

FPT-reductions compose, and preserve fixed-parameter tractability. Linear FPT-reductions compose, and preserve weak fixed-parameter tractability. Note that our notion of linear FPT-reduction is slightly different from the one given by Chen, Huang, Kanj and Xia [4].

Independent Set. Formally, an (undirected) *graph* is an ordered pair $G = (V, E)$ where V is a finite set of *vertices* and where E a set of 2-subsets of V. The elements of E are the *edges* of G. The elements of an edge are called its *endpoints*. An *independent set* of G is a subset $I \subseteq V$ such that, for each edge $e \in E$, at least one of its endpoint is not in I. The problem of finding an independent set of maximum cardinality in a given input graph plays a central role in computational complexity theory, as well as its decision version:

Name: Independent Set (IS)
Instance: A positive integer k and a graph $G = (V, E)$.
Question: Is there an independent set of G of cardinality k?

The version of IS parameterized by k is denoted IS[k]. This problem is not believed to be FPT as it is complete under FPT-reductions for the class W[1] [5]. Moreover, IS[k] is not WFPT either, unless SNP $\subseteq$ SE [4].

2 Parameterized Complexity of MAST

The decision version of MAST is called Agreement SubTree (AST). The AST problem is: given an integer $q \geq 1$ and a finite collection $\mathcal{T}$ of trees on a common leaf set, decide whether there is an agreement subtree of $\mathcal{T}$ of size q. We denote by AST[D] the version of AST parameterized by $D := \max_{T \in \mathcal{T}} \Delta(T)$. In this section we prove: that AST[D] is W[1]-hard, and Result ($R1$) stated at the end of Section 1.1. According to Section 1.2, it is sufficient to linearly FPT-reduce AST[D] to IS[k] (Theorem 1 below).

For each integer $p \geq 1$, we introduce the following problem:

Name: Partitioned Independent Set with multiplicity p (PIS$_p$)
Instance: An integer $k \geq 1$, a graph $G = (V, E)$, and k independent sets V_1, V_2, ..., V_k of G of equal cardinality partitioning V.
Question: Is there an independent set I of G such that $I \cap V_i$ has cardinality p for all $i \in [1, k]$?

The version of PIS_p parameterized by k is denoted $\text{PIS}_p[k]$. IS[k] is reduced to AST[D] going through $\text{PIS}_1[k]$. In the next section, the decision version of MCT is reduced to IS going through PIS_2.

Lemma 1. IS[k] *linearly FPT-reduces to* $\text{PIS}_1[k]$.

Proof. Reduce IS[k] to $\text{PIS}_1[k]$ in the same way as Pietrzak reduces Clique to Partitioned Clique [14]. Each instance (k, G) of IS is transformed into an instance $(k, \widetilde{G}, \widetilde{V}_1, \widetilde{V}_2, \ldots, \widetilde{V}_k)$ of PIS_1 where $\widetilde{G}$ and the $\widetilde{V}_i$'s are as follows.

Let V be the vertex set of G. $\widetilde{G}$ is the graph on $V \times [1, k]$ whose edge set is given by: for all (u, i), $(v, j) \in V \times [1, k]$, $\{(u, i), (v, j)\}$ is an edge of $\widetilde{G}$ iff i is distinct from j, and either $\{u, v\}$ is an edge of G or $u = v$. For each $i \in [1, k]$, let $\widetilde{V}_i := V \times \{i\}$.

Validity of our reduction. Each $\widetilde{V}_i$ is an independent set of $\widetilde{G}$ with the same cardinality as V, and the $\widetilde{V}_i$'s partition the vertex set $V \times [1, k]$ of $\widetilde{G}$. Hence, $(k, \widetilde{G}, \widetilde{V}_1, \widetilde{V}_2, \ldots, \widetilde{V}_k)$ is an instance of $\text{PIS}_1[k]$. Moreover, it is clear that $(k, \widetilde{G}, \widetilde{V}_1, \widetilde{V}_2, \ldots, \widetilde{V}_k)$ is computable in polynomial time from (k, G). It remains to check that (k, G) is a yes-instance of IS iff $(k, \widetilde{G}, \widetilde{V}_1, \widetilde{V}_2, \ldots, \widetilde{V}_k)$ is a yes-instance of PIS_1.

- Assume there exists an independent set $\widetilde{I}$ of $\widetilde{G}$ such that $\widetilde{I} \cap \widetilde{V}_i$ is a singleton for all $i \in [1, k]$. For each $i \in [1, k]$, let $v_i \in V_i$ such that (v_i, i) is the unique element of $\widetilde{I} \cap \widetilde{V}_i$. The set $I := \{v_1, v_2, \ldots, v_k\}$ is an independent set of G of cardinality k.
- Conversely, assume that there exists an independent set I of G of cardinality k. Arbitrarily number the elements of I, i.e. write I as $I = \{v_1, v_2, \ldots, v_k\}$. The set $\widetilde{I} := \{(v_1, 1), (v_2, 2), \ldots, (v_k, k)\}$ is an independent set of $\widetilde{G}$ and $\widetilde{I} \cap \widetilde{V}_i = \{(v_i, i)\}$ is a singleton for all $i \in [1, k]$. □

In order to clearly prove Theorem 1, we first introduce some useful vocabulary.

Definition 1. *Let T and T' be two trees and let L be a subset of $L(T) \cap L(T')$. We say that T and T'* disagree *on L if $T|L$ and $T'|L$ are distinct.*

Assume that $L(T) \subseteq L(T')$. If there exists a subset $L \subseteq L(T)$ such that T and T' disagree on L then T is not a restriction of T'. Conversely, if T is a not restriction of T' then T and T' disagree on some 3-subset of $L(T)$ [3]. This explains the central role played by 3-leaf sets of disagreement in the proofs of lemmas 2 and 3 below. Note that given three distinct leaf labels a, b and c, there are exactly four distinct trees on $\{a, b, c\}$: the non-binary tree $\langle a, b, c\rangle$, and the three binary trees $\langle\langle b, c\rangle, a\rangle$, $\langle\langle a, c\rangle, b\rangle$ and $\langle\langle a, c\rangle, b\rangle$.

Theorem 1. IS[k] *linearly FPT-reduces to* AST[D].

Proof. According to Lemma 1, it suffices to linearly FPT-reduce $\text{PIS}_1[k]$ to AST[D]. Each instance $(k, G, V_1, V_2, \ldots, V_k)$ of PIS_1 is transformed into an instance $(q, \mathcal{T})$ of AST where $q := k$ and where $\mathcal{T}$ is a collection of trees described below. Without loss of generality, we can assume that all V_i's ($i \in [1, k]$) have cardinality at least 3 and that k is at least 3.

The collection $\mathcal{T}$. We construct a collection $\mathcal{T}$ of gadget trees whose leaf set is the vertex set $V := V_1 \cup V_2 \cup \ldots \cup V_k$ of G.

For each $i \in [1, k]$, compute an arbitrary *binary* tree B_i on V_i. The tree on V whose root admits B_1, B_2, ..., B_k as child subtrees is denoted by C: $C = \langle B_1, B_2, \ldots, B_k\rangle$. Every tree of $\mathcal{T}$ defined below can be obtained by modifying the positions of exactly two leaves of C.

For all a, $b \in V$ with $a \neq b$, $C_{a,b}$ denotes the tree on V obtained from C, by first removing its leaves a and b, and then re-grafting both of them as new children of the root. Formally, $C_{a,b}$ is the tree

$$\langle B_1|(V_1 - \{a, b\}), B_2|(V_2 - \{a, b\}), \ldots, B_k|(V_k - \{a, b\}), a, b\rangle \ .$$

We set $\mathcal{C} := \{C\} \cup \{C_{a,b} : a, b \in V, a \neq b\}$.

Remark 1. There exist at most two indices i such that $B_i|(V_i - \{a, b\})$ is distinct from B_i, and since V_i has cardinality at least 3, $B_i|(V_i - \{a, b\})$ is a non-empty tree for all i.

Let E be the edge set of G: $G = (V, E)$. For each edge $e = \{a, b\} \in E$, S_e denotes the tree on V obtained from C, by first removing its leaves a and b, and then re-grafting $\langle a, b\rangle$ as a new child of the root. Formally, S_e is the tree

$$\langle B_1|(V_1 - e), B_2|(V_2 - e), \ldots, B_k|(V_k - e), \langle a, b\rangle\rangle \ .$$

The collection of trees $\mathcal{T}$ is defined as $\mathcal{T} := \mathcal{C} \cup \{S_e : e \in E\}$ (see Figure 1): $\mathcal{C}$ is the *control component* of our gadget and the S_e's ($e \in E$) are its *selection components.*

Lemma 2 (Control). *Let T be a tree with leaf labels in V. Statements (i) and (ii) below are equivalent.*

(i). T is an agreement subtree of $\mathcal{C}$ of size k.
(ii). $T = \langle c_1, c_2, \ldots, c_k\rangle$ for some $(c_1, c_2, \ldots, c_k) \in V_1 \times V_2 \times \cdots \times V_k$.

Proof. Let $(c_1, c_2, \ldots, c_k) \in V_1 \times V_2 \times \cdots \times V_k$. Distinct c_i's appear in distinct child subtrees of the root of C, resp. of $C_{a,b}$. Hence, $\langle c_1, c_2, \ldots, c_k\rangle$ is a restriction of C, resp. of $C_{a,b}$. This proves that (ii) implies (i). It remains to show that (i) implies (ii).

Assume (i): T is an agreement subtree of $\mathcal{C}$ of size k.

- We first prove that T has height 1. By contradiction, suppose that T has height at least 2. Then, one can find three distinct leaves a, b, $c \in L(T)$ such that $T|\{a, b, c\} = \langle\langle a, b\rangle, c\rangle$. (Indeed, there exists an internal node ν of T which is not the root of T. Pick a leaf c which is not a descendant of ν and two descendant leaves a and b of ν.) However, $C_{a,b}|\{a, b, c\} = \langle a, b, c\rangle$, and thus T and $C_{a,b}$ disagree on $\{a, b, c\}$: contradiction.

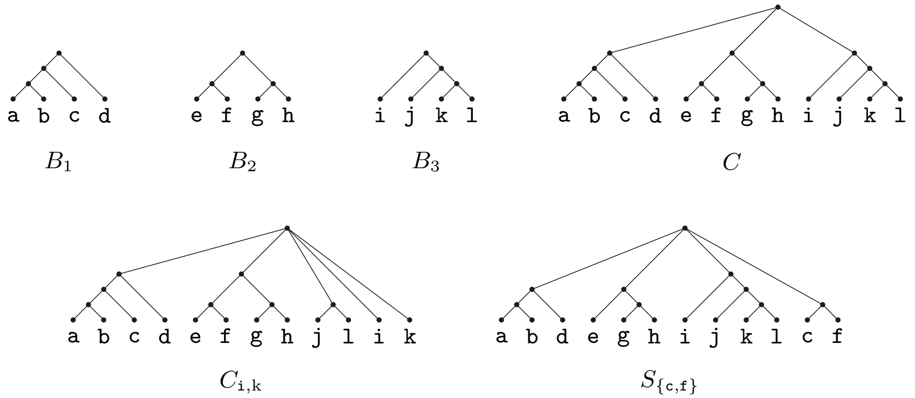


Fig. 1. Some of the gadget trees encoding an instance $(k, G, V_1, V_2, \ldots, V_k)$ of $\mathrm{PIS}_1[k]$ where $k = 3$, $V_1 = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$, $V_2 = \{\mathtt{e}, \mathtt{f}, \mathtt{g}, \mathtt{h}\}$, $V_3 = \{\mathtt{i}, \mathtt{j}, \mathtt{k}, \mathtt{l}\}$ and $\{\mathtt{c}, \mathtt{f}\}$ is an edge of G

Since T has height 1, there exist k distinct leaf labels c_1, c_2, …, $c_k \in V$ such that $T = \langle c_1, c_2, \ldots, c_k \rangle$.

- We now show that distinct c_j's belong to distinct V_i's. By contradiction, assume there exist i, j_1, $j_2 \in [1, k]$ with $j_1 \neq j_2$, and such that c_{j_1} and c_{j_2} both belong to V_i. Since k is at least 3, one can find $j \in [1, k]$ distinct from j_1 and j_2. If $c_j \in V_i$ then $C|\{c_{j_1}, c_{j_2}, c_j\} = B_i|\{c_{j_1}, c_{j_2}, c_j\}$ and if $c_j \notin V_i$ then $C|\{c_{j_1}, c_{j_2}, c_j\} = \langle\langle c_{j_1}, c_{j_2}\rangle, c_j\rangle$. In both cases, $C|\{c_{j_1}, c_{j_2}, c_j\}$ is a binary tree unlike to $T|\{c_{j_1}, c_{j_2}, c_j\}$. Thus, C and T disagree on $\{c_{j_1}, c_{j_2}, c_j\}$: contradiction.

Up to a permutation of the c_i's, one has $(c_1, c_2, \ldots, c_k) \in V_1 \times V_2 \times \cdots \times V_k$. This proves (ii) and concludes the proof of Lemma 2. □

Lemma 3 (Selection). *Let $e \in E$ be an edge of G and let $(c_1, c_2, \ldots, c_k) \in V_1 \times V_2 \times \cdots \times V_k$. The tree $\langle c_1, c_2, \ldots, c_k \rangle$ is a restriction of S_e iff at least an endpoint of e is not in $\{c_1, c_2, \ldots, c_k\}$.*

Proof. The "if part" is easy. Let us now show the "only if" part.

Assume that $\langle c_1, c_2, \ldots, c_k \rangle$ is a restriction of S_e and that $e \subseteq \{c_1, c_2, \ldots, c_k\}$. Let c_{i_1} and c_{i_2} be the two endpoints of e: $e = \{c_{i_1}, c_{i_2}\}$. Since k is at least 3, there exists $i \in [1, k]$ such that c_i is distinct from c_{i_1} and c_{i_2}. $S_e|\{c_{i_1}, c_{i_2}, c_i\} = \langle\langle c_{i_1}, c_{i_2}\rangle, c_i\rangle$, and thus S_e disagrees with $\langle c_1, c_2, \ldots, c_k \rangle$ on $\{c_{i_1}, c_{i_2}, c_i\}$: contradiction. This concludes the proof of Lemma 3. □

Validity of our reduction. It is clear that $(q, \mathcal{T})$ is computable in polynomial time from $(k, G, V_1, V_2, \ldots, V_k)$. Moreover, the root of C has degree k, the root of $C_{a,b}$ has degree $k + 2$, the root of S_e has degree $k + 1$, and any non-root internal node of a tree in $\mathcal{T}$ has degree 2. Hence, the maximum degree D of all trees in

$\mathcal{T}$ is equal to $k+2$: $D = O(k)$. Eventually, it follows from lemmas 2 and 3 that: $(k, G, V_1, V_2, \ldots, V_k)$ is a yes-instance of PIS$_1$ iff $(q, \mathcal{T})$ is a yes-instance of AST.

- Indeed, assume there exists an agreement subtree T of $\mathcal{T}$ of size $q = k$. The tree T is of the form $T = \langle c_1, c_2, \ldots, c_k \rangle$ for some $(c_1, c_2, \ldots, c_k) \in V_1 \times V_2 \times \cdots \times V_k$ by Lemma 2. Furthermore, the set $I := \{c_1, c_2, \ldots, c_k\}$ is an independent set of G by Lemma 3, and for every $i \in [1, k]$, $I \cap V_i = \{c_i\}$ is a singleton.
- Conversely, assume that there exists an independent set I of G such that $I \cap V_i$ is a singleton for all $i \in [1, k]$. Hence, there exists $(c_1, c_2, \ldots, c_k) \in V_1 \times V_2 \times \cdots \times V_k$ such that $I = \{c_1, c_2, \ldots, c_k\}$. The tree $\langle c_1, c_2, \ldots, c_k \rangle$ is both:
  - an agreement subtree of $\mathcal{C}$ (Lemma 2), and
  - an agreement subtree of $\{S_e : e \in E\}$ (Lemma 3).

Therefore, $\langle c_1, c_2, \ldots, c_k \rangle$ is an agreement subtree of $\mathcal{T}$ of size q. □

3 Parameterized Complexity of MCT

The decision version of MCT is called Compatible Tree (CT). The CT problem is: given an integer $q \geq 1$ and a finite collection $\mathcal{T}$ of trees on a common leaf set, decide whether there is a tree of size q compatible with $\mathcal{T}$. We denote by CT$[2^{\lfloor D/2 \rfloor}]$ the version of CT parameterized by $2^{\lfloor D/2 \rfloor}$ where $D := \max_{T \in \mathcal{T}} \Delta(T)$. In this section, IS$[k]$ is linearly FPT-reduced to CT$[2^{\lfloor D/2 \rfloor}]$ in order to obtain: the W[1]-hardness of the version of CT parameterized by D, and Result $(R2)$ stated at the end of Section 1.1. PIS$_2$ is used as an intermediate problem.

Lemma 4. IS$[k]$ *linearly FPT-reduces to* PIS$_2[k]$.

Proof. According to Lemma 1, it suffices to linearly FPT-reduce PIS$_1[k]$ to PIS$_2[k]$. We rely on a padding argument. Each instance $(k, G, V_1, V_2, \ldots, V_k)$ of PIS$_1$ is transformed into an instance $(k, \widetilde{G}, \widetilde{V}_1, \widetilde{V}_2, \ldots, \widetilde{V}_k)$ of PIS$_2$ where $\widetilde{G}$ and the $\widetilde{V}_i$'s are as follows.

Let $V := V_1 \cup V_2 \cup \ldots \cup V_k$ be the vertex set of G and let E be its edge set: $G = (V, E)$. Informally, $\widetilde{G}$ is obtained by adding k isolated vertices to G, and each $\widetilde{V}_i$ is obtained by adding a single one of these new vertices to V_i. More formally, let a_1, a_2, $\ldots$, a_k be k pairwise distinct elements not belonging to V. We construct $\widetilde{G} := (V \cup \{a_1, a_2, \ldots, a_k\}, E)$, and $\widetilde{V}_i := V_i \cup \{a_i\}$ for each $i \in [1, k]$.

Validity of our reduction. It is clear that $(k, \widetilde{G}, \widetilde{V}_1, \widetilde{V}_2, \ldots, \widetilde{V}_k)$ is an instance of PIS$_2$ computable in polynomial time from $(k, G, V_1, V_2, \ldots, V_k)$. It remains to check that $(k, G, V_1, V_2, \ldots, V_k)$ is a yes-instance of PIS$_1$ iff $(k, \widetilde{G}, \widetilde{V}_1, \widetilde{V}_2, \ldots, \widetilde{V}_k)$ is a yes-instance of PIS$_2$.

- Assume that there exists an independent set I of G such that $I \cap V_i$ is a singleton for every $i \in [1, k]$. Then $\widetilde{I} := I \cup \{a_1, a_2, \ldots, a_k\}$ is an independent set of $\widetilde{G}$, and $\widetilde{I} \cap \widetilde{V}_i$ is a doubleton for all $i \in [1, k]$.
- Conversely, assume that there exists an independent set $\widetilde{I}$ of $\widetilde{G}$ such that $\widetilde{I} \cap \widetilde{V}_i$ is a doubleton for every $i \in [1, k]$. For each $i \in [1, k]$, pick an element v_i in $\widetilde{I} \cap \widetilde{V}_i$ distinct from a_i. The set $I := \{v_1, v_2, \ldots, v_k\}$ is an independent set of G, and $I \cap V_i = \{v_i\}$ is a singleton for all $i \in [1, k]$. □

Remark 2. It is easy to see that the mapping $(k, G, V_1, V_2, \ldots, V_k) \longmapsto (k, \widetilde{G}, \widetilde{V}_1, \widetilde{V}_2, \ldots, \widetilde{V}_k)$, defined in the proof of Lemma 4 is a linear FPT-reduction from $\text{PIS}_p[k]$ to $\text{PIS}_{p+1}[k]$ for *any* integer $p \geq 1$. Since IS[k] linearly FPT-reduces to $\text{PIS}_1[k]$ (Lemma 1), an induction on p ensures that IS[k] linearly FPT-reduces to $\text{PIS}_p[k]$ for any integer $p \geq 1$.

In order to linearly FPT-reduce IS to CT, we introduce some useful notations in definitions 2, 3 and 4.

Definition 2. *Let T be a tree on $[1, n]$, and let T_1, T_2, ..., T_n be non-empty trees with pairwise disjoint leaf sets. The tree on $L(T_1) \cup L(T_2) \cup \ldots \cup L(T_n)$, obtained by replacing each leaf i in T by T_i, is denoted $T[T_1, T_2, \ldots, T_n]$.*

For instance, let $T := \langle\langle 1, 2\rangle, \langle 3, \langle 4, 5\rangle\rangle, 6\rangle$ and let T_1, T_2, T_3, T_4, T_5, T_6 be non-empty trees with pairwise disjoint leaf sets: $T[T_1, T_2, T_3, T_4, T_5, T_6] = \langle\langle T_1, T_2\rangle, \langle T_3, \langle T_4, T_5\rangle\rangle, T_6\rangle$.

Definition 3. *For each integer $n \geq 1$, R_n denotes the binary tree on $[1, n]$, defined recursively as follows: $R_1 = 1$, and if n is at least 2, then $R_n = \langle R_{n-1}, n\rangle$.*

According to Definition 3, one has $R_2 = \langle 1, 2\rangle$, $R_3 = \langle\langle 1, 2\rangle, 3\rangle$, $R_4 = \langle\langle\langle 1, 2\rangle, 3\rangle, 4\rangle$, $R_5 = \langle\langle\langle\langle 1, 2\rangle, 3\rangle, 4\rangle, 5\rangle$, $R_4[\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}] = \langle\langle\langle \mathtt{a}, \mathtt{b}\rangle, \mathtt{c}\rangle, \mathtt{d}\rangle$, ...

Property 1. Let v^1, v^2, ..., v^n be n pairwise distinct labels. A tree with leaf labels in $\{v^1, v^2, \ldots, v^n\}$ is compatible with $\left\{R_n[v^1, v^2, \ldots, v^n], R_n[v^n, \ldots, v^2, v^1]\right\}$ iff it is of size at most two.

Definition 4. *Let k be a positive integer. H_k denotes a binary tree on $[1, k]$ of minimum height $\lceil \log k \rceil$. For each i, $j \in [1, k]$, $H_k^{i,j}$ denotes the tree on $[1, k]$ obtained from H_k by collapsing all internal edges on the path connecting i and j. The least common ancestor of i and j in $H_k^{i,j}$ is denoted $\lambda_k^{i,j}$.*

Property 2. All internal nodes in $H_k^{i,j}$ are of degree 2, except maybe $\lambda_k^{i,j}$ whose degree is at most $2\lceil \log k \rceil$.

Theorem 2. IS[k] *linearly FPT-reduces to* CT[$2^{\lfloor D/2 \rfloor}$].

Proof. According to Lemma 4, it suffices to linearly FPT-reduce $\text{PIS}_2[k]$ to CT[$2^{\lfloor D/2 \rfloor}$]. Each instance $(k, G, V_1, V_2, \ldots, V_k)$ of $\text{PIS}_2[k]$ is transformed into an instance $(q, \mathcal{T})$ of CT where $q := 2k$ and where $\mathcal{T}$ is a collection of trees described below.

The collection $\mathcal{T}$. We construct a collection $\mathcal{T}$ of gadget trees on the vertex set $V := V_1 \cup V_2 \cup \ldots \cup V_k$ of G. Let n be such that V_i has cardinality n for every $i \in [1, k]$. For each $i \in [1, k]$, arbitrarily order V_i, i.e. write V_i as $V_i = \left\{v_i^1, v_i^2, \ldots, v_i^n\right\}$; $B_i := R_n[v_i^1, v_i^2, \ldots, v_i^n]$ and $\widetilde{B}_i := R_n[v_i^n, \ldots, v_i^2, v_i^1]$ encode V_i.

Let $C := H_k[B_1, B_2, \ldots, B_k]$ and let $\widetilde{C} := H_k[\widetilde{B}_1, \widetilde{B}_2, \ldots, \widetilde{B}_k]$ (see Figure 2): C and $\widetilde{C}$ are the *control components* of our gadget.

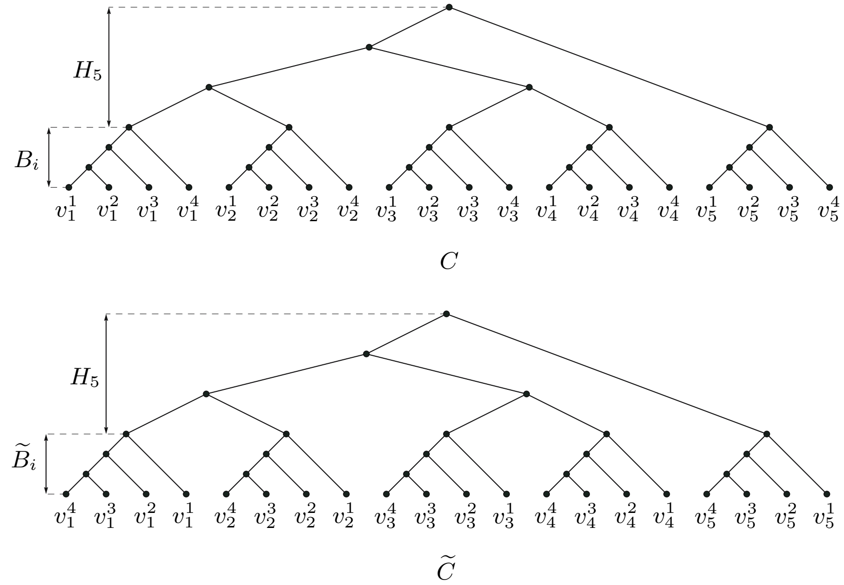

H_5
B_i
v_1^1 v_1^2 v_1^3 v_1^4 v_2^1 v_2^2 v_2^3 v_2^4 v_3^1 v_3^2 v_3^3 v_3^4 v_4^1 v_4^2 v_4^3 v_4^4 v_5^1 v_5^2 v_5^3 v_5^4
C
H_5
$\widetilde{B}_i$
v_1^4 v_1^3 v_1^2 v_1^1 v_2^4 v_2^3 v_2^2 v_2^1 v_3^4 v_3^3 v_3^2 v_3^1 v_4^4 v_4^3 v_4^2 v_4^1 v_5^4 v_5^3 v_5^2 v_5^1
$\widetilde{C}$

Fig. 2. The trees C and $\widetilde{C}$ in the case of $k = 5$ and $n = 4$

Let E be the edge set of G: $G = (V, E)$. For each edge $e = \{v_i^r, v_j^s\} \in E$, compute the tree S_e obtained from $H_k^{i,j}[B_1, B_2, \ldots, B_k]$ by first removing its leaves v_i^r and v_j^s, and then re-grafting $\left\langle v_i^r, v_j^s \right\rangle$ as a new child subtree of $\lambda_k^{i,j}$ (see Figure 3). The S_e's ($e \in E$) are the *selection components* of our gadget.

The collection of trees $\mathcal{T}$ is defined as $\mathcal{T} := \{C, \widetilde{C}\} \cup \{S_e : e \in E\}$.

Property 3 below is easily deduced from Property 1.

Property 3 (Control). Let T be a tree with leaf labels in V. Statements (i) and (ii) below are equivalent.

(i). T is a tree of size q, compatible with $\{C, \widetilde{C}\}$.

(ii). T is of the form $T = H_k[\langle a_1, b_1 \rangle, \langle a_2, b_2 \rangle, \ldots, \langle a_k, b_k \rangle]$ where, for each $i \in [1, k]$, a_i and b_i are two distinct elements of V_i.

Property 4 (Selection). Let $e \in E$ be an edge of G and let T be a tree of size q compatible with $\{C, \widetilde{C}\}$. Then, T refines $S_e|L(T)$ iff at least an endpoint of e is not in $L(T)$.

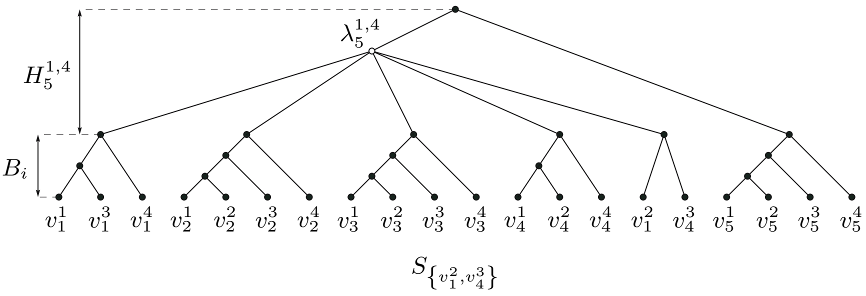

$\lambda_5^{1,4}$
$H_5^{1,4}$
B_i
v_1^1 v_1^3 v_1^4 v_2^1 v_2^2 v_2^3 v_2^4 v_3^1 v_3^2 v_3^3 v_3^4 v_4^1 v_4^2 v_4^4 v_1^2 v_4^3 v_5^1 v_5^2 v_5^3 v_5^4
$S_{\{v_1^2,v_4^3\}}$

Fig. 3. The tree $S_{\{v_1^2,v_4^3\}}$ in the case of $k = 5$ and $n = 4$

It is clear that $(q, \mathcal{T})$ is computable in polynomial time from $(k, G, V_1, V_2, \dots, V_k)$. Moreover, both C and $\widetilde{C}$ are binary, and all internal nodes in S_e have degree 2, except maybe $\lambda_k^{i,j}$ whose degree is at most $2\lceil\log k\rceil + 1$ (see Property 2). Hence, the maximum degree D of all trees in $\mathcal{T}$ is at most $2\lceil\log k\rceil + 1$, and thus $2^{\lfloor D/2 \rfloor} = O(k)$. Eventually, it remains to show that: $(k, G, V_1, V_2, \dots, V_k)$ is a yes-instance of PIS$_2$ iff $(q, \mathcal{T})$ is a yes-instance of AST.

- Assume that there exists a tree T of size q compatible with $\mathcal{T}$. Let $I := L(T)$: for every $i \in [1, k]$, $I \cap V_i$ is a doubleton by Property 3, and I is an independent set of G by Property 4.
- Conversely, assume that there exists an independent set I of G such that $I \cap V_i$ is a doubleton for all $i \in [1, k]$. For each $i \in [1, k]$, let a_i and b_i be the two elements of $I \cap V_i$. The tree $T := H_k[\langle a_1, b_1\rangle, \langle a_2, b_2\rangle, \dots, \langle a_k, b_k\rangle]$ is compatible with $\{C, \widetilde{C}\}$ according to Property 3. Furthermore, T is also compatible with $\{S_e : e \in E\}$ according to Property 4. We have thus exhibited a tree T of size q compatible with $\mathcal{T}$.

Remark 3. $2^{\lfloor D/2 \rfloor} = O(k)$ is enough to obtain Result ($R2$). But, our construction does not ensure that $2^{\lfloor D/2 \rfloor}$ is a function of k only. Hence, our reduction is not exactly an FPT-reduction yet. Anyway, this can be easily repaired. Collapse $2\lceil\log k\rceil - 1$ consecutive internal edges in B_1 to obtain a tree B_1' of maximum degree $2\lceil\log k\rceil + 1$ and add to $\mathcal{T}$ the tree $C' := H_k[B_1', B_2, \dots, B_k]$. □

References

1. A. Amir and D. Keselman. Maximum agreement subtree in a set of evolutionary trees: metrics and efficient algorithm. *SIAM Journal on Computing*, 26(6):1656–1669, 1997.
2. V. Berry and F. Nicolas. Maximum agreement and compatible supertrees. In S. C. Sahinalp, S. Muthukrishnan, and U. Dogrusoz, editors, *Proceedings of The 15th Annual Symposium on Combinatorial Pattern Matching (CPM'04)*, volume 3109 of *Lecture Notes in Computer Science*, pages 205–219. Springer-Verlag, 2004.

3. D. Bryant. *Building trees, hunting for trees and comparing trees: theory and method in phylogenetic analysis.* PhD thesis, University of Canterbury, Department of Mathemathics, 1997.
4. J. Chen, X. Huang, I. A. Kanj, and G. Xia. Linear FPT reductions and computational lower bounds. In L. Babai, editor, *Proceedings of the 36th Annual ACM Symposium on Theory of Computing (STOC'04)*, pages 212–221. ACM Press, 2004.
5. R. G. Downey and M. R. Fellows. *Parameterized Complexity.* Monographs in Computer Science. Springer, 1999.
6. M. Farach, T. M. Przytycka, and M. Thorup. On the agreement of many trees. *Information Processing Letters*, 55(6):297–301, 1995.
7. C. R. Finden and A. D. Gordon. Obtaining common pruned trees. *Journal of Classification*, 2:255–276, 1985.
8. G. Ganapathysaravanabavan and T. J. Warnow. Finding a maximum compatible tree for a bounded number of trees with bounded degree is solvable in polynomial time. In O. Gascuel and B. M. E. Moret, editors, *Proceedings of the 1st International Workshop on Algorithms in Bioinformatics (WABI'01)*, volume 2149 of *Lecture Notes in Computer Science*, pages 156–163. Springer-Verlag, 2001.
9. A. M. Hamel and M. A. Steel. Finding a maximum compatible tree is NP-hard for sequences and trees. *Applied Mathematics Letters*, 9(2):55–59, 1996.
10. J. Hein, T. Jiang, L. Wang, and K. Zhang. On the complexity of comparing evolutionary trees. *Discrete Applied Mathematics*, 71(1–3):153–169, 1996.
11. R. Impagliazzo, R. Paturi, and F. Zane. Which problems have strongly exponential complexity? *Journal of Computer and System Sciences*, 63(4):512–530, 2001.
12. M.-Y. Kao, T. W. Lam, W.-K. Sung, and H.-F. Ting. An even faster and more unifying algorithm for comparing trees via unbalanced bipartite matchings. *Journal of Algorithms*, 40(2):212–233, 2001.
13. C. H. Papadimitriou and M. Yannakakis. Optimization, approximation, and complexity classes. *Journal of Computer and System Sciences*, 43(3):425–440, 1991.
14. K. Pietrzak. On the parameterized complexity of the fixed alphabet shortest common supersequence and longest common subsequence problems. *Journal of Computer and System Sciences*, 67(4):757–771, 2003.

An Improved Algorithm for the Macro-evolutionary Phylogeny Problem

Behshad Behzadi and Martin Vingron

Computational Molecular Biology Department,
Max Planck Institute for Molecular Genetics,
Ihnestrasse 73, 14195 Berlin, Germany
{behshad.behzadi, martin.vingron}@molgen.mpg.de

Abstract. Macro-evolutionary processes (e.g., gene duplication and loss) have rarely been incorporated into gene phylogeny reconstruction methods. Durand et al. [5] have proposed a polynomial time dynamic programming algorithm to find the gene family tree that optimizes a macro-evolutionary criterion which is the weighted sum of the number of gene duplications and losses. The complexity of this algorithm is $O(nm^2)$ where n is the number of species and m is the maximum number of copies of the gene in a species. In this paper, we propose an improved algorithm with time complexity of $O(nm)$ for solving this problem. We also show, that the problem can be solved in $O(n)$ if unit costs are considered for both loss and duplication.

1 Introduction

One of the main goals of evolutionary biology is the reconstruction of the evolutionary history of the current species. Based on the assumption of common ancestors, this history can be represented as a tree, called a phylogenetic tree. The internal nodes correspond to ancestral species and the leaves are the current species. With the rise of molecular biology, DNA sequences of genes have been available. These sequences can be treated as characters from which one can estimate phylogenetic trees (see, for example, [7, 8, 15]). However determining which genes are comparable can be a problematic. There exists a large number of related genes that have evolved through the process of *gene duplication*. Once a gene has been duplicated, each copy can evolve distinct variations. Distinct copies of the same gene are called paralogues. As a result, a single species may contain none, one, or several copies of what was a single gene in an ancestor. In order to build a tree which reflects the evolution of species containing a given gene, it is essential to know which copies of the gene are the comparable ones. The tree explaining the evolution of a set of genes is called *gene tree* and the tree describing the evolution of species is the *species tree*. The gene trees and species tree may be different in topology because they present evolutionary relations of different entities. This is mainly due to the gene duplications and losses which are known also as the main *macro-evolutionary* events.

A combination of two types of events should be considered in the determination of the evolutionary history of a gene family: micro-evolutionary events

M. Lewenstein and G. Valiente (Eds.): CPM 2006, LNCS 4009, pp. 177–187, 2006.

(sequence evolution) and macro-evolutionary events (e.g., gene duplication and loss). A good review of the past models considering both micro- and macro-evolutionary operations can be found in [5]. The term *tree reconciliation* was introduced by Goodman et al. [9] for mapping a gene tree to a species tree. Algorithms and combinatorial properties of the reconciliated trees have been extensively studied (see for example [2, 6, 10, 11, 12, 13, 14, 16]). Arvestad et al. [1] have proposed a Bayesian approach for consideration of both macro- and micro-evolutionary events in a *unified* model. A good estimation of the parameters in the Bayesian approach needs large datasets. The expensive computation time is another problem of the Bayesian approach. On the other hand, as stated in [5] a unified Bayesian model is a strength when both sequence evolution and gene duplication and loss can be modeled by neutral stochastic process which is not the case for the data under strong selective pressure.

In a recent work on unified models, Durand et al. in [5] have developed a hybrid (two-phase) approach to gene tree reconstruction that incorporates sequence evolution, gene duplication and gene loss for the reconstruction of phylogenies. This unified approach is mainly based on a dynamic programming algorithm they propose to find all most parsimonious phlyogenies w.r.t a macro-evolutionary model of gene duplication and loss. The number of members of the gene family in each species is given in the input; the output is a tree (or trees) with fewest duplications and losses required to explain the data. Note that the existence of a polynomial time algorithm for this problem is interesting because most of the problems in phylogeny reconstruction are NP-Hard [3, 4]. In the first phase of the hybrid approach, only the micro-evolutionary events are considered and a tree is constructed using the sequence evolution operations. The regions of the tree which are not strongly supported by the sequence data are refined with respect to a macro-evolutionary parsimony model in the second phase of the approach. The parts of the tree with strong support are left intact. The macro-evolutionary events are used only for explaining the areas where the sequence data cannot resolve the topology, so the total search space is reduced. As a result, this method considers both types of events with modest computational requirements.

In this paper, we suggest a faster algorithm for the macro-evolutionary phylogeny problem defined in [5] which improves the overall time of reconstruction of the tree considering micro- and macro-evolutionary events. The worst case running time of our algorithm is a factor of m smaller than the previous worst case, where m is the maximum number of the gene copies in a species. Note that an improvement of $O(m)$ is important because a gene family (like, e.g. kinases) can have a large number of duplicates. We will focus on the combinatorial properties of the structure of optimal histories of the macro-evolutionary phylogeny problem. Using these properties, we propose an improved algorithm for solving this problem.

The paper is organized as follows: in Section 2, we describe the model and present the formal definition of the problem. In Section 3, we present some combinatorial properties of the optimal answers which will be useful for our algorithm design. In Section 4, we propose an improved algorithm for solving the macro-evolutionary phylogeny problem. We show that our improvement can be even more in the case of the unit cost duplication and loss events. Finally Section 5, goes to the conclusions.

2 Problem Description

As stated in the introduction, a single current species may have zero, one, or several copies of what was a single gene in an ancestor. The macro-evolutionary phylogeny problem tries to explain these different multiplicities of the genes in the current species by duplications and losses. Note that as the sequence information (sequence evolution) is not considered in this model, only the number of present genes of each species and the species tree is enough for the computations. There are infinitely many different histories which can be considered to generate the given numbers of copies of a gene in the current species. But here we will be interested by the histories which use the fewest number of losses and duplications for explaining the data.

The optimization criterion is based on the cost of duplication and loss. Let the cost of a duplication be denoted by c_δ and the cost of a loss be denoted by c_λ. The *D/L score* of a gene tree is $c_\lambda L + c_\delta D$, the weighted sum of the number of duplications, D, and the number of of losses, L, in the tree.

Each history can be represented as a species tree, where each node is annotated with its multiplicity; that is the number of gene copies extant in the species associated with that node. The multiplicity of the root must be one, while the multiplicity of the leaves, denoted $m_1, ..., m_s$ are specified in the input. One should note that each duplication operation, increases the number of gene copies in the species by one. Similarly, each loss decreases the number of gene copies of the species associated to a node. Another way to see the problem is to consider for each node the number of gene copies which it has received from its parent node and the number of genes it passes to its children. Suppose that i gene copies exist in species x and it passes j copies to each of its children. If $j > i$, $j - i$ duplications are needed to explain this change. When $j < i$, $i - j$ losses are needed to explain this part of the tree. Finally, if $i = j$, the minimal cost event can be a speciation (without any loss or duplication).

In Figure 1(a), a species tree together with multiplicities of a given gene in the current species is given. In Figure 1.b and 1.c, two macro-evolutionary histories are considered for this species tree. The tree (1.b) explains the data by one duplication and one loss while the tree (1.c) uses two duplications. If $c_\lambda > c_\delta$ then tree (1.c) is the optimal history. If $c_\lambda < c_\delta$ then tree (1.b) is the optimal history. Finally if $c_\lambda = c_\delta$ both trees are optimal histories of this species tree.

The formal definition of the macro-evolutionary phylogeny problem as defined in [5] is as follows:

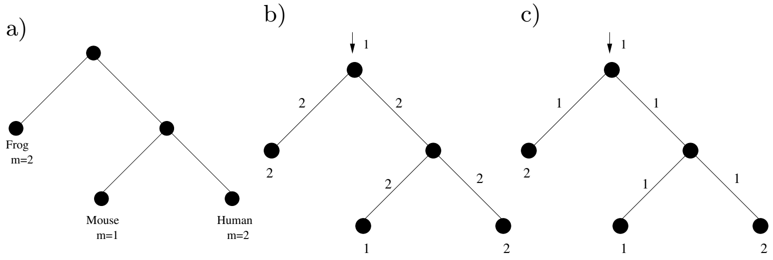


Fig. 1. a) A species tree with three species and multiplicities for each species; b) an annotation of the species tree with the number of gene copies: one duplication at root and one loss in mouse; c) another alternative of annotation (history): two duplications in human and in frog. Depending on the costs of duplication and loss, the optimal history for tree (a) may be tree (b) or (c) or both of them.

Macro-evolutionary Phylogeny Problem
Input: A rooted species tree, T_S with s leaves; a list of multiplicities $m_1, ..., m_s$, where m_l is the number of gene family members found in species l; weights c_λ and c_δ.
Output: The set of all rooted gene trees $\{T_G\}$ with $\Sigma_{l=1}^{s} m_l$ leaves such that D/L Score of T_G is minimal.

As mentioned above, the output can be represented only by annotation of the species tree by the number of gene copies in different nodes and number of genes a node passes to its children. Through this paper we use the *entering number of genes* for a given subtree rooted at v to denote the number of genes that node v has received from its parent. The entering number of genes for root in a history should be one.

In [5] the authors propose a dynamic programming approach for solving the macro-evolutionary phylogeny problem. The idea is to consider all possible multiplicities for each internal node. The algorithm fills a table $Cost[v, i, j]$ for this aim which is the minimum D/L score for the subtree rooted at v where v has i entering copies and it passes j gene copies to each of its children. The table entries are computed for any node v of the tree and any two numbers $1 \leq i, j \leq m$ where m is the maximum multiplicity for a leaf of the tree. Once this table is filled recursively, reconstructing the gene trees using this table is immediate. The complexity of the dynamic programming part (and the whole algorithm) is $O(nm^2)$ for giving one optimal history and $O(nm^2 + nmk)$ for reporting k optimal histories.

In this work we show that one can compute faster the optimal histories without filling the dynamic programming table $Cost[v, i, j]$. In Section 3, we study the properties the minimal generating cost function for the optimal histories. This leads us to an algorithm which runs $O(m)$ times faster than the previous algorithm. Then we show that this complexity can still be improved by an additional factor of $O(m)$ for the unit duplication/loss function ($c_\lambda = c_\delta = 1$).

3 Properties of Optimal Histories

In this Section, we study the properties of function $g(x, \mathcal{T})$ defined as follows:

Definition 1. *For a given tree $\mathcal{T}$ and a given list of multiplicities for its leaves, $g(x, \mathcal{T})$ is defined to be the minimum D/L score for duplication/loss history of tree $\mathcal{T}$ where the root of the tree $\mathcal{T}$ has x entering copies of genes.*

In an optimal history, in a given node there either is no event or a first duplication (or loss) event is followed by an optimal history. If we denote the left and the right subtree of tree $\mathcal{T}$ by $\mathcal{T}_L$ and $\mathcal{T}_R$ respectively, then we have:

$$g(x, \mathcal{T}) = \min \begin{cases} g(x+1, \mathcal{T}) + c_\delta & (1) \\ g(x-1, \mathcal{T}) + c_\lambda & (2) \\ g(x, \mathcal{T}_L) + g(x, \mathcal{T}_R) & (3) \end{cases} \tag{1}$$

Note that as the costs are positive ($c_\delta, c_\lambda > 0$), for a sufficiently large N, $g(x, \mathcal{T}_L) + g(x, \mathcal{T}_R)$ is smaller than $g(x+N, \mathcal{T}) + Nc_\delta$ and $g(x-N, \mathcal{T}) + Nc_\lambda$. This shows that the recurrences are finite for any tree $\mathcal{T}$ and integer x because $\mathcal{T}_L$ and $\mathcal{T}_R$ are smaller than $\mathcal{T}$. The following inequalities are the immediate result of this Equation (1):

$$\begin{aligned} g(x, \mathcal{T}) &\leq g(x+k, \mathcal{T}) + k.c_\delta \\ g(x, \mathcal{T}) &\leq g(x-k, \mathcal{T}) + k.c_\lambda \end{aligned} \tag{2}$$

Let us define the optimal generating set of entering genes number for a tree as follows:

Definition 2. *For a given tree $\mathcal{T}$ and a given list of multiplicities for its leaves, $OPT(\mathcal{T})$ is defined as the set of all integers x such that for any integer x', $g(x, \mathcal{T}) \leq g(x', \mathcal{T})$. This optimal cost itself is denoted by $opt(\mathcal{T})$.*

The following inequality relates the optimal cost of a tree with the optimal costs of its children.

Lemma 1. *Let $\mathcal{T}$ be a binary tree and $\mathcal{T}_L$ (resp. $\mathcal{T}_R$) be its left (resp. right) subtree. We have $opt(\mathcal{T}) \geq opt(\mathcal{T}_L) + opt(\mathcal{T}_R)$.*

This is due to the fact that the optimal history of $\mathcal{T}$ includes a generation of $\mathcal{T}_L$ and a generation of $\mathcal{T}_R$ as a part of it. The proof is easy and is omitted.

We will prove that $OPT(\mathcal{T})$ is an integer interval (a set of consecutive integer numbers) for any tree $\mathcal{T}$. We denote the integer interval $\{x, x+1, ..., y\}$ by $[x, y]$ for any $x \leq y$.

Proposition 1. *For any tree $\mathcal{T}$ with given input multiplicities for the leaves, $OPT(\mathcal{T})$ is an integer interval.*

Proposition 1, states that the function $g(x, \mathcal{T})$ is minimum in an integer interval $[x_1, x_2]$ which is denoted by $OPT(\mathcal{T})$. We show that the function $g(x, \mathcal{T})$, is strictly decreasing for $x \leq x_1$ and strictly increasing for $x \geq x_2$.

Proposition 2. *Let $\mathcal{T}$ be a binary tree with given multiplicities for leaves and let $OPT(\mathcal{T})$ be $[x_1, x_2]$. The function $g(x, \mathcal{T})$ is strictly decreasing for all x smaller than x_1 and strictly increasing for all x larger than x_2.*

We will also show that $g(x, \mathcal{T})$ is a convex function, which is $\Delta g(x, \mathcal{T}) = g(x + 1, \mathcal{T}) - g(x, \mathcal{T})$ is an increasing function.

Proposition 3. *Let $\mathcal{T}$ be a binary tree with given multiplicities for leaves. $g(x, \mathcal{T})$ is a convex function.*

These three propositions together show that the general structure of function $g(x, \mathcal{T})$ is like the function given in Figure 2.

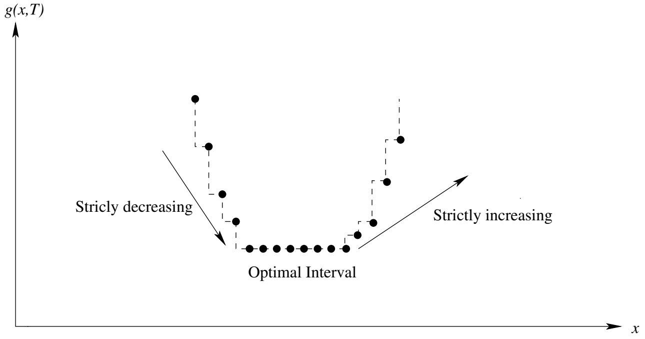


Fig. 2. The general structure of $g(x,T)$. In range $(-\infty, +\infty)$, $g(x, \mathcal{T})$ is firstly strictly decreasing then it takes its minimum on an interval (which may be just one point) and then it is strictly increasing. The function is convex; $\Delta g(x, \mathcal{T})$ is increasing. For large values of x, we have $\Delta g(x, \mathcal{T}) = c_\lambda$. For sufficiently small values of x, $\Delta g(x, \mathcal{T}) = -c_\delta$.

We also show that for sufficiently large values of x, we have $\Delta g(x, \mathcal{T}) = c_\lambda$. If would be convenient to extend the definition of function $g(x, \mathcal{T})$ for negative values of x [1]; then for sufficiently small values of x, we have $\Delta g(x, \mathcal{T}) = -c_\delta$.

Rather than proving the above three propositions separately, we prove them all together by induction on the size of the tree.

Proof: The proof is done by induction on the size of the tree. As the base step, let us consider a tree $\mathcal{T}$ which has one leaf with multiplicity p. In this case it is easy to verify that

$$g(x, \mathcal{T}) = \begin{cases} 0 & \text{if } x = p \\ (p - x).c_\delta & \text{if } x < p \\ (x - p).c_\lambda & \text{if } x > p \end{cases}$$

All the three propositions are true for this function. Now suppose that the three propositions are true for any tree with strictly less than k leaves ($k > 1$), and

[1] This is obviously only a theoretical extension because the number of genes cannot be negative.

consider a tree $\mathcal{T}$ with k leaves. Both left and right subtrees of $\mathcal{T}$ which are denoted by $\mathcal{T}_L$ and $\mathcal{T}_R$ have less than k leaves so the three propositions are true for them by induction hypothesis.

Consider an interval $\mathcal{I}$ of integers where $g(x, \mathcal{T}_L)$ is non-decreasing and $g(x, \mathcal{T}_R)$ is non-increasing. The main part of the proof is that we show:

$$\forall x \in \mathcal{I}: \qquad g(x, \mathcal{T}) = g(x, \mathcal{T}_L) + g(x, \mathcal{T}_R) \tag{3}$$

We prove this by contradiction. Suppose (3) is not true: there exists x in $\mathcal{I}$, such that $g(x, \mathcal{T}) < g(x, \mathcal{T}_L) + g(x, \mathcal{T}_R)$. By (1), without loss of generality we suppose that $g(x, \mathcal{T}) = g(x+1, \mathcal{T}) + c_\delta$. Remember that the number of consecutive duplications in a node in optimal generation is finite; there exists u such that $g(x, \mathcal{T}) = g(x+u, \mathcal{T}) + uc_\delta$ and $g(x+u, \mathcal{T}) = g(x+u, \mathcal{T}_L) + g(x+u, \mathcal{T}_R)$. So we have:

$$\begin{aligned}
& g(x+u, \mathcal{T}) + uc_\delta < g(x, \mathcal{T}_L) + g(x, \mathcal{T}_R) \\
& \Rightarrow g(x+u, \mathcal{T}_L) + g(x+u, \mathcal{T}_R) + uc_\delta < g(x, \mathcal{T}_L) + g(x, \mathcal{T}_R) \\
& \Rightarrow uc_\delta < \underbrace{g(x, \mathcal{T}_L) - g(x+u, \mathcal{T}_L)}_{\leq 0} + \underbrace{g(x, \mathcal{T}_R) - g(x+u, \mathcal{T}_R)}_{\leq u.c_\delta} \\
& \Rightarrow uc_\delta < uc_\delta
\end{aligned}$$

In the case that $g(x, \mathcal{T}) = g(x-1, \mathcal{T}) + c_\lambda < g(x, \mathcal{T}_L) + g(x, \mathcal{T}_R)$, similar contradiction can be obtained and this completes the proof of (3). Symmetrically if $g(x, \mathcal{T}_L)$ is decreasing and $g(x, \mathcal{T}_R)$ is increasing in an interval equality (3) is correct. As a consequent of this equality, $g(x, \mathcal{T})$ is convex in the interval $\mathcal{I}$ (because the sum of two convex functions is convex). Note that if the optimal generating interval for $\mathcal{T}_L$ and $\mathcal{T}_R$ are $[l_1, l_2]$ and $[r_1, r_2]$ respectively, then in the interval $\mathcal{I} = [\min\{l_1, r_1\}, \max\{l_2, r_2\}]$, the equality (3) is correct (see Fig. 3).

Now let us consider the interval $\mathcal{I}^+ = [\max\{l_2, r_2\} + 1, +\infty)$. In this interval both $g(x, \mathcal{T}_L)$ and $g(x, \mathcal{T}_R)$ are strictly increasing by the induction hypothesis. It is easy to verify that the function $g(x, \mathcal{T}) < g(x+1, \mathcal{T}) + c_\delta$ and so $g(x, \mathcal{T})$ is equal to the minimum of $g(x-1, \mathcal{T}) + c_\lambda$ and $g(x, \mathcal{T}_L) + g(x, \mathcal{T}_R)$. If for all values of x in this interval we have $g(x, \mathcal{T}) = g(x, \mathcal{T}_L) + g(x, \mathcal{T}_R)$ then $g(x, \mathcal{T})$ becomes strictly increasing and convex as the sum of two strictly increasing and convex functions. Otherwise, consider the first value of x_0 in the interval $\mathcal{I}^+$ such that $g(x_0, \mathcal{T}) = g(x_0 - 1, \mathcal{T}) + c_\lambda < g(x_0, \mathcal{T}_L) + g(x_0, \mathcal{T}_R)$. By the way we defined x_0, we have $g(x_0 - 1, \mathcal{T}) = g(x_0 - 1, \mathcal{T}_L) + g(x_0 - 1, \mathcal{T}_R)$. Consequently, we have $\Delta g(x_0 - 1, \mathcal{T}_L) + \Delta g(x_0 - 1, \mathcal{T}_R) > c_\lambda$. On the other hand $g(x, \mathcal{T}_L)$ and $g(x, \mathcal{T}_R)$ are convex and so $\Delta g(x, \mathcal{T}_L)$ and $\Delta g(x, \mathcal{T}_L)$ are increasing. So,

$$\forall x \geq x_0: \qquad g(x, \mathcal{T}) = g(x-1, \mathcal{T}) + c_\lambda \tag{4}$$

The function $g(x, \mathcal{T})$ is strictly increasing and convex for any $x \geq x_0$, so $g(x, \mathcal{T})$ is convex and strictly increasing for any x in $\mathcal{I}^+$. Similarly, we can show in an interval $\mathcal{I}^-$ where both $g(x, \mathcal{T}_L)$ and $g(x, \mathcal{T}_R)$ are strictly decreasing, $g(x, \mathcal{T})$ is strictly decreasing and convex.

In order to complete the proof we need to consider the different possible configurations of $g(x, \mathcal{T}_L)$ and $g(x, \mathcal{T}_R)$. Figure 3 shows the three possible arrangements of the optimal intervals of the two functions.

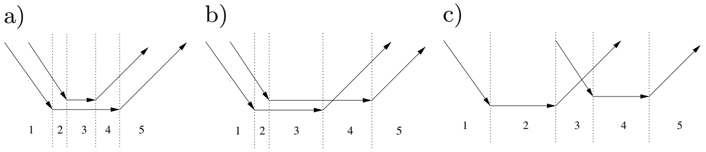


Fig. 3. Different possible configurations of $g(x, \mathcal{T}_L)$ and $g(x, \mathcal{T}_R)$

In all three cases intervals 1 and 5 refer to the $\mathcal{I}^-$ and $\mathcal{I}^+$ in the proof. Interval 2 and interval (3,4) refer to the interval $\mathcal{I}$ in the proof. The proof of the convexity in the exchange points of these intervals is easy and is omitted. It is also easy to show (by Lemma 1) that in cases (a) and (b) the optimal interval of $\mathcal{T}$ is inteval 3. In case (c) the optimal interval is an interval which is included in interval 3. □

4 Algorithm

In this section we present an algorithm for computation of the optimal D/L score histories for a given tree with multiplicities for the leaves. Algorithm 1, fills the table $g[x, T]$ (corresponding to function $g(x, \mathcal{T})$) for any $1 \leq x \leq m$ and for all subtrees of $\mathcal{T}$. $g(x, \mathcal{T})$ is not computed for non positive values of x because it is not biologically meaningful. On the other hand an optimal solution has never more than m genes present in a species so there is no need to compute $g(x, \mathcal{T})$ for $x > m$.

Algorithm 1. *GenCost(tree T)*

```
1. if $T$ is a leaf then
        1.1 for $i \leftarrow 1$ to $m$ do
                1.1.1 if $i \geq label(T)$ then $g[i, T] \leftarrow (i - label(T)) \times c_\lambda$
                1.1.2 if $i < label(T)$ then $g[i, T] \leftarrow (label(T) - i) \times c_\delta$
        1.2 exit
2. GenCost($T_L$); GenCost($T_R$);
3. $[l_1, l_2] \leftarrow OPT(T_L)$; $[r_1, r_2] \leftarrow OPT(T_R)$
4. $t_1 \leftarrow \min\{l_1, r_1\}$; $t_2 \leftarrow \max\{l_2, r_2\}$
5. for $i \leftarrow t_1$ to $t_2$ do $g[i, T] \leftarrow g[i, T_L] + g[i, T_R]$
6. for $i \leftarrow t_2 + 1$ to $m$ do $g[i, T] \leftarrow \min\{g[i-1, T] + c_\lambda, g[i, T_L] + g[i, T_R]\}$
7. for $i \leftarrow t_1 - 1$ downto $1$ do $g[i, T] \leftarrow \min\{g[i+1, T] + c_\delta, g[i, T_L] + g[i, T_R]\}$
```

Fig. 4. Algorithm 1 fills table $g[x, T]$ from leaves to the root

The correctness of the algorithm is a result of the proof of the propositions given in Section 3. The complexity of the algorithm is $O(mn)$ where n is the number of species and m is maximum number of gene copies in a species.

Once table g has been determined by Algorithm 1, finding one (or all) optimal D/L score history is easy. Starting with $g[1, T]$ at each step one checks how g is minimized (i.e. which of the lines in recurrences 1 is minimizing $g(x, T)$); if minimization is done by the first (or second) line, a duplication (or loss) will be reported at this node and recursively the computation of the optimal answer for $g(x+1, T)$ (or $g(x-1, T)$) is continued. In the case $g(x, \mathcal{T})$ is minimized by the recurrence 1.3, without giving any more events for the node $\mathcal{T}$, one adds to the output an optimal history of $g(x, \mathcal{T}_L)$ and $g(x, \mathcal{T}_R)$.

The total complexity of our algorithm will be $O(mn)$ for computing one optimal answer and $O(mn + nk)$ for computing k optimal answers if the number of optimal answers is not smaller than k. Note that multiple optimal histories correspond to the nodes and values of x such that g is minimized by two lines of recurrences 1.

In practice depending on the values of c_δ and c_λ this complexity can be improved. Let c_δ and c_λ be two positive integers such that $gcd(c_\lambda, c_\delta) = 1$. The function $g(x, \mathcal{T})$ is convex and the $\Delta g(x, \mathcal{T})$ is an increasing function; on the other hand $-c_\delta \leq \Delta g(x, \mathcal{T}) \leq c_\lambda$. This suggests to store and update the function g just by computing the points that (Δg) changes its value from x to $x+1$. This will reduce the complexity of the algorithm to $O(cn)$ where $c = c_\delta + c_\lambda$. The total complexity of the algorithm for generating k optimal histories is $O(n(\min\{m, c\}+k))$. A special case when $c_\delta = c_\lambda$ is commented below:

Unit Loss/Duplication Costs. When $c_\delta = c_\lambda = 1$, function $g(x, \mathcal{T})$ becomes very simple. If $OPT(\mathcal{T}) = [k_1, k_2]$, $g(x, \mathcal{T})$ is constant in $[k_1, k_2]$, increasing with step 1 for $x \geq k_2$ and decreasing with step -1 for $x \leq k_1$. As stated above the complexity of the algorithm will become $O(nk)$ for finding k optimal trees. The optimal intervals can be computed easily in this case. Let us define the $\otimes$ operation on integer intervals as follows:

Definition 3. *For any two integer intervals $[a_1, a_2]$ and $[b_1, b_2]$ where $a_1 \leq b_1$, the $\otimes$ intersection of these integer intervals is denoted by $[a_1, a_2] \otimes [b_1, b_2]$ and is defined as follows:*

1) if $b_2 \leq a_2$ then $[a_1, a_2] \otimes [b_1, b_2] := [b1, b2]$
2) if $b_1 \leq a_2 < b_2$ then $[a_1, a_2] \otimes [b_1, b_2] := [b1, a2]$
3) if $a_2 \leq b_1$ then $[a_1, a_2] \otimes [b_1, b_2] := [a2, b1]$

The following proposition which is a result of Propositions 1 to 3 can be used as a basis of an algorithm for the unit costs macro-evolutionary problem.

Proposition 4. *Let $\mathcal{T}$ be a binary tree , with more than one leaf, let $\mathcal{T}_L$ and $\mathcal{T}_R$ denote the left and the right subtrees of the root of the tree $\mathcal{T}$; $OPT(\mathcal{T})$ can be computed as follows:*

$$OPT(\mathcal{T}) := OPT(\mathcal{T}_L) \otimes OPT(\mathcal{T}_R).$$

5 Conclusion

In this paper, we have studied some combinatorial properties of the optimal D/L histories for a given species tree and the number of gene copies found in each species. Based on these properties we proposed an improved algorithm for finding the optimal histories in $O(m)$ order faster than the previous algorithm. We also showed that the improvement of the algorithm is $O(m^2)$ for the case of unit cost duplication/loss function. For a gene family like kinases with hundreds of duplicates, this improvement is important.

The macro-evolutionary phylogeny problem has been shown to be useful and interesting in order to build phylogenies based on both macro and micro-evolutionary processes (see [5]). The current work, improves the running time of such unified models which is essential for making large phylogenies.

Acknowledgement

We wish to thank Dannie Durand, Bjarni V. Halldórsson and Benjamin Vernot for their useful discussions and comments on the problem.

References

1. L. Arvestad, A. C. Berglund, J. Lagergren and B. Sennblad: Gene tree reconstruction and orthology analysis based on an integrated model for duplication and sequence evolution. In *Proc. RECOMB 2004*, pp 326-335, ACM Press, 2004.
2. L. Arvestad, A. C. Berglund, J. Lagergren and B. Sennblad: Bayesian gene/species tree reconciliation and orthology analysis using MCMC. *Bioinformatics*, 19 Suppl. 1, pp 7-15, 2003.
3. B. Chor and T. Tuller: Maximum likelihood of evolutionary trees is hard. in *Proc. RECOMB 2005*, LNCS 3500, pp 296-310, Springer 2005.
4. W. H. Day: Computational complexity of inferring phylogenies from dissimilarity matrices. *Bull. Math. Biol.*, 49(4):461-7, 1987.
5. D. Durand, B. V. Halldórsson and B. Vernot: A Hybrid Micro-Macroevolutionary Approach to Gene Tree Reconstruction. in *Proc. RECOMB 2005*, LNCS 3500, pp 250-264, Springer 2005.
6. O. Eulenstein, B. Mikrin, and M. Vingron: Duplication-based measures of difference between gene and species trees. *Journal of Computational Biology.* 5:135-148, 1998.
7. J. Felsenstein: Phylogenies from molecular sequences: Inference and reliability, *Annu. Rev. Genet.* 22 pp 521-565, 1988.
8. W. Fitch and E. Margoliash: Construction of phylogenetic trees. *Science.* 155 pp 279-284, 1967.
9. M. Goodman, J. Czelusniak, G.W. Moore, A.E. Romero-Herrera and G Matsuda: Fitting the gene lineage into its species lineage, a parsimony strategy illustrated by cladograms constructed from globin sequences. *Syst Zool*, 28, pp 138-163, 1979.
10. R. Guigo, I. Muchnik, and T.F. Smith: Reconstruction of ancient phylogenies. *Molecular Phylogenetics and Evolution*, 6, pp 189-213, 1996.
11. M. T. Hallett and J. Lagergren: New algorithms for the duplication-loss model. in *Proc. RECOMB 2000.*

12. B. Ma, M. Li, and L. Zhang: From gene trees to species trees. *SIAM J. on comput.*, 2000.
13. B. Mirkin, I Muchnik, T. F. Smith: A biologically consistent model for comparing molecular phylogenies. *Journal of Computational Biology*, 2, pp 493-507, 1995.
14. R. D. M. Page: Maps between trees and cladistic analysis of historical associations among genes, organisms and areas. *Syst Zool.*, 43, pp 58-77, 1994.
15. M. Nei: Molecular Evolution Genetics, Columbia University Press, New York, 1987.
16. C.M. Zmasek and S. R. Eddy: A simple algorithm to infer gene duplication and speciation events on a gene tree. *Bioinformatics*, 17(9), pp 821-828, Sep. 2001.

Property Matching and Weighted Matching

Amihood Amir[1], Eran Chencinski[2], Costas Iliopoulos[3],
Tsvi Kopelowitz[2], and Hui Zhang[3]

[1] Department of Computer Science, Bar-Ilan University, Ramat-Gan 52900, Israel
and College of Computing, Georgia Tech, Atlanta, GA 30332-0280
+972 3 531-8770
amir@cs.biu.ac.il

[2] Dept. of Computer Science, Bar-Ilan U., 52900 Ramat-Gan, Israel
(972-3)531-8408
{chenche, kopelot}@cs.biu.ac.il

[3] Department of Computer Science, King's College London, Strand,
London WC2R 2LS, United Kingdom
{csi, hui}@dcs.kcl.ac.uk

Abstract. Pattern Matching with Properties (Property Matching, for short), involves a string matching between the pattern and the text, and the requirement that the text part satisfies some property.

It is straightforward to do sequential matching in a text with properties. However, indexing in a text with properties becomes difficult if we desire the time to be output dependent. We present an algorithm for indexing a text with properties in $O(n \log |\Sigma| + n \log \log n)$ time for preprocessing and $O(|P| \log |\Sigma| + tocc_\pi)$ per query, where n is the length of the text, P is the sought pattern, Σ is the alphabet, and $tocc_\pi$ is the number of occurrences of the pattern that satisfy some property π.

As a practical use of Property Matching we show how to solve Weighted Matching problems using techniques from Property Matching. Weighted sequences have been introduced as a tool to handle a set of sequences that are not identical but have many local similarities. The weighted sequence is a "statistical image" of this set, where we are given the probability of every symbol's occurrence at every text location. Weighted matching problems are pattern matching problems where the given text is weighted.

We present a reduction from Weighted Matching to Property Matching that allows off-the-shelf solutions to numerous weighted matching problems including indexing, swapped matching, parameterized matching, approximate matching, and many more. Assuming that one seeks the occurrence of pattern P with probability ϵ in weighted text T of length n, we reduce the problem to a property matching problem of pattern P in text T of length $O(n(\frac{1}{\epsilon})^2 \log \frac{1}{\epsilon})$.

1 Introduction

One of the technical problems that pattern matching has had to deal with is that of matching a pattern in a text with properties. The idea is that the pattern matching itself is insufficient, but the particular text substring that is matched also needs to satisfy a desired property. Some examples come from molecular

M. Lewenstein and G. Valiente (Eds.): CPM 2006, LNCS 4009, pp. 188–199, 2006.

biology, where it has long been a practice to consider special genome areas by their structure.

It is straightforward (as we show later) to solve sequential pattern matching with properties since the intersection of the properties and matching can be done in linear time. However, the problem becomes more complex when it is required to *index* a text with properties. The classical pattern matching problem is that of finding all occurrences of *pattern* $P = p_1 p_2 \cdots p_m$ in *text* $T = t_1 t_2 \cdots t_n$, where T and P are strings over alphabet Σ. In the *indexing problem* we are given a large text that we want to preprocess in a manner that allows fast solution of the following queries: "Given a (relatively short) pattern P find all occurrences of P in T in time proportional to $|P|$ and the number of occurrences".

The indexing problem and its many variants have been central in pattern matching and information retrieval. However, when it comes to indexing a text with properties, intersecting the pattern with the properties may give a worst case that is not output-dependent.

In this paper we give a precise definition of pattern matching with properties and provide a data structure that preprocesses the text in $O(n \log |\Sigma| + n \log \log n)$ time and supports queries in $O(|P| \log |\Sigma| + tocc_\pi)$ time per query, where n is the text length, P is the sought pattern, $|\Sigma|$ is the alphabet, and $tocc_\pi$ is the number of occurrences of P that satisfy some property π. These are almost the same bounds that exist in the literature for ordinary indexing.

We now turn to an apparently unrelated problem. Among the challenges that the pattern matching field is currently grappling with are those of *motif discovery*, and *local alignment*. Recently, the concept of *weighted sequences* was introduced as a suggested method of satisfying the above needs. A *weighted sequence* is essentially what is also called in the biology literature *Position Weight Matrix* (PWM for short) [5]. The *weighted sequence* of length m is a $|\Sigma| \times m$ matrix that reports the frequency of each symbol in finite alphabet Σ (nucleotide, in the genomic setting) for every possible location.

Iliopoulos et al. [4] considered building very large Position Weight Matrices that correspond, for example, to complete chromosome sequences that have been obtained using a whole-genome shotgun strategy. By keeping all the information the whole-genome shotgun produces, it should be possible to identify information that was previously undetected after being faded during the consensus step. This concept is true for other applications where local similarities are thus encoded. It is therefore necessary to develop adequate algorithms on weighted sequences, that can be an aid to the application researchers for solving various problems they are liable to encounter.

It turns out that handling weighted sequences is algorithmically challenging [4] even for simple tasks such as exact matching. It is certainly desirable to be able to answer more ambitious questions, such as *scaled weighted matching*, *swapped weighted matching*, *parameterized weighted matching* as well as to *index* a weighted sequence.

We develop a general framework that allows solving all the problems mentioned above. In particular this presents the first known algorithms for problems such as scaled matching, swapped matching and parameterized matching in weighted sequences. Since most current methods for handling weighted matching use techniques that are not conductive to indexing (e.g., convolutions), it is surprising that our framework also enables indexing weighted sequences with the same query time as in the non-weighted case.

These results are all enabled by a reduction of weighted matching to property matching. This reduction creates an ordinary text of length $O(n(\frac{1}{\epsilon})^2 \log \frac{1}{\epsilon})$ for the weighted matching problem of length n text and desired probability ϵ. Since the outcome of the reduction is an ordinary text with a property, then **all** pattern matching problems that can be solved in ordinary text and pattern can have their weighted versions solved with the time degradation of the reduction.

The indexing problem for weighted text becomes a problem of indexing an ordinary (longer) text with properties. We can now use the indexing text with properties result to solve weighted indexing as well.

2 Property Matching – Definitions

For a string $T = t_1 \cdots t_n$, we denote by $T_{i \cdots j}$ the substring $t_i \cdots t_j$. The suffix $T_{i \cdots n}$ is denoted by T^i, and the suffix tree of T is denoted by $ST(T)$. The leaf corresponding to T^i in $ST(T)$ is denoted by $leaf(T^i)$. The label of an edge e in $ST(T)$ is denoted by $label(e)$.

For a node u in the suffix tree of a string T, we denote by ST_u the subtree of the suffix tree rooted by u. The label of u is the concatenation of the labels of the edges on the path from the root of the suffix tree to u, in the order they are encountered, and is denoted by $label(u)$.

We are now ready to define a property for a string.

Definition 1. *A property π of a string $T = t_1 \cdots t_n$ is a set of intervals $\pi = \{(s_1, f_1), \ldots, (s_t, f_t)\}$ where for each $1 \leq i \leq t$ it holds that: (1) $s_i, f_i \in \{1, \ldots, n\}$, and (2) $s_i \leq f_i$. The size of property π, denoted by $|\pi|$, is the number of intervals in the property (or in other words - t).*

We assume that the properties are given in standard form as defined below.

Definition 2. *A property π for a string of length n is said to be in standard form if: (1) it is in explicit form, (2) for any $1 \leq i \leq n$, there is at most one $(s_k, f_k) \in \pi$ such that $s_k = i$, and (3) $s_1 < s_2 < \cdots < s_{|\pi|}$.*

3 General Pattern Matching with Properties

This section defines the notion of general pattern matching with properties. following definition.

Definition 3. *Given a text $T = t_1 \cdots t_n$ with property π, pattern $P = p_1 \cdots p_m$, and a definition of a matching α, we say that P α-matches $T_{i \cdots j}$ under property π if P α-matches $T_{i \cdots j}$, and there exists $(s_k, f_k) \in \pi$ such that $s_k \leq i$ and $j \leq f_k$.*

The following definition will assist us in solving property matching problems.

Definition 4. *For a property π of a string $T = t_1 \cdots t_n$, the end location of* $1 \leq i \leq n$*, denoted by end*(i)*, is defined to be the maximal* f_k *such that* $(s_k, f_k) \in \pi$ *and* $s_k \leq i \leq f_k$*. If no such* f_k *exists, we say that end*$(i) = NIL$*.*

Note that $end(i)$ can easily be calculated for all locations i in T in time $O(n)$ (recall that π is given in standard form). Now, given a text $T = t_1 \cdots t_n$ and a pattern $P = p_1 \cdots p_m$, if there exists an algorithm for an α-matching problem that runs in time $O(g_\alpha(n, m))$, then given a text T with property π, and pattern P, we can find all $T_{i \cdots j}$ that α-match P in time $O(g_\alpha(n, m) + n) = O(g_\alpha(n, m))$.

However, the above reduction does not suffice for the property indexing problem (defined below). Before explaining why, we first provide a formal definition of the property indexing problem.

Definition 5. ***Property Indexing Problem (PIP)*** *Given a text string* $T = t_1 \cdots t_n$ *with property* π*, preprocess* T *such that on-line queries of the form "find all locations where a pattern string* P *occurs in* T *under* π*" can be answered in time proportional to the size of the pattern (rather than the text) and the output.*

The problem with the PIP is that known indexing data-structures do not suffice. For example, given a suffix tree for T, we can find all of the occurrences of P in T in time $O(P \log |\Sigma| + tocc)$ where $tocc$ is the number of the occurrences. However, $tocc$ is not the number of occurrences of P in T under π; it includes also the occurrences of P in T that are not occurrences under π. We could solve this problem by also preprocessing $end(i)$ for all locations i in T as we did before. However, this would require scanning all of the occurrences of P in T (taking $O(tocc)$ time), and we would like to answer indexing queries in time dependent on $tocc_\pi$, where $tocc_\pi$ is the number of occurrences of P in T under π, which might be much smaller than $tocc$. Also, keep in mind that we want a solution that takes minimal preprocessing time, and requires only linear space. This is the problem addressed by our new data-structure.

In the next sections we will define our data-structure, show how it is constructed in time $O(n \log |\Sigma| + n \log \log n)$, and finally, show how an indexing query can be answered in time $O(m \log |\Sigma| + tocc_\pi)$.

4 The Property Suffix Tree

We now define the data-structure used for solving the PIP. The data-structure we present is based on the suffix tree - thus, we name it the Property Suffix Tree, or PST for short. The construction is for a text $T = t_1 \cdots t_n$ with property π. The idea is based on a lemma that we provide following the next definition.

Definition 6. *For a string* T *with property* π *and a node* u *in the suffix tree of* T*, we denote by* S_u^π *the maximal set of locations* $\{i_1, \cdots, i_\ell\} \subseteq \{1, \cdots, n\}$ *such that for every* $i_j \in S_u^\pi$ *we have that: (1) leaf*(T^{i_j}) *is in* ST_u*, and (2) if end*$(i_j) \neq NIL$ *then end*$(i_j) - i_j > |label(u)|$*.*

Lemma 1. *Let T be a string with property π, and let u and v be two nodes in the suffix tree of T such that v is u's parent, then $S_u^\pi \subseteq S_v^\pi$.*

Proof. The proof follows from definition 6. For any location $i_j \in S_u^\pi$ we know $leaf(T^{i_j})$ is in ST_u, thus it is also in ST_v. We also know that $end(i_j) - i_j > |label(u)|$. Being that $|label(u)| > |label(v)|$, we have that $end(i_j) - i_j > |label(u)| > |label(v)|$. Due to the maximality of S_v^π, it must be that $i_j \in S_v^\pi$. □

Corollary 1. *For a string T with property π, the path from the root of $ST(T)$ to $leaf(T^i)$ can be split into the following two paths: (1) the path consisting of all nodes u such that $i \in S_u^\pi$, and (2) the path consisting of all nodes u such that $i \notin S_u^\pi$.*

Definition 7. *Consider the two paths from Corollary 1, and the i^{th} suffix of T. Let v be the deepest node on the first path. The location of i in the PST of T is defined as follows. If $end(i) - i = |label(v)| - 1$ then $loc(i) = v$. Otherwise, $loc(i)$ is the edge connecting the two paths.*

The idea behind the PST is to move each suffix T^i in $ST(T)$ up to $loc(i)$. We will later show why this solves the PIP. We now define the PST using an overview construction. First, we construct $ST(T)$ using, for example, [6]. Then, for every suffix T^i find $loc(i)$, and maintain a list of locations for each edge e consisting of all i such that $e = loc(i)$ and for each node u consisting of all i such that $u = loc(i)$. We denote these lists by $suf(e)$ and $suf(u)$ respectively. Next, we mark each node u in $ST(T)$ such that either $suf(u)$ is not an empty list, or u is connected to some edge e where $suf(e)$ is not an empty list, or u is an ancestor of a marked node. Now, we delete all of the nodes that are not marked, and compress non branching paths in the remaining tree to one edge (like we do in suffix trees). Of course, during the compression of a path into an edge, we must concatenate all of the $suf(u)$ and $suf(e)$ for all nodes u and edges e on the path, except for the last node. The concatenation of all of those lists forms the list of locations $loc(e')$ for the new edge e' that will replace the non-branching path. Finally, we will be interested in ordering $suf(e)$ for the remaining edges in order to allow efficient querying. This will be explained later.

Note that except for the stage in which we construct $suf(e)$ and $suf(u)$ for the edges e and nodes u in $ST(T)$ and the ordering of the lists of locations, the rest of the algorithm can be easily implemented to take $O(n \log |\Sigma|)$ by building a suffix tree and using a constant number of depth-first searches (DFS). Also note that the size of the data structure is clearly linear in the size of T. Thus, it remains to show how to construct $suf(e)$ and $suf(u)$ for the edges e and nodes u in $ST(T)$, and how to order them while allowing us to answer queries efficiently. This is explained in the next two subsections.

4.1 Constructing Lists of Locations

We now show how to construct $suf(e)$ and $suf(u)$ for every edge e and every node u in $ST(T)$. In the following subsection we show how to order $suf(e)$ in a way that will allow efficient querying.

In order to find $loc(i)$ for every suffix T^i, we use the weighted ancestor queries that were presented in [3], and improved upon in [1]. The weighted ancestor problem is defined as follows:

Definition 8. *Let T be a rooted tree where each node u has an associated value $value(u)$ from an ordered universe U such that if v is the parent of u then $value(v) < value(u)$. The weighted ancestor problem is given a query of the form $WA(u, i)$ where u is a node in T and $i \in U$, return the node v that is the lowest ancestor of u such that $value(v) < i$.*

Clearly, if we set the value of a node u to be $|label(u)|$, then given a leaf $leaf(T^i)$, the answer to the query $WA(leaf(T^i), end(i) - i)$ will either give us a node that is $loc(i)$, or a node that is connected to the edge that is $loc(i)$. In the later case, we can easily find $loc(i)$ in $O(\log |\Sigma|)$ time. In [1] the weighted ancestor problem was solved for suffix trees taking $O(n)$ preprocessing time, and $O(\log\log n)$ query time. Thus, we can find $loc(i)$ for all T^i's in $O(n(\log\log n + \log |\Sigma|))$ time. However, the suffixes on the edges are not ordered in a way that would allow efficient indexing queries. We cannot simply order the suffixes by descending $loc(i) - i$ because this would require sorting, and would take too much time (we would need to sort the locations on every edge in the tree according to the appropriate values). To solve this problem, we show in Subsection 4.2 how to preprocess a set of n' elements in $O(n')$ time such that given a value whose rank[1] in the set is k, we can find all of the elements less than or equal to that value in $O(k)$ time. In Subsection 4.3 we will show how this helps us answer indexing queries efficiently. Thus, we will run this algorithm on every edge in the tree, taking a total of linear time. Finally, the time required for constructing the PST is $O(n \log |\Sigma| + n \log\log n)$. Note that for constant size alphabets we are dominated by the $n \log\log n$ factor.

4.2 Ordering the Suffixes on an Edge

As we previously mentioned, we require a scheme such that given a set of n' elements we can preprocess those elements in $O(n')$ time such that given a value whose rank in the set is k, we can find all of the elements less than or equal to that value in $O(k)$ time. To solve this algorithm we use the fact that finding the median of a set of numbers can be done in linear time (e.g., by [2]). The preprocessing is as follows. First find the median of the set, and separate the set to the set of values smaller than the median, and the set of the values that greater than the median (for simplicity, we assume all values are distinct). For the set of items with value greater than the median, we put them in an array of size n', in the second part of the array. We recursively do the same for the elements less than the median, each time putting the items greater than the median in the left most part of the unfilled array, until we reach a set of size one, and we put the remaining element in the first location in the array. Note that the time required is $O(\sum_{i=0}^{\log n'} \frac{n'}{2^i}) = O(n')$.

[1] The rank of a value in a set is the number of elements in the set less than or equal to the value.

Now, given a query value t with rank k, we proceed as follows. We begin by comparing t with the first location. If t is smaller, than we output an empty set. If t is larger, we output the first element as part of the output and continue on to scan the next two elements in the array. If they are both less than or equal to t, we output them both, and continue on to the next four elements. We continue on such that at the i^{th} iteration, if all of the 2^{i-1} elements are less than or equal to t, we output them all, and continue to the next 2^i items. This continues until we reach some item whose value is less than t. Say this happens at iteration number i'. In such a case, we continue to scan all of the $2^{i'-1}$ items of the iteration, outputting only those items with value less than or equal to t, and then we are done.

Clearly, we output all elements that are less than or equal to t, as once we find an element that is greater than t in the i' iteration, we know that all the rest of the elements in the array (located after the $2^{i'-1}$ elements of the current iteration) have value greater than t (this follows directly from the way we arranged the array, dividing it around the median). Moreover, the running time is $O(k)$ as if we stop at iteration i', this means we output at least $\sum_{i=1}^{i'-1} 2^{i-1} = \Omega(2^{i'})$, and the running time is at most $\sum_{i=1}^{i'} 2^{i-1} = O(2^{i'})$. Finally, note that the same type of technique can be used if we are interested in finding all the elements that have value larger or equal to t. We will actually be interested in this version of the problem for ordering the suffixes on the edges.

4.3 Answering Indexing Queries

In this section we describe how to answer indexing queries in $O(m \log |\Sigma| + tocc_\pi)$. But first, for a node u in the PST we denote by PST_u the subtree of the PST rooted by u. The indexing query is answered as follows. We first begin by searching the PST like we search a suffix tree, until we reach a node or an edge. If we reach a node u, we run a DFS on PST_u, outputting $suf(w)$ and $suf(e')$ for every node w and every edge e' in PST_u. If when searching we reach an edge $e = (u, v)$ where we match the first ℓ characters of $label(e)$, then we first output $suf(w)$ and $suf(e')$ for every node w and every edge e' in PST_v using a DFS, and we also output every location i in $suf(e)$ such that $end(i) - i > |label(u)| + \ell$. In order to accomplish the second part, we use the scheme from Subsection 4.2. it remains to show that the additional amount of time spent (i.e. except for the search part that takes $O(m \log |\Sigma|)$) is linear in the size of the output. This follows from the following lemma.

Lemma 2. *Let $PST(T)$ be the PST of a string T under property π. Then in the subtree of any node in $PST(T)$, the size of the subtree is linear in the number of locations in the union of $suf(w)$ and $suf(e')$ for every node w and every edge e' in the subtree.*

Theorem 1. *The PIP can be solved in $O(n \log |\Sigma| + nlog \log n)$ preprocessing time, using linear space, where the query time is $O(m \log |\Sigma| + tocc_\pi)$.*

In the following sections we consider weighted matching problems and show a general framework for solving weighted matching problems using property matching.

5 Weighted Matching – Definitions

Definition 9. *A weighted sequence $T = t_1 \cdots t_n$ over alphabet Σ is a sequence of sets t_i, $i = 1, \cdots, n$. Every t_i is a set of pairs $(s_j,\ \pi_i(s_j))$, where $s_j \in \Sigma$ and $\pi_i(s_j)$ is the probability of having symbol s_j at location i. Formally,*

$$t_i = \left\{ (s_j,\ \pi_i(s_j)) \mid s_j \neq s_l \ for\ j \neq l,\ and \sum_j \pi_i(s_j) = 1 \right\}.$$

Definition 10. *Given a pattern $P = p_1 \cdots p_m$ over alphabet Σ, we say that the solid pattern P (or simply pattern P) occurs at location i of a weighted text T with probability of at least ϵ if $\prod_{j=1}^{m} \pi_{i+j-1}(p_j) \geq \epsilon$, where ϵ is a given parameter which we call the threshold probability.*

Notice that all characters having probability of appearance less than ϵ are not of interest to us, since any pattern using such a character will also have probability of appearance less than ϵ, which is below the threshold probability. Therefore, we are only interested in characters having probability of appearance of at least ϵ. We call such characters **heavy characters**.

Definition 11. *Given $0 < \epsilon \leq 1$, we classify each location i, $1 \leq i \leq n$, in the text into the following three categories: (1) Solid positions where there is one (and only one) character at location i with probability of appearance exactly 1, (2) Leading positions where there is at least one character at location i with probability of appearance greater than $1 - \epsilon$ (and less than 1), and (3) Branching positions where all characters at location i have probability of appearance at most $1 - \epsilon$.*

Notice that if $\epsilon \leq \frac{1}{2}$, then at every solid and leading position there is only one heavy character since only one character can have probability of appearance greater than $1 - \epsilon \geq \frac{1}{2}$, whereas in a branching position there maybe several heavy characters. However, if $\epsilon > \frac{1}{2}$ there are no heavy characters in a branching position since all characters have probability of appearance of at most $1 - \epsilon < \epsilon$.

In the following section we define the notions of Maximal Factors and Extended Maximal Factors and show how they are used in the reduction from weighted matching to property matching.

6 Maximal Factors and Extended Maximal Factors

A weighted pattern matching problem is a pattern matching problem where the text is weighted. The idea behind our framework is to create a regular text from the weighted text in a way that we can run regular pattern matching algorithms on the regular text while ensuring that the occurrences appear with probability of at least ϵ. In order to do so, we first define the notion of maximal factor.

Definition 12. *Let $T = t_1 \cdots t_n$ be a weighted text and let $X = x_1 \cdots x_l$ be a string. We denote $\pi_i(X) = \pi_i(x_1) \times \cdots \times \pi_{i+l-1}(x_l)$. Given $0 < \epsilon \leq 1$, we say that a string, X, is a maximal factor of T starting at location i if the following conditions hold: (1) $\pi_i(X) \geq \epsilon$, (2) if $i > 1$, then $\pi_{i-1}(s_j) \times \pi_i(X) < \epsilon$ for all $s_j \in \Sigma$, and (3) if $i + l \leq n$, then $\pi_i(X) \times \pi_{i+l}(s_j) < \epsilon$ for all $s_j \in \Sigma$.*

In other words, a maximal factor starting at location i is a string that when aligned to location i has probability of appearance at least ϵ. However, if we extend the string by even one character to the right and align it to location i or if we extend the string by even one character to the left and align it to location $i-1$, then the probability appearance of the string drops below ϵ.

A straightforward approach for transforming the weighted text T to a regular text would be to simply find all the maximal factors of the text and concatenate them to a new regular text T' (of course we will need some kind of a delimiter character to separate between the factors). The advantage of this approach is that every pattern that appears in T' appears also in T with probability of at least ϵ, since a maximal factor has probability of appearance at least ϵ and so have all of its substrings. Unfortunately, this approach does not suffice. It can be shown (due to lack of space details are omitted) that the total length of all maximal factors of a weighted text $T = t_1 \cdots t_n$ could be at least $\Omega(n^2)$, which is rather large. Therefore, we define the notion of extended maximal factor, and show a better upper bound on the total length of all extended maximal factors. In order to define the extended maximal factor we use the Leading to Solid Transformation.

Definition 13. *The Leading to Solid Transformation of a weighted sequence $T = t_1 \cdots t_n$ denoted LST(T), is a weighted sequence $T' = t'_1 \cdots t'_n$ such that:*

$$t'_i = \begin{cases} t_i & \textit{if } i \textit{ is a solid or a branching position} \\ \{(\sigma, 1)\} & \textit{if } i \textit{ is a leading position and } \sigma \textit{ is a heavy character} \\ \phi & \textit{if } i \textit{ is a leading position and there are no heavy characters} \end{cases}$$

In essence, $LST(T)$ is the same as T, where all leading positions become solid. The only exception is when all characters in a leading position are not heavy, thus, we ignore that location (set to by ϕ) and treat each part of $LST(T)$ divided by ϕ separately. For the rest of this paper, we assume $LST(T)$ has no ϕ's.

Notice that this transformation is uniquely defined, since either $\epsilon \leq \frac{1}{2}$ in which case there is one (and only one) character with probability $> 1 - \epsilon$, thus, it is also the only heavy character at that location or $\epsilon > \frac{1}{2}$ in which case at every location there is at most one heavy character.

Another important observation is that the size of $LST(T)$ is linear in the size of T and can easily be built in linear time. The LST transformation leads us to the following definition.

Definition 14. *Given $0 < \epsilon \leq 1$ and a weighted text T, we say that a string X is an extended maximal factor of T starting at location i if X is a maximal factor of $LST(T)$ starting at location i.*

We now prove a few properties on maximal factors and extended maximal factors, that will help us in bounding the total length of all extended maximal factors of a weighted text.

Lemma 3. *Given $0 < \epsilon \leq 1$ and a weighted text T, there are at most $\lfloor \frac{1}{\epsilon} \rfloor$ heavy characters at a branching position.*

Definition 15. *Given $0 < \epsilon \leq 1$ and a weighted text T, we say that a maximal factor $X = x_1 \cdots x_l$ passes by location i of T, if X starts at location i' such that $i' \in [i - l + 1, i]$.*

Lemma 4. *Given $0 < \epsilon \leq 1$ and a weighted text T, a maximal factor of T passes by at most $O(\frac{1}{\epsilon} \log \frac{1}{\epsilon})$ branching positions.*

Definition 16. *Given $0 < \epsilon \leq 1$ and a weighted text T, we say that location i is a starting location of T, if either $i = 1$ or $i > 1$ and t_{i-1} is not a solid position.*

Observe that a maximal factor of T always starts at a starting location, otherwise it could be extended to the left with solid positions without decreasing the probability of appearance, which contradicts the maximality of the factor.

The following lemma bounds the number of maximal factors starting from a starting location in a weighted text T, such that T has no leading positions. The fact that T has no leading positions implies that this is true for $LST(T)$ of any weighted text T, and thus actually bounds the number of extended maximal factors starting from any location in T.

Lemma 5. *Given $0 < \epsilon \leq 1$ and a weighted text T such that T has no leading positions, there are at most $\lfloor \frac{1}{\epsilon} \rfloor$ maximal factors starting at a starting location.*

Lemma 6. *Given $0 < \epsilon \leq 1$ and a weighted text T such that T has no leading positions, the number of maximal factors passing by each location i in the text is at most $O((\frac{1}{\epsilon})^2 \log \frac{1}{\epsilon})$.*

The following theorem bounds the total length of all extended maximal factors.

Theorem 2. *Given $0 < \epsilon \leq 1$ and a weighted text T, the total length of all extended maximal factors of T is at most $O(n(\frac{1}{\epsilon})^2 \log \frac{1}{\epsilon})$.*

Proof. This follows immediately from Lemma 6. □

The following lemma shows that this analysis is tight up to a logarithmic factor.

Lemma 7. *Given $0 < \epsilon \leq 1$ and a weighted text T, the total length of all extended maximal factors of T is $\Omega(n(\frac{1}{\epsilon})^2)$.*

In the next section we show how to efficiently find all extended maximal factors of a weighted sequence.

7 Finding All Extended Maximal Factors in a Weighted Sequence

Let $T = t_1 \cdots t_n$ be a weighted sequence such that $\{s_i^1, s_i^2, \cdots, s_i^{k_i}\}$ is the set of characters appearing at location i with positive probability, and $\{\alpha_i^1, \alpha_i^2, \cdots, \alpha_i^{k_i}\}$ is the matching set of probabilities of the s_i^j's.

We present a simple brute-force algorithm that given a weighted text T and a threshold probability ϵ, outputs all extended maximal factors in T. The algorithm first calculates $T' \leftarrow LST(T)$ in linear time (as mentioned above). Then, starting from each starting location i in T', we begin by extending all possible substrings from location i that appear with probability of at least ϵ. Each time we check if some string that we have extended so far can be extended even more to the right. Once we cannot extend a string, it is outputted (of course, using delimiters between consecutive outputs of substrings).

Noting that finding $LST(T)$ from T can be done in linear time, it is easy to see that the running time of this algorithm is linear in the size of the output, i.e. linear in the total length of all extended maximal factors. By combining this result with theorem 1, the corollary follows.

Corollary 2. *Given a constant threshold $0 < \epsilon \leq 1$ and a weighted text T, the total length of all extended maximal factors of T is linear in the length of T, and can be found in linear time.*

In the following section we show how to solve weighted matching problems by reducing weighted matching problems to property matching problems.

8 Solving Weighted Matching Problems

Weighted matching problems are regular pattern matching problems where the text is weighted, and an we say that a pattern appears in the text if the probability of appearance of the pattern is above some threshold probability ϵ. We now show how to reduce this problem to the Property Matching Problem.

Given a weighted string T, we find the string of the extended maximal factors of T as was described in section 7. Denote this string by $\hat{T}$. $\hat{T}$ is a regular string, but each location has an associated probability that comes from the original location of that letter in T (the delimiters are said to have probability 0). Thus, we can define a property as the set of all intervals (s_k, f_k) where the product of the probabilities from location s_k to location f_k is at least ϵ, and the product of the probabilities from location $s_k - 1$ to location f_k and from location s_k to location $f_k + 1$ is less than ϵ. Clearly, if a pattern matches $\hat{T}$ at some location under the defined property, then the pattern weight matches T at some location. Note that this location can be found simply by saving for each location in $\hat{T}$ the original location in T that it came from (that will be the match location).

This reduction immediately gives us the following.

Corollary 3. *Weighted matching problems can be solved in the same running times as property matching except for an $O((\frac{1}{\epsilon})^2 \log \frac{1}{\epsilon})$ degradation, where ϵ is the threshold probability.*

Finally, we can also solve the indexing problem for weighted strings using the reduction above in $O(n(\frac{1}{\epsilon})^2 \log \frac{1}{\epsilon} \log |\Sigma| + n(\frac{1}{\epsilon})^2 \log \frac{1}{\epsilon}(\log\log \frac{1}{\epsilon} + \log\log n))$ preprocessing time, and $O(|P| \log |\Sigma| + tocc_\pi)$ query time, where $tocc_\pi$ is the number of occurrences of P in T with probability at least ϵ.

9 Concluding Remarks

We remark that our framework for solving weighted matching problems yields solutions to hitherto unsolved problems in weighted matching, such as scaled matching, swapped matching, parameterized matching and indexing, as well as efficient solutions to others such as exact matching and approximate matching.

Furthermore, we note that in practice, when dealing with weighted matching problems, ϵ is usually considered as a constant. Thus, solving problems such as exact matching, scaled matching, swapped matching, parameterized matching, approximate matching and many more on weighted sequences can be done, using our framework, in the same running times as the best known algorithms for the non-weighted versions, while weighted indexing can be done in $O(n(\log |\Sigma| + \log\log n))$ preprocessing time and $O(|P| \log |\Sigma| + tocc_\pi)$ query time for text of length n, where $tocc_\pi$ is the number of occurrences of pattern P in T with probability of at least ϵ.

Acknowledgement. The first author was partially supported by NSF grant CCR-01-04494 and ISF grant 35/05.

References

1. A. Amir, D. Keselman, G. Landau, M. Lewenstein, N. Lewenstein, and M. Rodeh. Indexing and dictionary matching with one error. *Journal of Algorithms*, 37:309–325, 2000. (Preliminary version appeared in WADS 99.).
2. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press and McGraw-Hill, 2nd edition, 2001.
3. M. Farach and S. Muthukrishnan. Perfect hashing for strings: Formalization and algorithms. In *Proc. 7th Combinatorial Pattern Matching Conference (CPM)*, pages 130–140, 1996.
4. C. S. Iliopoulos, L. Mouchard, K. Perdikuri, and A. Tsakalidis. Computing the repetitions in a weighted sequence. In *Proceeding of the Prague Stringology Conference*, pages 91–98, 2003.
5. J.D. Thompson, D.G. Higgins, and T.J. Gibson. Clustal w: Improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. *Nucleic Acids Research*, 22:4673–4680, 1994.
6. P. Weiner. Linear pattern matching algorithm. *Proc. 14 IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973.

Faster Two Dimensional Scaled Matching

Amihood Amir[1] and Eran Chencinski[2]

[1] Department of Computer Science, Bar-Ilan University, Ramat-Gan 52900 Israel
and College of Computing, Georgia Tech, Atlanta, GA 30332-0280
+972 3 531-8770
amir@cs.biu.ac.il

[2] Dept. of Computer Science, Bar-Ilan U., 52900 Ramat-Gan, Israel
(972-3)531-8408
chenche@cs.biu.ac.il

Abstract. The rapidly growing need for analysis of digitized images in multimedia systems has lead to a variety of interesting problems in multidimensional pattern matching. One of the problems is that of *scaled matching*, finding all appearances of a pattern, proportionally enlarged according to an *arbitrary real-sized* scale, in a given text.

The best known algorithm for this problem uses techniques from dictionary matching to solve the problem in $O(nm^3 + n^2 m \log m)$ time using $O(nm^3 + n^2)$ space, where the text is a two-dimensional $n \times n$ array and the pattern is a two-dimensional $m \times m$ array.

We present a new approach for solving the scaled matching problem improving both the running times and the space requirements. Our algorithm runs in $O(n^2 m)$ time and uses $O(n^2)$ space. This time includes the preprocessing ($O(m^3)$ time and $O(m^2)$ space), since the problem is only defined for $m \leq n$.

1 Introduction

Wide advances in technology, e.g. computer vision, multimedia libraries, and web searches in heterogeneous data, point to a glaring lack of a theory of multidimensional matching.

The last decade has seen some progress in this direction. Issues arising from the digitization process were examined by Landau and Vishkin [11]. Once the image is digitized, one wants to search it for various data. A whole body of literature examines the problem of seeking an object in an image.

In reality one seldom expects to find an exact match of the object being sought, henceforth referred to as the *pattern*. Rather, it is interesting to find all text locations that "approximately" match the pattern. Various types of differences that make up these "approximations" were considered in the literature. The earliest work dealt with *local errors* - introduced by differences in the digitization process, noise, and occlusion (the pattern partly obscured by another object). Pattern matching with *rotation* - the pattern image appearing in the text in a different angle, has been also extensively researched [9, 8, 7].

The pattern matching with *scaling* problem is concerned with seeking all occurrences of the pattern in all sizes. Amir, Landau and Vishkin showed in 1992 that

M. Lewenstein and G. Valiente (Eds.): CPM 2006, LNCS 4009, pp. 200–210, 2006.

all occurrences of a given rectangular pattern in a text can be found in *all discrete scales* in linear time. By discrete scales we mean natural numbers, i.e. the pattern scaled to sizes $1, 2, 3, \ldots$. The algorithm was linear for fixed bounded alphabets, but was not linear for unbounded alphabets. This result was improved in [5].

The early attempt to handle real scales was done for one dimensional text and pattern [2]. A satisfactory rigorous definition of scaling in an "exact matching" sense of combinatorial pattern matching was presented in [3]. The running time and space of the algorithm presented there are $O(nm^3 + n^2 m \log m)$ and $O(nm^3 + n^2)$, respectively. An improved algorithm for the one dimensional case appeared in [4].

In this paper we improve the results of the two-dimensional scaled matching with real scales problem. Our algorithm runs in $O(n^2 m)$ time and uses $O(n^2)$ space. The authors apologize for the lack of proofs for some lemmas. They could not be included for lack of space and will appear in the journal version.

2 Preliminaries and Definitions

Definition 1. *Let T be a two-dimensional $n \times n$ array over some alphabet Σ.*

1. *The* unit pixels array *for T (T^{1X}) consists of n^2 unit squares, called pixels in the real plane $\Re^2$. The corners of the pixel $T[i,j]$ are $(i-1, j-1), (i, j-1), (i-1, j)$, and (i, j). Hence the pixels of T form a regular $n \times n$ array that covers the area between $(0,0), (n,0), (0,n)$, and (n,n). Point $(0,0)$ is the* origin *of the unit pixel array. The* center *of each pixel is the geometric center point of its square location. Each pixel $T[i,j]$ is identified with the value from Σ that the original array T had in that position. We say that the pixel has a color or a character from Σ. See Figure 1 for an example of the grid and pixel centers of a 7×7 array.*
2. *Let $r \in \Re$, $r \geq 1$. The* r-ary pixels array *for T (T^{rX}) consists of n^2* r-squares*, each of dimension $r \times r$ whose* origin *is $(0,0)$ and covers the area between $(0,0), (nr, 0), (0, nr)$, and (nr, nr). The corners of the pixel $T[i,j]$ are $((i-1)r, (j-1)r), (ir, (j-1)r), ((i-1)r, jr)$, and (ir, jr). The* center *of each pixel is the geometric center point of its square location.*

Notation: Let $r \in \Re$. $\|r\|$ denotes the *rounding* of r, i.e. $\|r\| = \begin{cases} \lfloor r \rfloor & \text{if } r - \lfloor r \rfloor < .5; \\ \lceil r \rceil & \text{otherwise.} \end{cases}$

Definition 2. *Let T be an $n \times n$ text array, P be an $m \times m$ pattern array over alphabet Σ, and let $r \in \Re$, $1 \leq r \leq \frac{n}{m}$. We say that there is an* occurrence of P scaled to r *at text location (i,j) if the following conditions hold:*

Let T^{1X} be the unit pixels array of T and P^{rX} be the r-ary pixel arrays of P. Translate P^{rX} onto T^{1X} in a manner that the origin of P^{rX} coincides with location $(i-1, j-1)$ of T^{1X}. Every center of a pixel in T^{1X} which is within the area covered by $(i-1, j-1), (i-1, j-1+mr), (i-1+mr, j-1)$ and $(i-1+mr, j-1+mr)$ has the same color as the r-square of P^{rX} in which it falls.

The colors of the centers of the pixels in T^{1X} which are within the area covered by $(i-1, j-1), (i-1, j-1+mr), (i-1+mr, j-1)$ and $(i-1+mr, j-1+mr)$

define a $\|mr\| \times \|mr\|$ array over Σ. This array is denoted by $P^{s(r)}$ and called P scaled to r.

Understanding, and properly using, the shift from the continuous to the discrete and back are key to the efficiency of our algorithms. To this effect we need the following functions.

Definition 3. *Let k be a discrete length of a pattern prefix in any dimension, i.e. the number of consecutive rows starting from the pattern's beginning, or the length of a row prefix. Let $r \in \Re$ be a scale, and let N be the natural numbers. We define the function $D : N \times \Re \rightarrow N$ as follows: $D(k, r) = \|kr\|$.*

We would like to define an "inverse" function $D^{-1} : N \times N \rightarrow \Re$ with the property $D^{-1}(k, D(k, r)) = r$. However, this is not possible since D is not injective. Lemma 1, which follows from the definition, below tells us that for a fixed k there is a structure to the real numbers r that are mapped to the same element $D(k, r)$, namely, they form an interval $[r_1, r_2)$.

Lemma 1. *Let $r_1, r_2 \in \Re$, $k \in N$ such that $D(k, r_1) = D(k, r_2)$ and let $r \in \Re$, $r_1 < r < r_2$. Then $D(k, r) = D(k, r_1)$.*

Definition 4. *Let $k, \ell \in N$. Define*

$$L^{-1}(k,\ell) = \begin{cases} 1 & \text{if } k = \ell; \\ \frac{(\ell-0.5)}{k} & \text{otherwise.} \end{cases}$$

and $R^{-1}(k,\ell) = \frac{(\ell+0.5)}{k}$.

It is easy to see that $L^{-1}(k, \ell) = \min\{r \in \Re | D(k, r) = \ell\}$ and that $R^{-1}(k, \ell) = \min\{r \in \Re | D(k, r) = \ell + 1\}$.

The L^{-1} and R^{-1} functions are designed to give a range of scales whereby a pattern sub-range of length k may scale to a sub-range of scale ℓ. The following claim follows from the definition.

Lemma 2. *Let P be an $m \times m$ pattern and T an $n \times n$ text. Let $k \leq m$ and $\ell \leq n$, and let $[L^{-1}(k, \ell), R^{-1}(k, \ell))$ be the range of scales defined by L^{-1} and R^{-1}. Then the difference in number of rows (or number of columns) between $P^{s(r_1)}$ and $P^{s(r_2)}$, for any two $r_1, r_2 \in [L^{-1}(k, \ell), R^{-1}(k, \ell))$ can not exceed $m+2$.*

For a scaled pattern occurrence α of an $m \times m$ pattern with scale r_α, we denote the length of the scaled pattern by m_α, which is exactly $\|m \cdot r_\alpha\|$. In other words, the scaled pattern is of size $m_\alpha \times m_\alpha$.

Below, we present an $O(n^2 m)$ algorithm for finding all scaled occurrences of an $m \times m$ pattern in an $n \times n$ text, but first we deal with the simple case where the pattern P of size $m \times m$ consists of a single character, i.e $P[x, y] = \sigma$ for every $x = 1, \ldots, m$ and $y = 1, \ldots, m$ for some $\sigma \in \Sigma$. In this case we can simply solve the scaled matching problem in linear time in the size of the text using *subrow repetition queries* and *subcolumn repetition queries* (see section 4).

For the rest of this paper, we will ignore this simple case and assume that at least one row of the pattern has at least two characters or at least one column of the pattern has at least two characters. Furthermore, we can always assume that at least one row has at least two characters, since the case where one column has at least two characters can be reduced to the first case by inverting the pattern (and the text) along the main diagonal such that the upper-right corner is swapped with the lower-left corner.

The following theorem and corollaries prove properties of scaling that are used by our algorithm.

Theorem 1. *If* $\|r_\alpha x\| < \|r_\beta x\|$ *then* $\|r_\alpha(x+y)\| - \|r_\alpha x\| \leq \|r_\beta(x+y)\| - \|r_\beta x\| + 1$, *where* $1 \leq r_\alpha, r_\beta$ *are real numbers and* x, y *are natural numbers.*

Proof: Using the fact that $rz - 0.5 \leq \|rz\| < rz + 0.5$ and that $r_\alpha < r_\beta$ we get the following inequality: $\|r_\alpha(x+y)\| - \|r_\alpha x\| \leq r_\alpha(x+y) + 0.5 - r_\alpha x + 0.5 = r_\alpha y + 1 < r_\beta y + 1 = r_\beta(x+y) - 0.5 - r_\beta x - 0.5 + 2 \leq \|r_\beta(x+y)\| - \|r_\beta x\| + 2$, and since $\|r_\alpha(x+y)\| - \|r_\alpha x\|$ and $\|r_\beta(x+y)\| - \|r_\beta x\|$ are integers the theorem follows. □

Corollary 1. *Let* α *and* β *be two scaled occurrences of a pattern* P *of size* $m \times m$. *Now suppose we split* P *vertically to a prefix of* m' *columns and a suffix of* $m - m'$ *columns, then if the scaled prefix of* α *is smaller than the scaled prefix of* β, *the scaled suffix of* α *can be larger than the scaled suffix of* β *by at most 1.*

Proof: Follows directly from Theorem 1. □

Corollary 2. *Let* T *be a text and* P *an* $m \times m$ *pattern such that the following holds:*

1. *P appears in T at least k times with pattern occurrences $\alpha_1, \ldots, \alpha_k$.*
2. *$\alpha_1, \ldots, \alpha_k$ all begin within the same row (α_1 is the leftmost, α_k is the rightmost).*
3. *Each occurrence of $\alpha_1, \ldots, \alpha_k$ overlaps with all the other occurrences, but no occurrence fully contains another occurrence.*
4. *The occurrences are ordered monotonically, i.e. either $m_{\alpha_1} < \ldots < m_{\alpha_k}$ or $m_{\alpha_1} > \ldots > m_{\alpha_k}$*

Then $k = O(m)$.

3 Algorithm Layout

Our algorithm is based on the alphabet independent exact two dimensional matching algorithm of Amir, Benson and Farach [1]. We now present the layout of the algorithm, which consists of four stages.

Stage 1. The *scale elimination* stage. The goal of this stage is to get an estimate on the size of the scale that can appear at each location. In the end of this stage we get, for each location (i, j) in the text T, an interval $[r^l_{i,j}, r^h_{i,j})$ such that if there is an occurrence of the pattern scaled to some scale r, then $r \in [r^l_{i,j}, r^h_{i,j})$.

Stage 2. The *candidate consistency* stage. Unlike the consistency stage in [1], in this stage we check for consistency only between candidates within the same row. The comparison it is done considering only the scale $r^l_{i,j}$ at each location (i,j) in the text, using a preprocessed witness table.

Stage 3. The *candidate verification* stage. As in [1], a wave is employed to verify which of the non-conflicting sources are indeed starts of pattern appearances. This wave is done for each row separately, using the solution for the *Maximal Interval Problem* that will be defined and solved in the final version.

Stage 4. The *occurrence recognition* stage. In this stage, we find all occurrences of the pattern at all the locations in the text with **all** possible scales (not only scales $r^l_{i,j}$ as in stages 2 and 3).

Each stage is done in $O(n^2m)$ time using at most $O(n^2)$ space, where the text is of size $n \times n$ and the pattern is of size $m \times m$. In the following section, we explain in detail how each stage is implemented.

4 Algorithm Implementation

4.1 Scale Elimination Stage

Definition 5. *Let P be an $m \times m$ pattern array. We say that a row $1 < x \leq m$ is a* row border *of P if P_x differs from P_{x-1}, where P_l is the l'th row of P.*

Similarly, we say that column $1 < y \leq m$ is a column border *of P if P^y differs from P^{y-1}, where P^l is the l'th column of P.*

Observe that the scale of a pattern P appearing at location (i,j) of a text T is fully dependent on the location of the (row and column) borders of P in T with respect to location (i,j). Finding all the borders of an $m \times m$ pattern can be easily done in $O(m^2)$, however, finding all the borders for every locations in the text is more involved. In order to find the borders more efficiently, we use the *subrow/subcolumn repetition queries.*

Definition 6. *A* subrow repetition query *is defined as follows: Given an $n \times n$ matrix T,*

Input: *Location $[i,j]$ and a natural number ℓ.*
Ouput: *The number of times the substring $T[i,j], T[i,j+1], ..., T[i,j+\ell-1]$ repeats itself starting at column j in rows $i, i+1, \ldots$ of T.*

The subcolumn repetition query is defined similarly on the columns. In [6] a method was presented that preprocesses an $n \times n$ text matrix in time $O(n^2)$ and subsequently allows answering every subrow (subcolumn) repetition query in $O(1)$ time. We also use inverse subrow repetition queries, similarly defined and solved.

Let P^y be some column in P, $y > 1$, we say that a row x *supports* y if $P[x, y] \neq P[x, y - 1]$. Row x is said to be a *pivot row* of P if x is the highest row that supports column y', where $P^{y'}$ is the rightmost column border in P. The *Estimate_Scale(T,P)* subroutine (details in the journal version) finds for each location in T all the borders of P in T that matches that location. Then, the borders in P are compared with the borders in T to give a scale estimate for each location.

Corollary 3. *The Scale Elimination Stage can be done in $O(n^2m)$ time using $O(n^2)$ space.*

4.2 Candidate Consistency Stage

A *candidate* is a location in the text where the pattern may occur. We note that in this stage we consider only scale $r^l_{i,j}$ for each location (i, j). The reason is that all scales in the range $[r^l_{i,j}, r^h_{i,j})$ share the same borders. Thus, if there is a scale $r \in [r^l_{i,j}, r^h_{i,j})$ such that P^{rX} appears at location (i, j) then $P^{r^l_{i,j}X}$ must also appear at location (i, j) since $P^{r^l_{i,j}X}$ is a subpattern of P^{rX} (it is the smallest and agrees both on the border and the characters). We will deal with all other scales in the range $[r^l_{i,j}, r^h_{i,j})$ at the occurrence recognition stage.

We say that two candidates are *consistent* if they expect the same text characters in their region of overlap. The idea of this stage is to perform duels between candidates and eliminate candidates that are not consistent. The duels are done only between candidates within the same row, using a special witness table.

Two candidates are *consistent* if they expect the same characters in their overlap region. In this stage we perform duels between candidates within the same row and eliminate candidates that are not consistent.

Witness Table Construction. We describe the witness table construction for the case that the overlapping candidate from the right is of scale greater than or equal to the scale of the candidate from the left. The opposite case is similar.

Let P be an $m \times m$ pattern, and let $y_1, \ldots, y_t$ be the column borders of P. The construction of the witness table for P is as follows. For each column c consider the suffix of P starting from c (i.e. columns $c, c + 1, \ldots, m$), denoted by *suff*(P, c), and the borders within *suff*(P, c), denote by $x_{c_1}, x_{c_2}, \ldots, x_{c_k}$ and $y_{c_1}, y_{c_2}, \ldots, y_{c_l}$.

Now, assume there is a candidate α at location (i, j) in the text of scale r_α such that there is another candidate, β, at location (i, j'), $j' > j$ of scale $r_\beta \geq r_\alpha$, such that column j' is overlapping the scaled column c of α. Then, it is clear that the two candidates agree on the overlap, *iff* the upper-left area of P scaled to r_β agrees with *suff*(P, c) scaled to r_α both on the borders **and** on the characters between the borders (see Figure 2).

Denote the borders of *suff*(P, c) by $SB(P, c)$ and the matching borders of the upper left area of P by $PB(P, c)$. Now, for each column c, naively search for matching borders and characters in the upper left area of P. If matching borders and characters are found, we mark column c as a consistent column, and save $PB(P, c)$ and $SB(P, c)$. Otherwise, we save the location of the conflict, denoted by *conflict_loc(c)*.

Observe, that $SB(P, c{+}1) \subseteq SB(P, c)$ for every column $1 \leq c < m$. Therefore, instead of storing all the $PB(P, c)$ and $SB(P, c)$ for every column c, we will only store $SB'(P, c)$ which is defined to be $SB(P, c) \setminus SB(P, c + 1)$ for $c < m$ and $SB(P, m)$ for $c = m$. Similarly, we define and store only the $PB'(P, c)$ for every column c in P.

Corollary 4. *The Witness Table can be constructed in $O(m^3)$ time using $O(m^2)$ space.*

The Dueling Order. Before we begin the actual dueling we perform a pre-dueling step where we compare $PB(P, c)$ and $SB(P, c)$ for every $1 < c < m$ and for every candidate in the row. All candidates that do not match the borders are eliminated. Now, when we want to compare two overlapping candidates all we need to do is check if they are consistent or conflicting candidates using the witness table. Thus, performing a duel takes $O(1)$ time, with additional $O(m)$ work for each candidate for the pre-dueling step, which is done using the $PB'(P, c)$ and $SB'(P, c)$.

For this subsection, we assume there are no inclusions between the candidates within the same row. Subsection 4.2 handles inclusions between candidates.

The dueling stage for row i is done as follows. Denote by $CCL(i)$ the consistent candidates list for row i in the text. At the beginning, $CCL(i)$ is empty. Then, we start adding candidates to $CCL(i)$ by traversing all the candidates in row i from right to left, such that at each step, all the candidates in $CCL(i)$ are consistent with each other.

The following subroutine determines if a candidate α should be added to $CCL(i)$ or be eliminated, assuming all candidates to the right of α are either in $CCL(i)$ or eliminated. Note that $FRL(\beta)$ of a candidate β is defined to be the first (non-eliminated) candidate to the right of β such that it is larger in size than β, i.e. $m_\beta < m_{FRL(\beta)}$, and is overlapping β. If no such candidate exist, $FRL(\beta)$ is set to NIL. In the journal version we will describe how to find FRL in $O(1)$ time, where updates are done in $O(m)$ time when a candidate is eliminated.

$AddToCCL(\alpha)$

```
0. Let $B = \{\beta_1, \beta_2, \ldots, \beta_k\} \subseteq CCL(i)$ be the candidates overlapping $\alpha$, ordered
   from left to right.
1. $leftmost \leftarrow 1$, $last\_success \leftarrow$ NIL
2. $\beta \leftarrow \beta_{leftmost}$
3. if $\beta \notin B$ - return TRUE
4. duel $\alpha$ with $\beta$
5. if $\alpha$ and $\beta$ agree on the overlap then:
   (a) if $m_\alpha \leq m_\beta$ - return TRUE
   (b) if $m_\alpha > m_\beta$: Set $last\_success \leftarrow \beta$, $\beta \leftarrow FRL(\beta)$ and goto step 3
6. if $\alpha$ and $\beta$ do not agree on the overlap then:
   (a) if $\alpha$ is eliminated - return FALSE
   (b) if $\beta$ is eliminated then:
        i. if $\beta = \beta_{leftmost}$ then $leftmost \leftarrow leftmost + 1$ and goto step 2
       ii. else $\beta \leftarrow FRL(last\_success)$ and goto step 3
```

Lemma 3. *If AddToCCL(α) returns TRUE, then α is consistent with all the candidates in $CCL(i)$.*

Denote a successful duel to be a duel where the two candidates agree on the overlap.

Lemma 4. *Each candidate performs at most $O(m)$ successful duels.*

Corollary 5. *The total running time of the candidate consistency stage is $O(n^2m)$ using $O(n^2)$ space.*

Handling Inclusions. At this point we assume there are at least two column borders in the pattern, otherwise, we can skip the dueling step and simply verify each candidate in $O(m)$ time using subrow and subcolumn repetition queries.

Lemma 5. *Let α be a scaled occurrence of an $m \times m$ pattern P. Then, α has $O(m)$ occurrences, within the same row, fully contained in α. Furthermore, all of them are of distance of $O(m)$ from α.*

Note that we do not need to duel between candidates such that one contains the other, since we know they share the same borders and thus expect the same characters. So before the actual dueling is done, after the pre-dueling step, we perform a scan for inclusions. We traverse the candidates in the current row from left to right, each candidate needs to checks only in the first $O(m)$ locations from it (by Lemma 5).

Once a candidate has found candidates that are included within him, the included candidates are removed from the candidates list, and the containing candidate, marked as such, saves those candidates in a private list. After this step is done, there are no inclusions between the candidates within the same row, thus, we can perform the dueling in the order described in Subsection 4.2.

When comparing the including candidate α with other candidates (not included in α), α is considered as a "representative" of the candidates included in α, such that if a candidate is consistent with α, it is also consistent with all the candidates included in α. On the other hand, if a candidate is inconsistent with α it might be consistent with other candidates in α. Thus, if α is eliminated, we cannot just eliminate all candidates in α.

To handle the case of α being eliminated, we define a tree-like hierarchy between α and the candidates contained in α, so that if α is eliminated, new candidates will "appear" and be the "representatives" of the rest. We use the following lemma to define the hierarchy.

Lemma 6. *Let α be a candidate containing the candidates $\beta_1, \ldots, \beta_k$, where β_1 is the leftmost and β_k is the rightmost. Then, there are two possibilities:*

1. *β_1 contains $\beta_2, \ldots, \beta_k$.*
2. *β_1 and another candidate β_i contains the rest, i.e., every candidate within $\{\beta_2, \ldots, \beta_k\} \setminus \{\beta_i\}$ is either contained in β_1 or in β_i (or in both). In addition β_1 does not contain β_i and vice versa.*

We define the hierarchy tree as follows. The root of the tree is, of course, α. α has one or two children, the left child is β_1 and (in the case 2) the right child is β_i, defined in Lemma 6. If α has two children, then the rest of the candidates are divided such that all candidates that are contained in β_i are in β_i's subtree, and the rest are in β_1's subtree. Since each child has a containing relation with the other candidates in it's subtree, we can recursively define the inner-hierarchy just like α's hierarchy.

Finally, when a containing candidate is eliminated, the child/children become the root/s of the "containing hierarchy", and thus, each root becomes a separate "representative" of a "containing hierarchy".

The only thing left is to show is how the appearance of new candidates, when a containing candidate is eliminated, fits in the dueling-order defined in Subsection 4.2. There are two issues to address. The first is the case where a new candidate is contained in another candidate. In order to solve this problem, each new candidate checks $O(m)$ location to the left (defined by Lemma 5) to see if it is contained or not. If a containing candidate is found, the new candidate searches the containing candidate's tree for its new location. This whole process takes $O(m)$ time and is done at most twice per eliminated candidate.

The second issue is the updating of the FRL values. Whenever a new candidate appears, it searches for the FRL exactly the same way the initial FRL values are set, by traversing an FRL series. Corollary 2 guarantees this series is of length $O(m)$. Then, the new candidate searches the FRL^{-1} list to find the right location in the list, which again takes $O(m)$ time (detailed proof of this will appear in the journal version). We note that the updating of the FRL values pointing to α when α is eliminated, is done exactly as the updating when a regular candidate is eliminated.

In summary, the additional work done to handle inclusions for each row consists of an inclusion scan, which takes $O(n)$ time, building the containing hierarchy, which could take up to $O(k^2)$ time for k candidate, and since $k = O(m)$, gives a total of $O(nm)$ time per row, and the updating of new inclusions and new FRL values which can take up to $O(m)$ time per eliminated candidate. All together, the additional work done is $O(nm)$ time per row, $O(n^2m)$ for the whole consistency stage, using $O(n^2)$ additional space.

4.3 Candidate Verification Stage

In the candidate verification stage we verify that each candidate left has exactly the character values it expects within its bounds. Since all the candidates left are consistent, we do not need to check overlapping areas more than once. The verification stage, just like the consistency stage, is done for each row separately.

The verification for row i is done by traversing candidates in the row, and for each candidate at location (i, j), compare the characters at column j starting from row i, with the scaled column that one of the candidates that contains location (i, j) is expecting.

However, we must verify each column with the largest candidate α overlapping that column. After we verify the column against α, we save the number of characters that the text and α agree on, starting from location (i,j) and going down. In order to find the largest candidate that is overlapping every column, we use the Maximal Interval Problem, which will be formally defined and solved in the journal version. The intervals are $[j, j + m_{r^l_{i,j}} - 1]$ for every candidate at location (i,j). Notice that finding the largest candidates that is containing column j is exactly the problem of finding the largest interval containing location j within this set of intervals. Finding the maximal interval for each location in the row is done in $O(n)$ time and space.

The actual verification is done for each row in $O(m)$ time using subcolumn repetition queries. The ouput is stored in an array V, such that $V[j]$ contains the number of characters the text and the overlapping candidate at column j agree on until the first mismatch (or the end of the column).

Finally, each candidate (i,j) is verify separately, by performing a Range Minimum Query on V with the interval $[j, j + m_{r^l_{i,j}} - 1]$. If the result is at least $m_{r^l_{i,j}}$, then the candidate is successfully verified, and we know that there is a pattern occurrence at location (i,j) in the text with scaled of $r^l_{i,j}$. Otherwise, there are no pattern occurrences at location (i,j) in the text. We note the Range Minimum Query problem is well known in the literature and can be solved in $O(1)$ time per query, using linear time for preprocessing (e.g. [10]).

Corollary 6. *The Candidate Verification Stage can be done in $O(n^2m)$ time using $O(n^2)$ space.*

4.4 Occurrence Recognition Stage

At the candidate consistency and candidate verification stages we've considered only the $r^l_{i,j}$ scales for every location (i,j). As mentioned above, the reason is because all scales in the range $[r^l_{i,j}, r^h_{i,j})$ share the same borders, and thus, there is a pattern occurrence at location (i,j) only if there is a pattern occurrence at location (i,j) with scale $r^l_{i,j}$.

At this stage, we traverse all locations in the text where a pattern occurrence was found with scale $r^l_{i,j}$ for each location (i,j). Then, we check if there are any other pattern occurrences of size larger than $m_{r^l_{i,j}} \times m_{r^l_{i,j}}$ that also appear at location (i,j). Note that we only search for scales in the range $[r^l_{i,j}, r^h_{i,j})$.

Since all occurrences at location (i,j) share the same borders, (a) Lemma 2 guarantees that there are no more than $m+2$ different occurrences and (b) if we know that a $k \times k$ occurrence α appears at location (i,j), all we need to do in order to confirm that there is also a $(k+1) \times (k+1)$ occurrence, is to check if the last column of α is equal the next column on the right, the last row is equal to the next row on the bottom and the character in $(i+k-1, j+k-1)$ is equal to the one in $(i+k, j+k)$.

Thus, we can begin checking occurrences starting at size $(m_{r^l_{i,j}}+1) \times (m_{r^l_{i,j}}+1)$ up until size of $(m_{r^l_{i,j}} + m + 1) \times (m_{r^l_{i,j}} + m + 1)$, each time comparing a row a column and a character, which can be done in $O(1)$ time using subrow and subcolumn repetition queries. The corollary follows.

Corollary 7. *The Occurrence Recognition Stage can be done in $O(m)$ time per text location, $O(n^2m)$ time for the entire text and using $O(n^2)$ space.*

Acknowledgement. This research was conducted while the first author was partially supported by NSF grant CCR-01-04494 and ISF grant 35/05.

References

1. A. Amir, G. Benson, and M. Farach. An alphabet independent approach to two dimensional pattern matching. *SIAM J. Comp.*, 23(2):313–323, 1994.
2. A. Amir, A. Butman, and M. Lewenstein. Real scaled matching. *Information Processing Letters*, 70(4):185–190, 1999.
3. A. Amir, A. Butman, M. Lewenstein, and E. Porat. Real two dimensional scaled matching. In *Proc. 8th Workshop on Algorithms and Data Structures (WADS '03)*, pages 353–364, 2003.
4. A. Amir, A. Butman, M. Lewenstein, E. Porat, and D. Tsur. Efficient one dimensional real scaled matching. In *Proc. 11th Symposium on String Processing and Information Retrieval (SPIRE '04)*, pages 1–9, 2004.
5. A. Amir and G. Calinescu. Alphabet independent and dictionary scaled matching. *Proc. 7th Annual Symposium on Combinatorial Pattern Matching (CPM 96)*, pages 320–334, 1996.
6. A. Amir, G.M. Landau, and U. Vishkin. Efficient pattern matching with scaling. *Journal of Algorithms*, 13(1):2–32, 1992.
7. A. Amir, D. Tsur, and O. Kapah. Faster two dimensional pattern matching with rotations. In *Proc. 15th Annual Symposium on Combinatorial Pattern Matching (CPM '04)*, pages 409–419, 2004.
8. K. Fredriksson, V. Mäkinen, and G. Navarro. Rotation and lighting invariant template matching. In *Proceedings of the 6th Latin American Symposium on Theoretical Informatics (LATIN'04)*, LNCS, pages 39–48, 2004.
9. K. Fredriksson and E. Ukkonen. A rotation invariant filter for two-dimensional string matching. In *Proc. 9th Annual Symposium on Combinatorial Pattern Matching (CPM '98)*, pages 118–125. Springer, LNCS 1448, 1998.
10. H. N. Gabow, J. L. Bentley, and R. E. Tarjan. Scaling and related techniques for geometry problems. *Proc. 16th ACM Symposium on Theory of Computing*, 67:135–143, 1984.
11. G. M. Landau and U. Vishkin. Pattern matching in a digitized image. *Algorithmica*, 12(3/4):375–408, 1994.

Approximation of RNA Multiple Structural Alignment*

Marcin Kubica[1], Romeo Rizzi[2], Stéphane Vialette[3], and Tomasz Waleń[1]

[1] Institute of Informatics, Warsaw University
Banacha 2, 02-097 Warszawa, Poland
{kubica, walen}@mimuw.edu.pl
[2] Dipartimento di Matematica ed Informatica (DIMI),
Università di Udine, Via delle Scienze 208, I-33100 Udine, Italy
Romeo.Rizzi@dimi.uniud.it
[3] Laboratoire de Recherche en Informatique (LRI), UMR CNRS 8623
Faculté des Sciences d'Orsay - Université Paris-Sud, 91405 Orsay, France
Stephane.Vialette@lri.fr

Abstract. In the context of non-coding RNA (ncRNA) multiple structural alignment, Davydov and Batzoglou introduced in [7] the problem of finding the largest nested linear graph that occurs in a set $\mathcal{G}$ of linear graphs, the so-called Max-NLS problem. This problem generalizes both the *longest common subsequence* problem and the *maximum common homeomorphic subtree* problem for rooted ordered trees.

In the present paper, we give a fast algorithm for finding the largest nested linear subgraph of a linear graph and a polynomial-time algorithm for a fixed number (k) of linear graphs. Also, we strongly strengthen the result of [7] by proving that the problem is **NP**-complete even if $\mathcal{G}$ is composed of nested linear graphs of height at most 2, thereby precisely defining the borderline between tractable and intractable instances of the problem. Of particular importance, we improve the result of [7] by showing that the Max-NLS problem is approximable within ratio $O(\log m_{opt})$ in $O(kn^2)$ running time, where m_{opt} is the size of an optimal solution. We also present $\mathcal{O}(1)$-approximation of Max-NLS problem running in $\mathcal{O}(kn)$ time for restricted linear graphs. In particular, for ncRNA derived linear graphs, an $\frac{1}{4}$-approximation is presented.

1 Introduction

Non-coding RNA, unlike regular genes, are not translated to proteins, but perform a variety of catalytic, structural and regulatory functions; transfer RNA, ribosomal RNA and spliceosomal RNA are textbook examples. The RNA sequences consist of four kinds of nucleotides: A (adenine), C (cytosine), G (guanine) and U (uracil). RNA sequences tend to fold, forming secondary and tertiary

* This research was partially supported by the Polish Scientific Research Committee (KBN) under grant GR-1946 and by the French-Italian Galileo Project PAI 08484VH.

M. Lewenstein and G. Valiente (Eds.): CPM 2006, LNCS 4009, pp. 211–222, 2006.

structures, stabilized by bonds between nucleotides. Three kinds of such bonds are most frequent: A–U, G–C and U–G.

The structural stability and function of ncRNA genes are largely determined by the formation of stable secondary structures through complementary bases (see [18] for a detailed introduction to RNA secondary structures). Much research work has been done on the structural comparison of ncRNA sequences [2, 3, 4, 7, 13, 14]. Davydov and Batzoglou proposed in [7] a new model for structural alignment of multiple ncRNA sequences, based on finding the largest common secondary structure. The problem of computing the largest common secondary structure for the given set of k ncRNA sequences can be solved using the stochastic context-free grammars, in $O(n^{3k})$ time complexity [12].

In this paper, we focus on the problem of secondary structure alignment for multiple ncRNA sequences. We follow a general model presented in [7], where ncRNA sequences are modelled by linear graphs and the common secondary structure is modelled using the maximum common nested linear subgraphs (Max-NLS problem). Unfortunately this problem is NP-complete, but can be approximated within $O(\log^2 n)$ factor in $O(k \cdot n^5)$ running time [7]. The authors proposed to approximate Max-NLS with maximum level linear subgraphs (Max-LLS problem).

This paper is organized as follows. Section 2 presents some preliminaries. We give in Section 3 a fast and simple dynamic programming algorithm for finding a nested linear graph in a linear graph, and present in Section 4 a polynomial-time algorithm for a fixed number of linear graphs. Section 5 strongly refines the hardness result of [7] by giving a tight description of the borderline between **NP**-completeness and **P**. We present in Section 6 a faster algorithm for the Max-NLS problem and in Section 7 we improve the approximation ratio of [7]. Finally, Section 8 deals with the Max-NLS problem, for restricted linear graphs. Due to space constraints, several details and proofs are not presented in this paper.

2 Preliminaries

Basic familiarity with graph-theoretic terminology is assumed. For a graph G, we denote by $\mathbf{V}(G)$ the set of vertices and by $\mathbf{E}(G)$ the set of edges. The *order* and the *size* of G stand for $|\mathbf{V}(G)|$ and $|\mathbf{E}(G)|$, respectively. A *linear graph* of order n is a vertex-labeled graph where each vertex is labeled by a distinct integer from $\{1, 2, \ldots, n\}$. In case of linear graphs, we write an edge between vertices i and j, $i < j$, as the pair (i, j). Two edges of a graph are called *independent* if they do not share a vertex. A linear graph G is called *edge-independent* if it is composed of independent edges, *i.e.*, G is a matching.

Of particular interest are the relations between independent edges [17]. Let $e = (i, j)$ and $e' = (i', j')$ be two independent edges in a linear graph G. We write (i) $e < e'$ if $i < j < i' < j'$, (ii) $e \sqsubset e'$ if $i' < i < j < j'$, and (iii) $e \between e'$ if $i < i' < j < j'$. Two edges e and e' are said to be R-comparable for some $\mathrm{R} \in \{<, \sqsubset, \between\}$ if $e \,\mathrm{R}\, e'$ or $e' \,\mathrm{R}\, e$. Observe, that any two independent edges are R-comparable for some $\mathrm{R} \in \{<, \sqsubset, \between\}$. An edge-independent linear graph

G is called an *$\mathcal{R}$-comparable linear graph* for some non-empty $\mathcal{R} \subseteq \{<, \sqsubset, \between\}$ if any two distinct edges in G are R-comparable for some R $\in \mathcal{R}$. Let G be a linear graph. The *width* (resp. *height*) of G is the size of a maximum cardinality $\{<\}$-comparable (resp. $\{\sqsubset\}$-comparable) subset of $\mathbf{E}(G)$.

We now define the notion of *occurrence* of one linear graph in another. Let G_1 and G_2 be two linear graphs. The graph G_1 is said to *occur* in G_2 (or G_1 is called a *subgraph* of G_2) if one can obtain G_1 from G_2 (regardless of precise vertex labels) by a sequence of edge and vertex deletions. More formally, the deletion of vertex i consists in (1) the deletion of all the edges incident to vertex i, (2) the deletion of vertex i and (3) the relabeling of all vertices $j > i$ to $j - 1$.

For the purpose of ncRNA multiple structural alignment, convenient graphs are needed [7]. A linear graph is *non-crossing* if it does not contain two edges e and e' such that $e \between e'$. A $\{<, \sqsubset\}$-comparable linear graph is also called a *nested linear graph.* A $\{\sqsubset\}$-comparable subgraph of a linear graph G is called a *nested loop*. Let (i, j) be the outermost edge of a nested loop in G. Value $j - i$ is called the *diameter* of the nested loop.

Nested linear graphs can be viewed as collections of trees. Each edge in the nested linear graph corresponds to a node in a tree — e is the parent of e' iff $e' \sqsubset e$ and there is no such edge e'' that $e' \sqsubset e'' \sqsubset e$. Each $\sqsubset$-maximal edge in the nested linear graph is a root of some tree.

To shorten notation, a nested linear graph G of size n can be represented by a Dyck word of semi-length n over the alphabet $\mathcal{A} = \{a, b\}$. By abuse of notation, we continue to write G for the corresponding Dyck word. A nested linear graph G is *flat* if it can be written as $G = a^{h_1}b^{h_1}\ a^{h_2}b^{h_2}\ \ldots\ a^{h_k}b^{h_k}$ for some positive integers $h_1, h_2, \ldots, h_k$. A flat linear graph is called *level* if it can be written as $G = (a^h b^h)^w$ for some positive integers h and w.

For a given linear graph G we can consider its level subgraphs of given height and maximum width. The relation between the height and the maximum width of level subgraphs is called *level signature* of the linear graph. More formally, level signature of G is a function $s : \mathbb{N} \to \mathbb{N}$ such, that: (i) $s(h)$ is the maximum width of a level subgraph of G with height h; (ii) if G has no level subgraph of height h, then $s(h) = 0$.

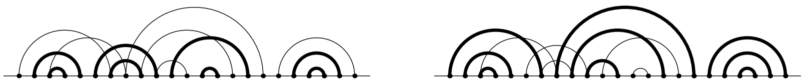


Fig. 1. Maximum level subgraphs of G with height 2 (on the left), and height 3 (on the right). The level signature of the graph is: $s(1) = 5$, $s(2) = 4$, $s(3) = 3$, $s(4) = 0$.

Genomic sequences can be naturally viewed as linear graphs. A sequence of nucleotides $S = (a_1, a_2, \ldots, a_n)$ corresponds to a linear graph whose vertices are the nucleotides and there is an edge between two nucleotides iff there can be a bond between them. In this paper we also investigate more abstract correspondence. Let $S = (a_1, a_2, \ldots, a_n)$ be a sequence over some fixed finite alphabet Σ, and let $\xi \subseteq \Sigma^2$ be a fixed **symmetrical** relation. A linear graph *derived* from S, denoted as $\mathbf{G}_\xi(S)$, is such a linear graph of order n, in which there is an edge (p, q) iff $p \neq q$ and $a_p \,\xi\, a_q$. Throughout this paper we assume that Σ and ξ are fixed. We will call linear graphs derived from sequences over Σ, *restricted linear graphs* (RLG). Clearly, for $\Sigma_{\mathsf{RNA}} = \{\mathrm{A}, \mathrm{U}, \mathrm{G}, \mathrm{C}\}$ and $\xi_{\mathsf{RNA}} = \{(\mathrm{A}, \mathrm{U}), (\mathrm{U}, \mathrm{A}), (\mathrm{U}, \mathrm{G}), (\mathrm{G}, \mathrm{U}), (\mathrm{G}, \mathrm{C}), (\mathrm{C}, \mathrm{G})\}$ we get RLG modelling ncRNA sequences.

Let $\mathcal{G} = \{G_1, G_2, \ldots, G_k\}$ be a set of linear graphs. The Max-NLS problem is to find a maximum size common nested linear subgraph of $G_i \in \mathcal{G}$. We denote it by Max-NLS$(G_1, \ldots, G_k)$. Nested linear subgraphs and Max-NLS problem have been introduced in [7] to represent possible structural alignments of ncRNA. Nested linear subgraphs correspond to different structural alignments of a sequence of nucleotides, and the edges of a nested linear subgraph correspond to bonds stabilising the structure of ncRNA, as shown in fig. 2.

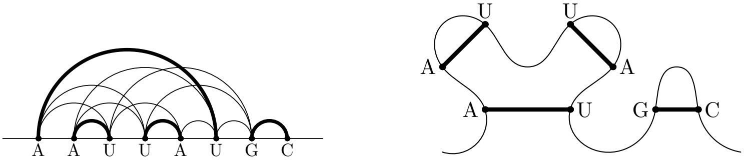


Fig. 2. Example of linear graph for sequence AAUUAUGC, its Max-NLS and corresponding bonds between nucleotides

The Max-LLS problem is to find a maximum size common level linear subgraph of $G_1, \ldots, G_k$ (denoted by Max-LLS$(G_1, \ldots, G_k)$). Level linear subgraphs have been used to approximate Max-NLS problem in [7]. The MNL(G) problem is to find a maximum size nested loop in G. Nested loops and the MNL problem have been used to model ncRNA structural alignments in [4]. The authors also present $\mathcal{O}(n^2)$ algorithm for computing MNL.

3 Finding a Maximum Size Nested Linear Graph in a Linear Graph

Felsner *et al.* considered in [8] the matching problem regardless of a precise pattern definition. In this context, they introduced the concept of *circle trapezoid graphs* (CT-graphs), a class of graphs that contains trapezoid graphs, circle graphs and circular-arc graphs as subclasses, and proved that, given a CT-graph G with $m = |\mathbf{E}(G)|$, a maximum size nested subgraph of G can be found in $\mathcal{O}(m^2)$ time. On the other hand the Max-NLS(G) can be computed in $\mathcal{O}(n^3)$ time [16], where $n = |\mathbf{V}(G)|$.

In this brief section, we improve that result for linear graphs by giving a simple dynamic programming algorithm for finding a maximum size nested subgraph of a linear graph in $\mathcal{O}(n^2 + nm)$ time and $\mathcal{O}(n^2)$ space. For each pair (i, j) with $1 \leq i < j \leq n$, let $G_{i,j}$ denote the subgraph of G induced by the vertices $i, \ldots, j$.

We denote by $N^-(j)$ the set of vertices $N^-(j) = \{q : q < j \wedge (q,j) \in \mathbf{E}(G)\}$. For $1 \leq i < j \leq n$, let $\mathsf{opt}[i,j]$ denote the maximum size of a nested subgraph of the linear graph $G_{i,j}$ (for $i \geq j$ we have $\mathsf{opt}[i,j] = 0$). For $1 \leq i < j \leq n$, $\mathsf{opt}[i,j]$ can be obtained using the following formula:

$$\mathsf{opt}[i,j] = \max \begin{cases} \mathsf{opt}[i,j-1]\,, \\ 1 + \mathsf{opt}[i,q-1] + \mathsf{opt}[q+1,j-1] : q \in N^-(j) \ \wedge \ q \geq i \end{cases}$$

The maximum nested subgraph of $G_{i,j}$ either does not not take vertex j, in which case $\mathsf{opt}[i,j] = \mathsf{opt}[i,j-1]$, or it takes an edge $(q,j) \in \mathbf{E}(G)$ with $i \leq q < j$.

Using dynamic programming, the array opt can be computed in $O(n^2 + nm)$ time, since computation of each $\mathsf{opt}[i,j]$ requires $O(|N^-(j)|)$ steps.

4 A Polynomial-Time Algorithm for Fixed $|\mathcal{G}|$

According to [11], given a linear graph G_1 of size m_1 and a nested linear graph G_2 of size m_2, an occurrence of G_2 in G_1 can be found in $\mathcal{O}(m_2 \log m_2 + m_1 m_2)$ time. Valiente proposed in [15] a dynamic programing algorithm for finding the largest nested linear graph that occurs in two nested linear graphs (see also [19]). In this section, we give a $\mathcal{O}(m^{2k} \log^{k-2} m^k \log\log m^k)$ time dynamic programming algorithm, where $m = \max\{|\mathbf{E}(G_i)| : G_i \in \mathcal{G}\}$ and $k = |\mathcal{G}|$, for finding the largest nested linear graph that occurs in a fixed number of linear graphs.

We need some additional definitions and notations. We use the notions of *trapezoid diagrams* and *d-trapezoid diagrams* introduced in [6] and [9], respectively. Assume d is a non-negative integer and let $L^1, L^2, \ldots, L^{d+1}$ be $d+1$ parallel lines indexed by their ordering in the plane. A graph G is called a *d-trapezoid graph* [8, 5] if there exist families of intervals $T_u = \{I^i_u = [l^i_u, r^i_u] : l^i_u, r^i_u \in L^i, 1 \leq i \leq d+1\}$, $u \in \mathbf{V}(G)$, satisfying $\{u,v\} \in \mathbf{E}(G)$ if and only if $Q_u \cap Q_v \neq \emptyset$, where Q_x denotes the closed polygon $(l^1_x, l^2_x, \ldots, l^{d+1}_x, r^{d+1}_x, r^d_x, \ldots, r^1_x)$. We refer to the family $\mathcal{T}_G = \{T_u : u \in \mathbf{V}(G)\}$ to as the *d-trapezoid diagram* of G. Note that 0-trapezoid graphs are precisely interval graphs [10], 1-trapezoid graphs are the usual trapezoid graphs [6], and d-trapezoid graphs are comparability graphs of posets with interval dimension at most $d+1$ [9].

Based on a geometric representation of 1-trapezoid graphs by boxes in the plane, Felsner *et al.* [8] designed an optimal $\mathcal{O}(n \log n)$ algorithm for finding a maximum weighted independent set on such graphs. Of particular importance, they proved that the ideas behind the weighted independent set for trapezoid graphs carry over to higher dimension leading to a $\mathcal{O}(n \log^d n)$ time algorithm for d-trapezoid graphs of order n. This has been improved in [1] to $\mathcal{O}(n \log^{d-1} n \log\log n)$ time. We now turn to defining an irreflexive, transitive and anti-symmetric relation $\sqsubset$ on $\mathcal{T}_G$. Let G be a d-trapezoid graph and $\mathcal{T}_G = \{T_u : u \in \mathbf{V}(G)\}$ be the corresponding d-trapezoid diagram. Let $T_u = \{I^i_u = [l^i_u, r^i_u] : l^i_u, r^i_u \in L^i, 1 \leq i \leq d+1\}$ and $T_v = \{\mathcal{I}^i_v = [l^i_v, r^i_v] : l^i_v, r^i_v \in L^i, 1 \leq i \leq d+1\}$ be two d-trapezoids of $\mathcal{T}_G$. We say that the d-trapezoid T_u is *strictly contained* in T_v, written $T_u \sqsubset T_v$, if $l^i_v < l^i_u$ and $r^i_u < r^i_v$ for all $1 \leq i \leq d+1$.

Let $\mathcal{G} = \{G_1, G_2, \ldots, G_k\}$ be an instance of the Max-NLS problem. We associate to $\mathcal{G}$ an $(k-1)$-trapezoid diagram as follows. For each $1 \leq i \leq k$, the graph G_i is associated to Line L^i, and for each $(x, y) \in \mathbf{E}(G_i)$ we define an interval $I^i_{x,y} = [x, y]$ on line L^i. We denote by $\mathcal{I}_{G_i}$ the set of all intervals on L^i that are associated to the graph G_i, *i.e.*, $\mathcal{I}_{G_i} = \{I^i_{x,y} : (x, y) \in \mathbf{E}(G_i)\}$. The $(k-1)$-*trapezoid diagram induced by* $\mathcal{G}$, written $\mathcal{T}[\mathcal{G}]$, is defined as follows:

$$\forall I_1 \in \mathcal{I}_{G_1},\ \forall I_1 \in \mathcal{I}_{G_2},\ \ldots,\ \forall I_k \in \mathcal{I}_{G_k}, \qquad \{I_1, I_2, \ldots, I_k\} \in \mathcal{T}[\mathcal{G}]\,.$$

Clearly, $|\mathcal{T}[\mathcal{G}]| = \prod_{G_i \in \mathcal{G}} |\mathbf{E}(G_i)|$. Having disposed of these preliminaries, we now turn to presenting our algorithm, referred hereafter to as Algorithm nested-linear-subgraph, for finding the largest nested linear graph G that occurs in a family of linear graphs $\mathcal{G}$. The basic idea is to associate to each $(k-1)$-trapezoid $T \in \mathcal{T}[\mathcal{G}]$ a weight $\omega(T)$ denoting the maximum size of a nested linear graph that occurs in the family of linear graphs induced by T and all $(k-1)$-trapezoids strictly included in T. This is done in turn by dynamic programming according to a linear extension of $(\mathcal{T}[\mathcal{G}], \sqsubset)$. We need the following subroutine: Given an $(n-1)$-trapezoid diagram $\mathcal{T}$ and a function $\omega : \mathcal{T} \to \mathbb{N}^+$, we refer to the algorithm for finding a maximum weighted disjoint subset of $\mathcal{T}$ (in terms of disjoint induced closed polygons) as max-weighted-independent-set$(\mathcal{T}, \omega)$ [8, 1]. A more schematic description of Algorithm nested-linear-subgraph$(\mathcal{G} = \{G_1, G_2, \ldots, G_k\})$ is given below:

```
begin
    Compute the $(k-1)$-trapezoid diagram $\mathcal{T}[\mathcal{G}]$ induced by $\mathcal{G}$
    Compute any linear extension $\Phi$ of $(\mathcal{T}[\mathcal{G}], \sqsubset)$
    foreach $T \in \mathcal{T}[\mathcal{G}]$ do $\omega(T) := 0$
    foreach $T \in \mathcal{T}[\mathcal{G}]$ with respect to $\Phi$ do
        $\mathcal{T}' := \{T' : T' \sqsubset T\}$.
        $\mathcal{T}'' :=$ max-weighted-independent-set$(\mathcal{T}', \omega)$
        $\omega(T) := 1 + \sum_{T'' \in \mathcal{T}''} \omega(T'')$
    end
    $\mathcal{T}^* :=$ max-weighted-independent-set$(\mathcal{T}[\mathcal{G}], \omega)$
    return $\sum_{T \in \mathcal{T}^*} \omega(T)$
end
```

Proposition 1. *The* Max-NLS *problem is solvable in* $\mathcal{O}(m^{2k} \log^{k-2} m^k \log\log m^k)$ *time, where* $m = \max\{|\mathbf{E}(G_i)| : G_i \in \mathcal{G}\}$ *and* $k = |\mathcal{G}|$.

This result gains in interest if we compare Proposition 1 to the related LAPCS problem restricted to two nested arc-annotated sequences, *i.e.*, the LAPCS(Nested, Nested) problem, restricted to unary alphabet, which has been proved to be **NP**-complete in [13] (only two nested arc-annotated sequences, and hence two linear graphs here).

5 Hardness Results

It is proved in [7] that the Max-NLS problem is **NP**-complete even when restricted to edge-independent linear graphs. We sharply strengthen this result by showing that the problem is hard even for flat linear graphs of height at most 2. Observe that our result gives a precise borderline between tractable and intractable instances of the Max-NLS problem (the problem is indeed trivially polynomial-time solvable for nested linear graphs of height at most 1: find the stability of $|\mathcal{G}|$ associated interval graphs and return the maximum stability found). Our result is a two-step procedure. We begin by proving the **NP**-hardness of a new list problem, *i.e.*, the Longest Common Sublist problem. Next, we give a polynomial-time reduction from the Longest Common Sublist problem to prove that the Max-NLS problem is **NP**-complete even when restricted to simple instances solely composed of flat linear graphs of height at most 2.

We need new additional notations. When L is a list of integers, we denote by $\mathsf{len}(L)$ the length of L and by $L[i]$ the value of the i-th integer in L, $1 \leq i \leq \mathsf{len}(L)$. A *sublist* of L is any list obtained from L by dropping some of the elements in L. Clearly, L admits $2^{\mathsf{len}(L)}$ sublists. We can now state formally the Longest Common Sublist decision problem.

Longest Common Sublist

Instance: Lists of positive integers $L_1, L_2, \ldots, L_k$, and a positive integer m.

Question: Are there lists $\tilde{L}_1, \tilde{L}_2, \ldots, \tilde{L}_k$, where $\tilde{L}_i$ is a sublist of L_i, $1 \leq i \leq k$, and $\mathsf{len}(\tilde{L}_1) = \mathsf{len}(\tilde{L}_2) = \ldots = \mathsf{len}(\tilde{L}_k) = \ell$, such that $\sum_{1 \leq j \leq \ell} \min\{\tilde{L}_i[j] : 1 \leq i \leq k\} \geq m$?

Proposition 2. *The* Longest Common Sublist *problem is* **NP**-*complete even if all integer values in the lists are either* 1*'s or* 2*'s.*

Most of the interest in the Longest Common Sublist problem stems from the following proposition.

Proposition 3. *The* Max-NLS *problem for flat linear graphs of height at most* 2 *is* **NP**-*complete.*

Proof. Let an instance of the Longest Common Sublist problem - where all integer values in the lists are either 1's or 2's - be given by n lists of positive integers $L_1, L_2, \ldots, L_k$, and by a positive integer m. We construct a corresponding set $\mathcal{G}$ of k linear graphs as follows (we abbreviate in a natural way a nested linear graph G of size m to a Dyck word of semi-length m over the alphabet $\mathcal{A} = \{a, b\}$). For each list L_i, $1 \leq i \leq k$, we add to $\mathcal{G}$ the nested linear graph G_i defined by $G_i = a^{L_i[1]}b^{L_i[1]}\ a^{L_i[2]}b^{L_i[2]}\ \ldots\ a^{L_i[\mathsf{len}(L_i)]}b^{L_i[\mathsf{len}(L_i)]}$.. It is easily seen that $\mathcal{G}$ is composed of k flat linear graphs of height at most 2 since all integer values in the lists are either 1's or 2's, and that our construction can be carried on in polynomial-time.

Suppose that there exist lists $\tilde{L}_1, \tilde{L}_2, \ldots, \tilde{L}_k$, where $\tilde{L}_i$ is a sublist of L_i, $1 \leq i \leq k$, and $\mathsf{len}(\tilde{L}_1) = \mathsf{len}(\tilde{L}_2) = \ldots = \mathsf{len}(\tilde{L}_k) = \ell$, such that $\sum_{1 \leq j \leq \ell} \min\{\tilde{L}_i[j] : 1 \leq i \leq k\} \geq m$. Let $\tilde{L}$ be the list of length ℓ defined by $\tilde{L}[j] = \min\{\tilde{L}_i[j] : 1 \leq i \leq k\}$ for $1 \leq j \leq \ell$. Now, consider the flat linear graph G_{sol} defined by $G_{\mathrm{sol}} = a^{\tilde{L}[1]} b^{\tilde{L}[1]}\ a^{\tilde{L}[2]} b^{\tilde{L}[2]}\ \ldots\ a^{\tilde{L}[\ell]} b^{\tilde{L}[\ell]}$. It can be easily verified that G_{sol} is a flat linear graph of size $\sum_{1 \leq j \leq \ell} \min\{\tilde{L}_i[j] : 1 \leq i \leq n\} \geq m$ that occurs in each $G_i \in \mathcal{G}$.

Conversely, suppose that there exists a linear graph G_{sol} of size at least m that occurs in each $G_i \in \mathcal{G}$. Since $\mathcal{G}$ is composed of flat linear graphs, G_{sol} is a flat linear graph. Therefore, G_{sol} can be written $G_{\mathrm{sol}} = a^{z_1} b^{z_1}\ a^{z_2} b^{z_2}\ \ldots\ a^{z_\ell} b^{z_\ell}$ for some non-negative integers $z_1, z_2, \ldots, z_\ell$, and $z_1 + z_2 + \ldots + z_\ell \geq m$. Consider the list of positive integers $\tilde{L}$ of length ℓ defined by $\tilde{L}[j] = z_j$ for $1 \leq j \leq \ell$. Clearly $\sum_{1 \leq j \leq \ell} \tilde{L}[j] = z_1 + z_2 + \ldots + z_\ell \geq m$. By construction, for all $1 \leq i \leq k$, there exists a sublist $\tilde{L}_i$ of L_i of length ℓ such that $\tilde{L}_i[j] \geq \tilde{L}[j]$. □

Here below we offer a reformulation of Proposition 3 that may be of independent interest.

Proposition 4. *Finding the largest common homeomorphic subtree in a set of ordered rooted trees of height at most* 3 *is an* **NP***-complete problem.*

6 Max-LLS Problem

Max-LLS problem was defined in [7], where it was used to approximate Max-NLS. The algorithm proposed in [7] to solve Max-NLS problem for k linear graphs of order n has $\mathcal{O}(k \cdot n^5)$ time complexity. In this section, we present an algorithm solving this problem in $\mathcal{O}(k \cdot n^2)$ time. For this and the following section, let $G_1, \ldots, G_k$ be given linear graphs of order n. First, for each G_i we compute its level signature. Then we compute minimum of k level signatures. This minimum gives us the shape of $\mathsf{Max\text{-}LLS}(G_1, G_2, \ldots, G_k)$. Knowing its shape, we can then reconstruct it.

For each G_i its level signature is computed in three steps. First, we compute an array MNL_i describing sizes of maximum nested loops for all fragments of G_i. Let $G'_{i,p,q}$ be a linear subgraph of G_i induced by vertices $p, \ldots, q$. For $1 \leq p < q \leq n$, $\mathsf{MNL}_i[p, q]$ is the size of $\mathsf{MNL}(G'_{i,p,q})$ (for $p \geq q$ we have $\mathsf{MNL}_i[p, q] = 0$). If $(p, q) \in \mathbf{E}(G_i)$ then $\mathsf{MNL}_i[p, q] = \max(\mathsf{MNL}_i[p{+}1, q],\ \mathsf{MNL}_i[p, q{-}1], \mathsf{MNL}_i[p{+}1, q{-}1]{+}1)$, otherwise $\mathsf{MNL}_i[p, q] = \max(\mathsf{MNL}_i[p + 1, q],\ \mathsf{MNL}_i[p, q - 1])$. Clearly, MNL_i can be computed in $\mathcal{O}(n^2)$ time using dynamic programming.

Next, we compute an array[1] NLW_i containing minimum diameter of nested loops of given size starting at given position. For $1 \leq p \leq n$ and $1 \leq h \leq \frac{n}{2}$, $\mathsf{NLW}_i[p, h]$ is the minimum such integer, that $v_{i,p}, \ldots, v_{i,p+\mathsf{NLW}_i[p,h]}$ contains a nested loop of size h (or 0 if such an integer does not exist).

[1] We assume that all arrays are implicitly initialized with zeros.

```
for $h = 1$ to $\lfloor \frac{n}{2} \rfloor$ do
    for $p = n - 1$ downto $1$ do
        if $\mathsf{NLW}_i[p+1,h] > 0$ then $\mathsf{NLW}_i[p,h] = \mathsf{NLW}_i[p+1,h] + 1$
        else if $\mathsf{MNL}_i[p,n] \geq h$ then $\mathsf{NLW}_i[p,h] = n - p$;
        while $\mathsf{NLW}_i[p,h] > 1 \wedge \mathsf{MNL}_i[p, p + \mathsf{NLW}_i[p,h] - 1] \geq h$ do
            $\mathsf{NLW}_i[p,h] = \mathsf{NLW}_i[p,h] - 1$
        end
    end
end
```

Let us have a closer look at the inner `while` loop. Please note that, for given p and h, if $\mathsf{NLW}_i[p,h] > 0$ and $\mathsf{NLW}_i[p+1,h] > 0$ then the `while` loop performed $\mathsf{NLW}_i[p+1,h] - \mathsf{NLW}_i[p,h] + 1$ iterations. If $\mathsf{NLW}_i[p,h] > 0$ and $\mathsf{NLW}_i[p+1,h] = 0$ then the `while` loop performed $n - p - \mathsf{NLW}_i[p,h]$ iterations. Otherwise $\mathsf{NLW}_i[p,h] = 0$ and the `while` loop performed no iterations. Hence, the total number of iterations made by this loop for given h is not greater than $n - 1 - \mathsf{NLW}_i(1,h) \leq n$. Therefore, the running time of this step is $\mathcal{O}(n^2)$.

Finally, for $1 \leq p \leq n$, we compute as $\mathsf{SIG}_i[p,h]$ the level signature of $G'_{i,p,n}$ for height h. It can be done in $\mathcal{O}(n^2)$ time, using dynamic programming and the following formula:

$$\mathsf{SIG}_i[p,h] = \begin{cases} \mathsf{SIG}_i[p + \mathsf{NLW}_i[p,h] + 1, h] + 1, & \text{if } \mathsf{NLW}_i[p,h] > 0 \\ 0 & \text{otherwise} \end{cases}$$

Obviously, the level signature of G_i for height h is in $\mathsf{SIG}_i[1,h]$ and the total time complexity of all three steps is $\mathcal{O}(n^2)$. The level signature SIG of common level subgraphs of $G_1, G_2, \ldots, G_k$ equals $\mathsf{SIG}[h] = \min_{i=1,\ldots,k} \mathsf{SIG}_i[1,h]$, and the size of the maximum common level subgraph is equal $\max_{h=1,\ldots,\lfloor \frac{n}{2} \rfloor} h \cdot \mathsf{SIG}[h]$. The actual maximum common level subgraph can be reconstructed basing on its height and from arrays SIG_i, NLW_i and MNL_i, in $\mathcal{O}(kn)$ time.

7 Approximation of Max-NLS

In [7], the authors prove that the optimal solution for Max-LLS problem gives $\mathcal{O}(\log^2 m_{opt})$-approximation for Max-NLS problem (where m_{opt} is the size of the optimal solution for Max-NLS). The proof from [7] consists of two steps: first Max-NLS problem is reduced to the problem of finding a maximum flat linear subgraph, and then it is further reduced to Max-LLS. In this section, we show how to reduce Max-NLS problem to Max-LLS directly, achieving $\mathcal{O}(\log m_{opt})$ approximation ratio. Please note, that we study here exactly the same approximation, but we prove a better approximation ratio.

Theorem 1. *Let $G_1, \ldots, G_k$ be given linear graphs of order n, m_{opt} and h_{opt} be respectively the size and the height of* Max-NLS$(G_1, \ldots, G_k)$*, and l be the size of* Max-LLS$(G_1, \ldots, G_k)$*. Then we have: $m_{opt} \leq \Theta(\log h_{opt}) \cdot l \leq \Theta(\log m_{opt}) \cdot l$.*

Proof. Let T be a collection of trees representing $\mathsf{Max\text{-}NLS}(G_1, \ldots, G_k)$. Obviously, T contains m_{opt} nodes. For each such node v, by $l(v)$ we will denote the height of the subtree rooted in v, and by $\mathsf{path}(v)$ we will denote a fixed path of length $l(v)$ starting in v and ending in some leaf of the subtree rooted in v. For all leaves we have $l(v) = 1$. Please note, that $\mathsf{path}(v)$ represents a nested loop of $l(v)$ edges. By $L(h)$ we will denote the set of nodes in T of height h, $L(h) = \{v : l(v) = h\}$. Clearly, $\sum_{i=1}^{h_{opt}} |L(i)| = m_{opt}$. We also define $S(h) = \{\mathsf{path}(v) : v \in L(h)\}$. Let us observe that $S(h)$ represents a set of nested loops that form a level linear subgraph with width $|L(h)|$, height h and size $s(h) = h \cdot |L(h)|$. Let h_{max} be such a height, for which $s(h)$ is maximum. Clearly, $s(h_{\mathsf{max}}) \leq l$. For any $i = 1, \ldots, h_{opt}$, we have $|L(i)| \leq \frac{s(h_{\mathsf{max}})}{i} \leq \frac{l}{i}$.

$$m_{opt} = \sum_{i=1}^{h_{opt}} |L(i)| \leq \sum_{i=1}^{h_{opt}} \frac{l}{i} = l \cdot \sum_{1}^{h_{opt}} \frac{1}{i} \leq \Theta(\log h_{opt}) \cdot l \leq \Theta(\log m_{opt}) \cdot l \qquad \square$$

This bound is asymptotically tight. The family of trees defined in [7] gives $\Theta(\log m_{opt})$ approximation factor.

8 Max-NLS Problem for Restricted Linear Graphs

In this section we deal with Max-NLS problem, for restricted linear graphs. We will show for this problem an $\mathcal{O}(1)$-approximation algorithm running in $\mathcal{O}(kn)$ time. For this section, let $S = (a_1, \ldots, a_n) \in \Sigma^n$ be a given sequence of characters, and $\#_p(X)$ denote the number of characters p in sequence X.

For $p, q \in \Sigma$, by $\mathsf{MNL}_{(p,q)}(S)$ we will denote MNL of a subgraph of $\mathbf{G}_\xi(S)$ containing only such edges (i, j), that $a_i = p$ and $a_j = q$. In other words, we focus only on edges whose left endpoints are at characters p and right endpoints are at characters q. By $\mathsf{MNL}_\xi(S)$ we will denote the maximum nested loop among all $\mathsf{MNL}_{(p,q)}(S)$ for $(p, q) \in \xi$.

Theorem 2. *$\mathsf{MNL}_\xi(S)$ can be computed in $\mathcal{O}(n)$ time.*

Proof. The following two arrays: $l[i, p] = \#_p(a_1, \ldots, a_i)$, $r[i, p] = \#_p(a_i, \ldots, a_n)$ (for $1 \leq i \leq n$, $p \in \Sigma$) can be computed in $\mathcal{O}(n)$ time. The $|\mathbf{E}(\mathsf{MNL}_\xi(S))|$ can be obtained using the following formula:

$$|\mathbf{E}(\mathsf{MNL}_\xi(S))| = \max_{i \in 1, \ldots, n-1,\ (p,q) \in \xi} \min(l[i, p], r[i + 1, q]).$$

Since the size of ξ is $\mathcal{O}(1)$, the total running time of this algorithm is $\mathcal{O}(n)$. $\square$

Lemma 1. *Let $(p, q) \in \xi$. If $\#_p(S) \geq l$ and $\#_q(S) \geq l$ then $|\mathbf{E}(\mathsf{MNL}_\xi(S))| \geq \frac{l}{2}$.*

Proof. One can note, that there exists a $1 \leq i \leq n-1$ such that $\#_p(a_1, \ldots, a_i) = \#_q(a_{i+1}, \ldots, a_n) = c$. There are two cases:

- If $c \geq \frac{l}{2}$, then obviously $|\mathbf{E}(\mathsf{MNL}_{(p,q)})| \geq c \geq \frac{l}{2}$.

- If $c < \frac{l}{2}$, then we have $\#_q(a_1, \ldots, a_i) \geq l - c$ and $\#_p(a_{i+1}, \ldots, a_n) \geq l - c$. Hence $|\mathbf{E}(\mathsf{MNL}_{(q,p)})| \geq l - c \geq \frac{l}{2}$. □

Theorem 3. *Let d be the number of such unordered pairs $\{p, q\}$, that $(p, q) \in \xi$. For given k sequences $S_i \in \Sigma^n$, $\mathsf{Max\text{-}NLS}(\mathbf{G}_\xi(S_1), \ldots, \mathbf{G}_\xi(S_k))$ can be approximated using MNL_ξ within factor $\frac{1}{2d}$ and in $\mathcal{O}(kn)$ time complexity.*

Proof. For each S_i we can calculate $\mathsf{MNL}_\xi(S_i)$ in $\mathcal{O}(n)$ time. Then we choose such $1 \leq j \leq k$ for which $|\mathbf{E}(\mathsf{MNL}_\xi(S_j))|$ is minimum and $\mathsf{MNL}_\xi(S_j)$ is our approximation. Clearly $\mathsf{MNL}_\xi(S_j)$ is a common subgraph of $\mathbf{G}_\xi(S_1), \ldots, \mathbf{G}_\xi(S_k)$, and the overall complexity of this algorithm is $\mathcal{O}(kn)$.

Let $S_j = (a_1, \ldots, a_n)$ and let $e = |\mathbf{E}(\mathsf{Max\text{-}NLS}(\mathbf{G}_\xi(S_j)))|$. It is enough to show, that $|\mathbf{E}(\mathsf{MNL}_\xi(S_j))| \geq \frac{e}{2d}$. We label each edge (r, s) in $\mathsf{Max\text{-}NLS}(\mathbf{G}_\xi(S_j))$ with an unordered pair $\{a_r, a_s\}$. Since there are at most d different labels on e edges, there exists such a pair of characters $(p, q) \in \xi$, that there are at least $\frac{e}{d}$ edges labeled with $\{p, q\}$. Hence $\#_p(S_j) \geq \frac{e}{d}$ and $\#_q(S_j) \geq \frac{e}{d}$. Using lemma 1, we have that $|\mathbf{E}(\mathsf{MNL}_\xi(S_j))| \geq \frac{e}{2d}$. □

It also proves that the approach presented in [7], where general linear graphs were used to model structural alignment of ncRNA, is too abstract. The set of nucleotides is definitely finite. Taking this into account can lead to faster algorithms or better approximation ratios.

For the ncRNA sequences, from the theorem 3 we can achieve an approximation ratio $\frac{1}{6}$ (since there are three possible kinds of unordered bonds in ξ_{RNA}). However, the lower bound on the approximation ratio can be improved, using properties of relation ξ_{RNA}, to $\frac{1}{4}$.

Proposition 5. *For given k sequences $S_i \in \Sigma^n_{\mathsf{RNA}}$, $\mathsf{Max\text{-}NLS}(\mathbf{G}_{\xi_{\mathsf{RNA}}}(S_1), \ldots, \mathbf{G}_{\xi_{\mathsf{RNA}}}(S_k))$ can be approximated using MNL within factor $\frac{1}{4}$.*

9 Conclusion

In this paper, we have investigated the problem of structural alignment of multiple ncRNA sequences. The problem have been modeled in the graph theoretical framework, where ncRNA correspond to linear graphs and their structural alignments correspond to nested linear subgraphs. We described a polynomial-time algorithm for fixed k, and gave improved approximability results, $\mathcal{O}(\log n)$-approximation algorithm running in $\mathcal{O}(kn^2)$ time for arbitrary linear graphs and $\mathcal{O}(1)$-approximation of Max-NLS problem running in $\mathcal{O}(kn)$ time for restricted linear graphs. In particular, for ncRNA derived linear graphs, an $\frac{1}{4}$-approximation is presented.

In conclusion, we mention some interesting directions for future works. First, the approximation aspect of the Max-NLS problem has to be improved. In particular, is the Max-NLS problem approximable to within some constant? Second, we do believe that investigating generalizations of the Max-NLS problem involving more complex structures is of particular importance from both a theoretical and a practical computational biology point of view (bi-secondary structures seem to be an interesting starting point).

References

1. M.I. Abouelhoda and E. Ohlebusch:. Multiple genome alignment: Chaining algorithms revisited. In *Proc. of the 14th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 2676 of *LNCS*, pages 1–16, 2003.
2. V. Bafna, S. Muthukrishnan, and R. Ravi. Computing similarity between RNA strings. Number 937, pages 1–16. Springer-Verlag, Berlin, 1995.
3. Vineet Bafna, Haixu Tang, and Shaojie Zhang. Consensus folding of unaligned rna sequences revisited. *RECOMB*, 2005.
4. Sergey Bereg and Binhai Zhu. RNA multiple structural alignment with longest common subsequences. *COCOON*, 3595:32–41, 2005.
5. H.L. Bodlaender, T. Kloks, D. Kratsch, and H. Müller. Treewidth and minimum fill-in on d-trapezoid graphs. *Journal of Graph Algorithms and Applications*, 2(5): 1–23, 1998.
6. I. Dagan, M.C. Golumbic, and R.Y. Pinter. Trapezoid graphs and their coloring. *Discrete Applied Mathematics*, 21:35–46, 1988.
7. E. Davydov and S. Batzoglou. A computational model for RNA multiple structural alignment. In *Proc. of the 15th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS, pages 254–269. Springer-Verlag, 2004.
8. S. Felsner, R. Müller, and L. Wernisch. Trapezoid graphs and generalizations: Geometry and algorithms. *Discrete Applied Math.*, 74:13–32, 1997.
9. C. Flotow. on powers of m-trapezoid graphs. *Discrete Applied Mathematics*, 63(2):187–192, 1995.
10. M.C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs.* Academic Press, New York, 1980.
11. J. Gramm, J. Guo, and R. Niedermeier. Pattern matching for arc-annotated sequences. In *Proc. of the the 22nd Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, volume 2556 of *LNCS*, pages 182–193, 2002.
12. Ian Holmes and Gerald M. Rubin. Pairwise RNA structure comparison with stochastic context-free grammars. In *Pacific Symposium on Biocomputing*, pages 163–174, 2002.
13. G. Lin, Z-Z. Chen, T. Jiang, and J. Wen. The longest common subsequence problem for sequences with nested arc annotations. *Journal of Computer and System Sciences*, 65(3):465–480, 2002. Special issue on computational biology.
14. Jianghui Liu, Jason TL Wang, Jun Hu, and Bin Tian. A method for aligning RNA secondary structures and its application to RNA motif detection. *BMC Bioinformatics*, 6(89), 2005.
15. A. Lozano and G. Valiente. On the maximum common embedded subtree problem for ordered trees. In C. Iliopoulos and T Lecroq, editors, *String Algorithmics*, chapter 7. King's College London Publications, 2004.
16. R. Nussinov, G. Pieczenik, J.R. Griggs, and D.J. Kleitman. Algorithms for loop matching. *SIAM Journal of Applied Mathematics*, 35(1):68–82, 1978.
17. S. Vialette. On the computational complexity of 2-interval pattern matching. *Theoretical Computer Science*, 312(2-3):223–249, 2004.
18. M.S. Waterman. *Introduction to computational biology - Maps, sequences and genomes.* Chapman and Hall, London, 1995.
19. K. Zhang and D. Shacha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM journal of computing*, 18(6):1245–1262, 1989.

Finding Common RNA Pseudoknot Structures in Polynomial Time

Patricia A. Evans

Faculty of Computer Science
University of New Brunswick
Fredericton, NB, Canada
pevans@unb.ca

Abstract. This paper presents the first polynomial time algorithm for finding common RNA substructures that include pseudoknots and similar structures. While a more general problem is known to be NP-hard, this algorithm exploits special features of RNA structures to match RNA bonds correctly in polynomial time. Although the theoretical upper bound on the algorithm's time and space usage is high, the data-driven nature of its computation enables it to avoid computing unnecessary cases, dramatically reducing the actual running time. The algorithm works well in practice, and has been tested on sample RNA structures that include pseudoknots and pseudoknot-like tertiary structures.

1 Introduction

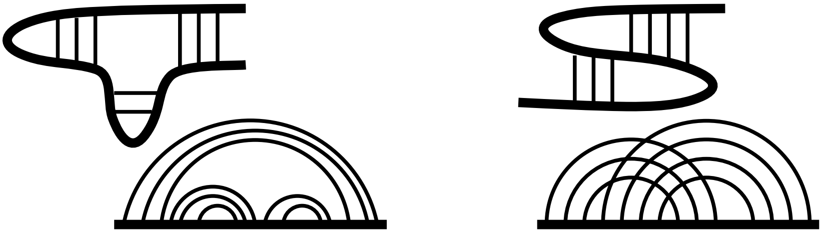
Features in biomolecules are frequently discovered by comparing sequences. Ribonucleic acid (RNA) strands, however, have structures formed largely from bonds between pairs of bases from the sequences, and can have common structures that affect their function but do not show significant sequence similarity. In order to find common features and relationships, we need to compare RNA molecules by their structure and find common substructures.

We can view RNA structures at their most basic as a set of pairs of ordered sequence positions, using arcs between sequence positions to represent those bases that are bonded. There are some known fundamental structures in RNA, such as stems formed from sets of adjacent bonds, and these can group bonds together. However, we want to be able to look for arbitrary common substructures, and so this work will operate at the bond level. This approach is therefore complementary to the high level based structural comparison presented in [1].

A bond structure X is a substructure of a bond structure Y if the positions of X can be mapped onto positions of Y while preserving both the sequence order and the bonds. Finding common substructures for two sets of ordered pairs is NP-complete in general, and remains NP-complete even if restricted to RNA pair-bonds where each position can participate in at most one pair [6, 13]. Solutions for these problems must thus use nonpolynomial algorithms, approximation techniques, or address restricted types of structures.

RNA structures are usually divided into two types by the permitted relationships between pairs of bonds (h, l) and (i, j) with respect to the sequence order,

M. Lewenstein and G. Valiente (Eds.): CPM 2006, LNCS 4009, pp. 223–232, 2006.



a) nested structure

b) pseudoknot

Fig. 1. Knot-free and Pseudoknot structures, both folded and unfolded

.

as illustrated in Figure 1. Knot-free bond structures restrict the pairwise bond relationships to those that either nest ($h < i < j < l$) or simply occur in sequence ($h < l < i < j$). Bonds that cross with respect to the sequence order ($h < i < l < j$), on the other hand, form *pseudoknot* substructures, and are more difficult to predict and to compare, especially when there are additional substructures contained within a complex pseudoknot. Previous polynomial-time algorithms for finding common substructures [2, 12] were restricted to structures without pseudoknots, and thus could not handle many known RNA structures.

This work investigates specific restrictions that fit most known RNA structures, including those with pseudoknots, and presents an algorithm that finds the maximum common substructure for two RNA structures in polynomial time. While the theoretical worst-case running time and space is large, a computation-reducing technique is used to make the algorithm feasible and effective.

2 Background

In our search for common structural features, we can initially restrict our objective to finding, for two given structures, a common substructure that has the largest number of arcs that link its positions.

Our basic problem Maximum Common Ordered Substructure is:

Input: structures S_1 and S_2,
  where S_1 is the arc structure for a sequence of n_1 positions
    and S_2 is the arc structure for a sequence of n_2 positions,
    so $S_1 \subseteq \{1..n_1\} \times \{1..n_1\}$ and $S_2 \subseteq \{1..n_2\} \times \{1..n_2\}$.

Output: substructure S_c, maximizing $|S_c|$
  where $S_c \subseteq \{1..n_c\} \times \{1..n_c\}$ for some positive integer n_c, such that:
    $\exists$ one-to-one functions $f_1 : \{1..n_c\} \rightarrow \{1..n_1\}$ and $f_2 : \{1..n_c\} \rightarrow \{1..n_2\}$
      where $\forall i, j \in \{1..n_c\}$, $i < j$ if and only if $f_1(i) < f_1(j)$ and $f_2(i) < f_2(j)$
        and if $(i, j) \in S_c$, then $(f_1(i), f_1(j)) \in S_1$ and $(f_2(i), f_2(j)) \in S_2$

If there are no additional restrictions on arcs, this problem is the same as the general contact map overlap problem used to compare protein structures

[8]. It is also similar to the problem of finding the longest arc-preserving common subsequence (LAPCS) [6]; though it differs in maximizing arcs rather than positions matched, and in allowing arcs to be broken, Maximum Common Ordered Substructure does inherit some of the clique-based hardness results from LAPCS.

A variety of results for these related contact map overlap [8, 9], RNA [2, 6, 13], and 2-interval matching problems [11] delineate the known feasible and infeasible cases for Maximum Common Ordered Substructure. The general problem as given above without any further restrictions is NP-complete [6, 8].

Restrictions on the structures are defined based on the allowed relationships between two arcs, (h, l) and (i, j) with $h \leq i$, from the same structure. We consider the arcs to be directed, with $h < l$ and $i < j$. Arcs can:

- *share an endpoint*: where $h = i$, $l = i$, or $l = j$.
- *precede* : where $l < i$, so the interval of one arc occurs before the other
- *nest* : where $h < i < j < l$, so the interval of one arc is contained entirely within the other
- *cross* : where $h < i < l < j$, so the intervals overlap

Polynomial time algorithms exist for finding known substructures in restricted models that do not permit nested arcs [9] or do not permit crossing arcs [11].

A *pair-bond* structure would allow preceding, nesting, and crossing, but not endpoint sharing, since RNA bases generally participate in at most one bond each. A *knot-free pair-bond* structure would further prohibit crossing. Results for these restrictions show that the maximum common ordered substructure problem for two general pair-bond structures is NP-hard [6, 13], and for two knot-free pair-bond structures can be solved in $O(n^4)$ time and space [2].

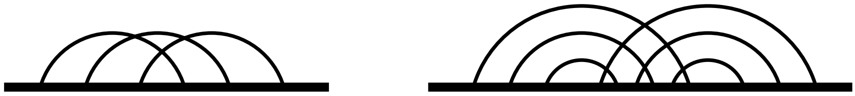
The hardness results for pair-bond structures, however, use constructions with arbitrarily crossing arcs that do not resemble actual RNA structures. Almost all RNA structures can be divided, or *2-coloured*, into two layers of arcs such that each layer's arcs only cross arcs from the other layer [8, 4]. Work on contact maps has shown that if each input structure is the union of two such layers, the problem is still NP-hard [8]. However, this variation also allows endpoint sharing between arcs from different layers, which would not occur for RNA. If the input is restricted to 2-colourable pair-bond structures, prohibiting all endpoint sharing, then the problem's status is still unknown.

3 Becoming Specific to RNA Structures

As mentioned in Section 2, hardness results for the more general variants of the maximum common ordered substructure problem have constructions with little resemblance to actual RNA structures. We therefore need to consider additional restrictions that still allow pseudoknots but are consistent with the characteristics of known RNA structures.

First, we will restrict our structures to 2-colourable pair-bond structures, where arcs must have different colours if they cross. This restriction disallows

3-knots, substructures that have 3 mutually crossing arcs, as shown in Figure 2(a). This restriction enables us to avoid the hardness constructions of [6, 13] which depend on arbitrary k-knots that do not occur in practice [4].



a) a three-knot

b) interleaved left-right endpoints

Fig. 2. Excluded cases

Furthermore, while we are operating at the bond level, bonds occur in groups; interleaving the right endpoints of one set of bonds with the left endpoints of another set does not occur in examined structures, nor would it be biochemically stable. In order to prevent overlap between the segments of RNA containing sets of arcs, we also disallow these interleaved endpoints as shown in Figure 2(b).

All structures consistent with these restrictions can be decomposed into, and assembled from, sets of segments and linked segment pairs (LSPs). An independent segment of an RNA structure is a set of consecutive positions such that no position is linked with an arc to any position outside the segment. Structures without pseudoknots can be decomposed into independent segments. Pseudoknot structures, however, need to allow arcs to link out of segments. The restrictions we have adopted as to how they can cross limit these arcs to coherent groups that link a segment to specific other segments to the left and right. Linking a non-independent segment to the segment that contains the remaining other endpoints of its arcs provides sufficient context to compute the optimal result of the linked pair. LSPs and segments can then be assembled into independent segments and larger LSPs, as illustrated in Figure 3.

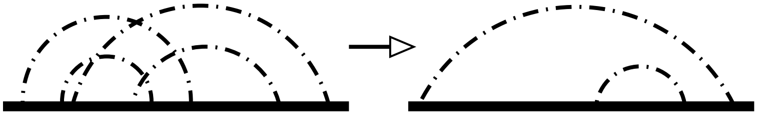


Fig. 3. Merging Linked Segment Pairs

4 Breaking Down Structures

4.1 Decomposition Overview

In the work of Bafna *et al.* on matching knot-free RNA structures, mapping an arc between two structures decomposes the remaining parts of each structure into two independent segments [2]. For pesudoknotted structures, the segments remaining could still have arcs linking them, and thus could not be considered independently; we need to consider them together as a linked segment pair (LSP). Mapping an arc that is part of an LSP breaks the remaining substructures down further, into one or two LSPs and potentially an additional segment. Note that unlike techniques that require the structure to have been previously parsed [1], these breakdowns are computed along with the optimal mapping.

Breaking a structure into its substructures only occurs when one of its arcs is mapped to a similarly-situated arc in the other structure. We can therefore visualize the common decomposition of the two structures as if it was a single structure. The following subsections examine what decomposition cases could occur, and give the recurrences that show how the independent segments and LSPs decompose into smaller structures. Note that that gaps between the two segments of an LSP will have one or more associated arcs between the gap and the region after the LSP. The use of LSP cases is restricted to situations that have at least one arc linking the segments.

4.2 Matching Segments

For matching segment (i_1, j_1) from S_1 to segment (i_2, j_2) from S_2, we can follow the recurrences from [2], and maximize the results of:

s1: value of matching segment $(i_1, j_1 - 1)$ to (i_2, j_2)
s2: value of matching segment (i_1, j_1) to $(i_2, j_2 - 1)$
s3: if j_1 links to k_1 and j_2 links to k_2:
1+ (value of matching segment $(i_1, k_1 - 1)$ to $(i_2, k_2 - 1)$) +
(value of matching segment $(k_1 + 1, j_1 - 1)$ to $(k_2 + 1, j_2 - 1)$)

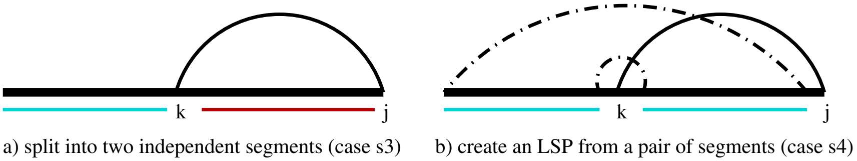
To match pseudoknots, we add an additional case; if j_1 links to k_1, j_2 links to k_2, and in both cases the arc (k, j) is crossed by arcs linking segments $(i, k - 1)$ and $(k + 1, j - 1)$:

s4: 1+ (value of matching LSP $(i_1, k_1 - 1, k_1 + 1, j_1 - 1)$ to $(i_2, k_2 - 1, k_2 + 1, j_2 - 1)$)

4.3 Matching Linked Segment Pairs

To match LSP (h_1, l_1, i_1, j_1) to LSP (h_2, l_2, i_2, j_2), we maximize the results of the following cases, based on the different possibilities for the current last position in each LSP.

Cases for Unmatched and Unlinked Positions
The first cases are analogous to the initial segment cases, where terminal positions are not matched.



a) split into two independent segments (case s3)
b) create an LSP from a pair of segments (case s4)

Fig. 4. Cases for matching segments

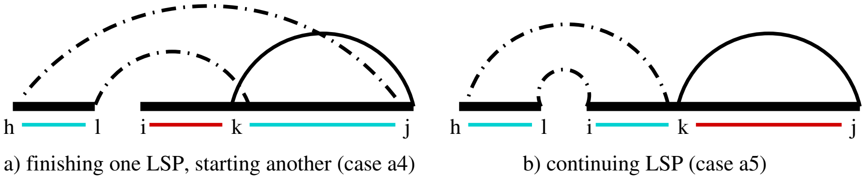
a1: value of matching LSP $(h_1, l_1, i_1, j_1 - 1)$ to LSP (h_2, l_2, i_2, j_2)
a2: value of matching LSP (h_1, l_1, i_1, j_1) to LSP $(h_2, l_2, i_2, j_2 - 1)$
a3: (value of matching segment (h_1, l_1) to (h_2, l_2)) +
(value of matching segment (i_1, j_1) to(i_2, j_2))

This third case, if used together with the last segment case, enables a new LSP to be made from the right segments of corresponding LSPs.

Cases Mapping Arcs That Are Within the Right LSP Segment

If there are matching arcs that link j_1 to k_1 and j_2 to k_2, and $i_1 \leq k_1 < j_1$, $i_2 \leq k_2 < j_2$ (so both arcs are within the right segment of the LSP):

a4: 1+ (value of matching LSP $(h_1, l_1, k_1 + 1, j_1 - 1)$ to $(h_2, l_2, k_2 + 1, j_2 - 1)$) +
(value of matching segment $(i_1, k_1 - 1)$ to $(i_2, k_2 - 1)$)
a5: 1+ (value of matching LSP $(h_1, l_1, i_1, k_1 - 1)$ to $(h_2, l_2, i_2, k_2 - 1)$) +
(value of matching segment $(k_1 + 1, j_1 - 1)$ to $(k_2 + 1, j_2 - 1)$)



a) finishing one LSP, starting another (case a4)

b) continuing LSP (case a5)

Fig. 5. Cases for matching LSPs if arc is within right side

Cases Mapping Arcs That Cross the Gap to the Left LSP Segment

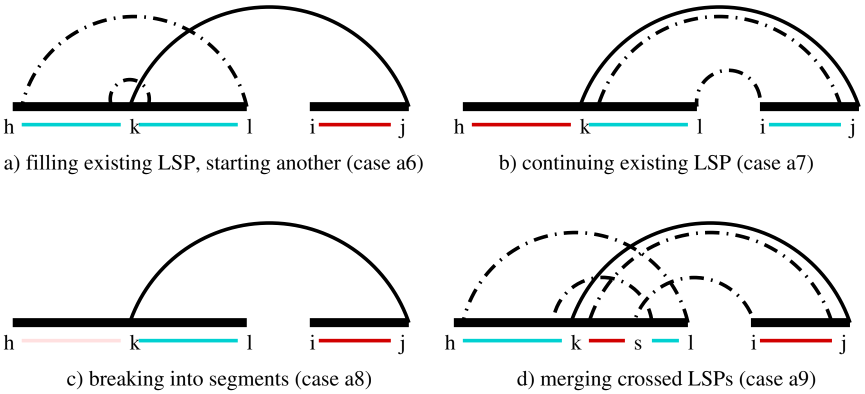
If there are matching arcs that link j_1 to k_1 and j_2 to k_2, and $h_1 \leq k_1 \leq l_1$, $h_2 \leq k_2 \leq l_2$ (so both arcs cross to the left segment of the LSP):

a6: 1+ (value of matching LSP $(h_1, k_1 - 1, k_1 + 1, l_1)$ to $(h_2, k_2 - 1, k_2 + 1, l_2)$) +
(value of matching segment $(i_1, j_1 - 1)$ to $(i_2, j_2 - 1)$)
a7: 1+ (value of matching LSP $(k_1 + 1, l_1, i_1, j_1 - 1)$ to $(k_2 + 1, l_2, i_2, j_2 - 1)$) +
(value of matching segment $(h_1, k_1 - 1)$ to $(h_2, k_2 - 1)$)
a8: 1+ (value of matching segment $(h_1, k_1 - 1)$ to $(h_2, k_2 - 1)$) +
(value of matching segment $(k_1 + 1, l_1)$ to $(k_2 + 1, l_2)$) +
(value of matching segment $(i_1, j_1 - 1)$ to $(i_2, j_2 - 1)$)

We do not attempt to match LSPs $(h, k-1, i, j-1)$ since it would form a three-knot with the arc linking k to j and the arc that produced the gap between l and i. Any of these cases matching LSPs should be restricted to situations where there are arcs present in the LSP.

Additionally, if both $(h, k-1, k+1, l)$ and $(k+1, l, i, j-1)$ are potential LSPs in both structures, then we apply the following crossed-LSPs case:

a9: 1+ $\max_{k_1 < s_1 < l_1,\ k_2 < s_2 < l_2}$ [(value of matching LSPs $(h_1, k_1 - 1, s_1 + 1, l_1)$ to $(h_2, k_2 - 1, s_2 + 1, l_2)$) +
(value of matching LSPs $(k_1 + 1, s_1, i_1, j_1 - 1)$ to $(k_2 + 1, s_2, i_2, j_2 - 1)$)]



a) filling existing LSP, starting another (case a6)

b) continuing existing LSP (case a7)

c) breaking into segments (case a8)

d) merging crossed LSPs (case a9)

Fig. 6. Cases for matching LSPs if arc crosses to left side

Some of these cases do allow for some instances of three-knots or interleaved endpoints; however, they do not allow arbitrary instances of them.

5 Implementation

These recurrences and the likely overlap between them suggest the use of dynamic programming to store the intermediate computation results for use and reuse, as is done for the algorithm for knot-free structures [2].

An iterative dynamic approach has significant problems. The mutual dependence of the independent segment and LSP recurrences poses difficulties in determining a suitable computation order. The tables will be extremely large, with one 4-dimensional table for the independent segment recurrence and one 8-dimensional table for the LSP recurrence; each of these is far too large to allocate in its entirety for sizes for many RNA comparisons, eg. $n = 1000$.

However, many potential cases will not be consistent with the data. Many intervals are not legitimate segments in a structure, many pairs of intervals are not valid occurring LSPs, and many of the valid substructures from each input structure cannot be matched to each other. We need to allow the input data to drive the computation, restricting the cases computed and the space allocated based on the segments and LSPs consistent with the data.

For these reasons, the recurrences have been implemented as a recursive algorithm using memoization. The tables are progressively allocated dimension by dimension to avoid allocating space for entire hyperplanes of the table if there are no results consistent with the data that will be stored in the hyperplane.

In the following algorithm, note that the $link()$ function encodes arcs, so that $(i, link(i))$ will represent the arc between i and its unique partner $link(i)$.

Each table has the first level allocated, and the recursion is started by calling $doublepair(0, n_1 - 1, 0, n_2 - 1)$ where n_1 is the number of positions in RNA structure S_1, and n_2 is the number of positions in RNA structure S_2.

```
$doublepair(i_1, j_1, i_2, j_2)$ :
    if table4$[i_1, j_1, i_2, j_2]$ exists and contains a value
        return that value
    else
        calculate the maximum of the applicable segment cases (s1..s4).
        for each level $(j_1, i_2, j_2)$ of the table, in order
            if the level does not exist
                allocate this level from previous index to $link(i_1 - 1)$
                                    or $link(i_2 - 1)$ as applicable
            save maximum value in table4$[i_1, j_1, i_2, j_2]$

$quadpair(h_1, l_1, i_1, j_1, h_2, l_2, i_2, j_2)$ :
    if table8$[h_1, l_1, i_1, j_1, h_2, l_2, i_2, j_2]$ exists and contains a value
        return that value
    else
        calculate the maximum of the applicable LSP cases (a1..a9).
        for each level $(l_1, i_1, j_1, h_2, l_2, i_2, j_2)$ of the table, in order
            if the level does not exist
                allocate this level from previous index to $link(h_1 - 1)$
                                    or $link(h_2 - 1)$ as applicable
            save maximum value in table8$[h_1, l_1, i_1, j_1, h_2, l_2, i_2, j_2]$
```

The rest of the allocation is done progressively, layer by layer. The $link(i-1)$ and $link(h-1)$ values are used to reduce the size of the hyperplane allocated to no more than needed. The initial position (i for segments, h for LSPs) can only increase when its previous value is the initial endpoint of a mapped arc. For independent segments, the final endpoint of this arc will be after the end of the segment, so we know that all segments that start at position i will end before position $link(i-1)$, and can limit the size of our table allocation accordingly. A similar situation is true for LSPs; all possible LSPs that start at h must end before $link(h-1)$, and this constrains the values for the indices l, i, and j.

In the worst case, if the algorithm needed to compute every index combination, it would require $O(n^8)$ space. Since case a9 is quadratic, this would lead to a worst-case running time $\in O(n^{10})$. However, the measures discussed above allows the data characteristics to greatly reduce the index combinations computed. The use of case a9 is also very limited in any RNA structure.

Testing reveals that this data-driven approach does drastically reduce the space and time needed to feasible amounts.

6 Testing Results

To determine the feasibility and the effectiveness of this algorithm, it was implemented in C and tested on three types of RNA data: 16S ribosomal RNA (*H. sapiens* and *D. melanogaster*)[5]; segments of mosaic viral RNA (tobacco

RNA type	# of bonds mapped	locations calculated	recursive calls
16S rRNA	335	4.9 $\times 10^8$	1.3 $\times 10^9$
viral RNA	52	6.2 $\times 10^6$	2.2 $\times 10^7$
RNase P	109	1.8 $\times 10^8$	6.7 $\times 10^8$

Fig. 7. Computations for RNA structure test cases

mosaic[7], turnip yellow mosaic and chimera [10]); and Ribonuclease P (*H. influenza* and *B. pertussis*)[3]. All types contain multiple pseudoknots and have other structures contained within pseudoknots. For all cases, the algorithm was able to find the correct common substructures including all pseudoknots.

These experiments showed that structures of up to 400 arcs could be compared using this algorithm in 4Gb of space. The proportion of the worst-case space used by the algorithm's execution on these cases was approximately 10^{-14}, a very significant reduction due to the data-driven and space-limiting approach. The amount of space used, however, was not completely related to the number of arcs, due to space usage being data-driven; some structures, particularly those of RNase P, were more complex and required more calculations than larger simpler structures.

7 Conclusion

This algorithm finds common RNA substructures that can include pseudoknots and pseudoknot type structures, and shows that this problem can be solved in polynomial time. Although the theoretical worst case resource usage is high, this is reduced severely by memoization and careful memory management, and test results show that this algorithm finds useful patterns in RNA structures. Using a compressed RNA bond structure will increase the usability of this algorithm.

While this algorithm shows that the maximum common substructure for two RNA structures can be found in polynomial time for most RNA structures, some open problems still remain. This algorithm and the NP-completeness of matching 2-colourable structures leaves open the problem of finding maximum common substructures for two 2-colourable pair-bond structures. Also, there are some rare exceptions to the restrictions adopted for this algorithm, including rare 3-way bonds, that should be investigated.

As this algorithm is simply maximizing the number of common bonds, there also need to be extensions of this work to adapt it more thoroughly to RNA structures, using weights and structural information to produce common substructures that will be most relevant to RNA.

References

1. J. Allali and M.-F. Sagot. A new distance for high level RNA secondary structure comparison. *IEEE/ACM Transactions on Computational Biology and Bioinformatics* **2(1)** (2005), 3-14.
2. V. Bafna, S. Muthukrishnan, and R. Ravi. Computing similarity between RNA strings. *DIMACS Technical Report* **96-30** (1996).

3. J. Brown. The Ribonuclease P Database. *Nucleic Acids Research* **27** (1999), 314.
4. M. Burkard, D. Turner, and I. Tinoco. Schematic diagrams of secondary and tertiary structure elements. In: R. Gesteland, T. Cech, and J. Atkins, editors, *The RNA World (Second Edition).* Cold Spring Harbor Laboratory Press: Cold Spring Harbor, New York (1999), 681-685.
5. J. Cannone et al. The comparative RNA web (CRW) site: an online database of comparative sequence and structure information for ribosomal, intron, and other RNAs. *BioMed Central Bioinformatics* **3** (2002), 15.
6. P. Evans. Finding common subsequences with arcs and pseudoknots. *Proceedings of Combinatorial Pattern Matching '99*, Springer-Verlag **LNCS 1645** (1999), 270-280.
7. B. Felden, C. Florentz, R. Giegé, and E. Westhof. A central pseudoknotted three-way junction imposes tRNA-like mimicry and the orientation of three 5' upstream pseudoknots in the 3' terminus of tobacco mosaic virus RNA. *RNA* **2** (1996), 201-212.
8. D. Goldman, S. Istrail, and C. Papadimitriou. Algorithmic aspects of protein similarity. *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS 99)* (1999).
9. J. Gramm. A polynomial-time algorithm for the matching of crossing contact map patterns. *IEEE/ACM Transactions on Computational Biology and Bioinformatics* **1(4)** (2004), 171-180.
10. J. Skuzeski, C. Bozarth, and T. Dreher. The turnip yellow mosaic virus tRNA-like structure cannot be replaced by generic tRNA-like elements or by heterologuous 3' untranslated regions known to enhance mRNA expression and stability. *Journal of Virology* **70** (1996), 2107-2115.
11. S. Vialette. On the computational complexity of 2-interval pattern matching problems. *Theoretical Computer Science* **312(2-3)** (2004), 223-249.
12. K. Zhang. Computing similarity between RNA secondary structures. *Proceedings of IEEE International Joint Symposia on Intelligence and Systems* (1998), 126-132.
13. K. Zhang, L. Wang, and B. Ma. Computing similarity between RNA structures. *Proceedings of Combinatorial Pattern Matching '99*, Springer-Verlag **LNCS 1645** (1999), 281-293.

A Compact Mathematical Programming Formulation for DNA Motif Finding

Carl Kingsford[1], Elena Zaslavsky[2], and Mona Singh[2]

[1] Center for Bioinformatics & Computational Biology, University of Maryland,
College Park, MD
carlk@umiacs.umd.edu

[2] Department of Computer Science and Lewis-Sigler Institute for Integrative
Genomics, Princeton University, Princeton, NJ
{elenaz, msingh}@cs.princeton.edu

Abstract. In the *motif finding problem* one seeks a set of mutually similar subsequences within a collection of biological sequences. This is an important and widely-studied problem, as such shared motifs in DNA often correspond to regulatory elements. We study a combinatorial framework where the goal is to find subsequences of a given length such that the sum of their pairwise distances is minimized. We describe a novel integer linear program for the problem, which uses the fact that distances between subsequences come from a limited set of possibilities. We show how to tighten its linear programming relaxation by adding an exponential set of constraints and give an efficient separation algorithm that can find violated constraints, thereby showing that the tightened linear program can still be solved in polynomial time. We apply our approach to find optimal solutions for the motif finding problem and show that it is effective in practice in uncovering known transcription factor binding sites.

1 Introduction

A central challenge in post-genomic biology is to reconstruct the regulatory network of an organism. A key step in this process is the discovery of regulatory elements. A common approach finds novel sites by searching for a set of mutually similar subsequences within DNA sequences. These subsequences, when aligned, form *motifs*, and are putative binding sites for a shared transcription factor. The effectiveness of identifying regulatory elements in this manner has been demonstrated when considering sets of sequences identified via shared co-expression, orthology and genome-wide location analysis (e.g., [19, 8, 11]).

Numerous problem formalizations and computational approaches have been developed for motif finding (see [21], and references therein). Probabilistic approaches typically maximize the information content of the chosen motif instances (e.g., [10, 3, 7]). Combinatorial methods either enumerate all allowed motifs or attempt to optimize some measure based on sequence similarity (e.g., [13, 12]). Here, we take a combinatorial approach and model the motif finding problem as that of

M. Lewenstein and G. Valiente (Eds.): CPM 2006, LNCS 4009, pp. 233–245, 2006.

finding the gapless local multiple sequence alignment of fixed length that minimizes a sum-of-pairs (SP) distance measure. Such a formulation provides a reasonable scheme for assessing motif conservation [15, 18]. The problem is equivalent to that of finding a minimum weight clique of size p in a p-partite graph (e.g., [16]). For general notions of distance, this problem is NP-hard to approximate within any reasonable factor [4]. The problem and its variants remain NP-hard in the context of biological sequences [1, 22], though in the motif finding setting, where the distances obey the triangle inequality, constant-factor approximation algorithms exist [2]. Nevertheless, the ability to find the optimal solution in practice is preferable.

We introduce and extensively explore a mathematical programming approach to motif finding. We propose a novel integer linear programming (ILP) formulation of the motif finding problem that uses the discrete nature of the distance metric imposed on pairs of subsequences. Considering its linear programming (LP) relaxation, we show that while it is weaker than an alternative LP formulation for motif finding [23], an exponentially-sized class of constraints can be added to make the two formulations equivalent. We then show that it is not necessary to explicitly add all these constraints by giving a separation algorithm, based on identifying minimum cuts in a graph constructed to model the ILP, that identifies violated constraints and thus permits a polynomial-time solution to the tightened LP.

We test the effectiveness of our approach in identifying DNA binding sites of *E. coli* transcription factors. We demonstrate that our new ILP framework is able to find optimal solutions often an order of magnitude faster than the previously known mathematical programming formulation, and that its performance in identifying motifs is competitive with a widely-used probabilistic Gibbs-sampling approach [20]. Finally, we note that in practice the LP relaxations often have integral optimal solutions, making solving the LP sufficient in many cases for solving the original ILP.

2 Formal Problem Specification

We are given p sequences, which are assumed without loss of generality to each have length N', and a motif length ℓ. In our formulation, the goal is to find a subsequence s_i of length l in each sequence i so as to minimize the sum of the pairwise distances between the subsequences. Here, the distance between two substrings s_i and s_j is computed as the Hamming distance between them ($\mathbf{HD}(s_i, s_j)$), and thus our goal is to choose the substrings such that $\sum_{i<j} \mathbf{HD}(s_i, s_j)$ is minimized.

The problem can be reformulated in graph-theoretic terms. For p input sequences, we define a complete, weighted p-partite graph, with a part V_i for each sequence. In V_i, there is a node for every window of length ℓ in sequence i. Thus there are $N := N' - \ell + 1$ nodes in each V_i, and the vertex set $V = V_1 \cup \cdots \cup V_p$ has size Np. For every pair u and v in different parts there is an edge $(u, v) \in E$. Letting $\mathbf{seq}(u)$ denote the subsequence corresponding to node u, the weight w_{uv}

on edge (u, v) equals $\mathbf{HD}(\mathbf{seq}(u), \mathbf{seq}(v))$. The goal is to choose a node from each part so as to minimize the weight of the induced subgraph.

3 Integer Programming Formulations

Original integer linear programming formulation. We first give the integer linear programming formulation presented in [23] for solving the motif finding problem. In this ILP formulation, there is a variable X_u for each node u in the graph described above. The variable X_u is set to 1 if node u is chosen, and 0 otherwise. Additionally, there is one variable X_{uv} for each edge in the graph (X_{uv} is the same as X_{vu}). These edge variables are set to 1 if both end points of the edge are chosen. In the integer program, all variables are constrained to take values from $\{0, 1\}$. The following ILP is easily seen to model the above graph problem:

$$\begin{array}{lll} \text{Minimize} & \sum_{\{u,v\} \in E} w_{uv} \cdot X_{uv} & \\ \text{subject to} & & \\ \quad \sum_{u \in V_i} X_u = 1 & \text{for } i = 1, \ldots, p & \\ \quad \sum_{u \in V_i} X_{uv} = X_v & \text{for } i = 1, \ldots, p \text{ and } v \in V \setminus V_i & \\ \quad X_u, X_{uv} \in \{0, 1\} & & \end{array} \tag{IP1}$$

The first set of constraints ensures that one node is chosen from each part, and the second set requires that an edge is chosen if its end points are. This ILP is the same as the ILP formulation for side-chain positioning presented in [9].

More compact integer linear program. We now introduce an alternative ILP that better exploits the structure of the combinatorial problem. In particular, we use the fact that there are typically only a small number of possible pairwise distances. For example, in the case of Hamming distances, edge weights can only take on $\ell + 1$ different values. We can take advantage of the small number of possible weights and the fact that the edge variables of IP1 are only used to ensure that if two nodes u and v are chosen in the optimal solution then w_{uv} is added to the cost of the clique. In our new ILP formulation, we no longer have edge variables X_{uv}. Instead, in addition to the node variables X_u, we have a variable Y_{ujc} for each node u, each position j such that $u \notin V_j$, and each edge weight c. These Y variables model groupings of the edges by cost into *cost bins*, as shown in Fig. 1. The intuition is that Y_{ujc} is 1 if node u and some node $v \in V_j$ are chosen such that $w_{uv} = c$.

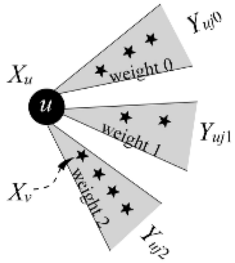


Fig. 1. Schematic of IP2. Adjacent to a node $u \in V_i$ there are at most $|D|$ cost bins for each position $j > i$, each associated with a variable Y_{ujc}. For each cost c there are the nodes $v \in V_j$ for which $w_{uv} = c$ (stars).

Formally, let D be the set of possible edge weights and let $W = \{(u, j, c) : c \in D, u \in V, j \in 1, \ldots p \text{ and } u \notin V_j\}$ be the set of triples over which the Y_{ujc} variables are indexed, and let $\mathrm{part}(u) = i$ if $u \in V_i$. Then the following ILP models the motif-finding graph problem:

$$\begin{array}{lll}
\text{Minimize} & \sum_{(u,j,c)\in W:\mathrm{part}(u)<j} c \cdot Y_{ujc} & \\
\text{subject to} & & \\
\quad \sum_{u\in V_i} X_u = 1 & \text{for } i = 1, \ldots, p & \text{(IP2a)} \\
\quad \sum_{c\in D} Y_{ujc} = X_u & \text{for } j \in 1, \ldots, p \text{ and } u \in V \setminus V_j & \text{(IP2b)} \\
\quad \sum_{v\in V_j:w_{uv}=c} Y_{vic} \geq Y_{ujc} & \text{for } (u, j, c) \in W \text{ s.t. } u \in V_i \text{ and } i < j & \text{(IP2c)} \\
\quad X_u, Y_{ujc} \in \{0, 1\} & & \text{(IP2)}
\end{array}$$

As in IP1, the first set of constraints forces a single node to be chosen in each part. The second set of constraints makes certain that if a node u is chosen, for each j, one of its "adjacent" cost bins must also be chosen (Fig. 1). The third set of constraints ensures that Y_{ujc} can be selected only if some node $v \in V_j$, such that $w_{uv} = c$, is also selected. We discard variables Y_{ujc} if there is no $v \in V_j$ such that $w_{uv} = c$. Fig. 1 gives a schematic drawing of these constraints.

Lemma 1. *IP2 correctly models the sum-of-pairs motif finding problem.*

Proof. For any choice of p-clique $\{u_1, \ldots, u_p\}$ of weight $\gamma = \sum_{i<j} w_{u_i u_j}$, a solution of cost γ to IP2 can be found by taking $X_{u_i} = 1$ for $i = 1 \ldots, p$, and taking $Y_{u_i jc} = 1$ for all $1 \leq j \leq p$ such that $w_{u_i u_j} = c$. This solution is feasible, and between any pair of positions i, j it contributes cost $w_{u_i u_j}$; therefore, the total cost is γ. On the other hand, consider any solution (X, Y) to IP2 of objective value γ. Consider the clique formed by the nodes u such that $X_u = 1$. Between every two positions $i < j$, the constraints (IP2a) and (IP2b) imply that exactly one Y_{ujc} and one $Y_{vic'}$ are set to 1 for some $u \in V_i$ and $v \in V_j$ and costs c, c'. Constraint (IP2c) corresponding to (u, j, c) with Y_{ujc} on its right-hand side can only be satisfied if the sum on its left-hand side is 1, which implies $c = c' = w_{uv}$. Thus, a clique of weight γ exists in the motif-finding graph problem. □

Advantages of IP2. In practice, IP2 has many fewer variables than IP1. Letting $d = |D|$, the number of kinds of weights, IP2 has $Np((p - 1)d + 1)$ variables in the case that a Y_{ujc} variable exists for every allowed choice of (u, j, c), while IP1 has $Np(N(p-1)/2+1)$ variables. If $d < N/2$, the second IP will have fewer variables. In general, d is expected to be much smaller than N: while N could reasonably be expected to grow large as longer and longer sequences are considered, d is constrained by the geometry of transcription factor binding and will remain small. Also, in practice, it is likely that many Y_{ujc} variables are removed because $\mathbf{seq}(u)$ does not have matches of every possible weight in each of the other sequences. On the other hand, IP2, will have $O(d)$ times more constraints than IP1, with the number of constraints being $p+Np(p-1)(d/2+1)$ for IP2, and $p+Np(p-1)$ for IP1. However, the decrease in variables of IP2 tends to be more dramatic than the increase in the number of constraints, resulting in faster execution times (see **Computational Results** and Fig. 3).

4 Linear Programming Relaxations

The typical approach to solving an ILP is to solve as a subproblem the linear program relaxation derived from the ILP by dropping the requirement that the variables be in $\{0, 1\}$, and instead requiring only that the variables lie in the continuous range $[0, 1]$. While finding a solution to the ILP is computationally difficult, its relaxed LP can be solved in polynomial-time. If the solution to the relaxed LP is integral, then we have found a solution to the original ILP. Alternatively, if the solution to the LP is fractional, then branch and bound or other techniques can be used to obtain optimal solutions to the ILP.

The LP relaxation of IP1, which we refer to as LP1, is stronger than the LP relaxation of IP2. Because tighter LP relaxations are often more useful subroutines for finding optimal integer solutions, we first present a natural (though exponential) class of constraints that, if added to the LP relaxation of IP2, makes the two formulations equivalent. We refer to this fully constrained relaxation of IP2 as LP2. Later we give a separation algorithm for finding violated constraints, and thereby show that LP2 can still be solved in polynomial-time.

Additional constraints. Focus on a pair of positions i and j. In IP1 the edge variables between V_i and V_j explicitly model the bipartite graph between those two positions. In IP2, however, the bipartite graph is only implicitly modeled by an understanding of which Y variables are compatible to be chosen together. We study this implicit representation by considering the bipartite *compatibility* graph $\mathcal{C}_{ij}$ between two positions i and j. Intuitively, we have a node in this compatibility graph for each Y_{ujc} and Y_{vic}, and there is an edge between the nodes corresponding to Y_{ujc} and Y_{vic} if $w_{uv} = c$. These two Y variables are compatible in that they can both be set to 1 in IP2. More formally, $\mathcal{C}_{ij} = (A_{ij}, A_{ji}, F)$, where $A_{ij} = \{(u, j, c) : u \in V_i, c \in D\}$ is the set of indices of Y variables adjacent to nodes in V_i, going to position j, and A_{ji} is defined analogously, going in the opposite direction. The edge set F is defined in terms of the neighbors of a triple (u, j, c). Let $\mathcal{N}(u, j, c) = \{(v, i, c) : u \in V_i, (v, i, c) \in A_{ji} \text{ and } w_{uv} = c\}$ be the *neighbors* of (u, j, c). They are the indices of the Y_{vic} variables adjacent to position j going to position i so that the edge $\{u, v\}$ has weight c. There is an edge in F going between (u, j, c) and each of its neighbors. We call c the *cost* of triple (u, j, c). All this notation is summarized in Fig. 2(a).

In any feasible integral solution, if $Y_{ujc} = 1$, then some Y_{vic} for which $(v, i, c) \in \mathcal{N}(u, j, c)$ must also be 1. Extending this insight to subsets of the Y_{ujc} variables yields a class of constraints that will ensure that the resulting LP formulation is as tight as LP1. That is, choose any set of Y_{ujc} variables adjacent to position i. Their sum must be less than or equal to the sum of the Y variables for their neighbors. Formally, if $Q_{ij} \subseteq A_{ij}$, then let $\mathcal{N}(Q_{ij}) = \bigcup_{(u,j,c) \in Q_{ij}} \mathcal{N}(u, j, c)$ be the set of indices that are neighbors to any vertex in Q_{ij}. If $Q_{ij} \subseteq A_{ij}$ then $\mathcal{N}(Q_{ij}) \subseteq A_{ji}$. The following constraint is true in IP2 for any such Q_{ij}:

$$\sum_{(u,j,c) \in Q_{ij}} Y_{ujc} \leq \sum_{(v,i,c) \in \mathcal{N}(Q_{ij})} Y_{vic} \,. \tag{1}$$

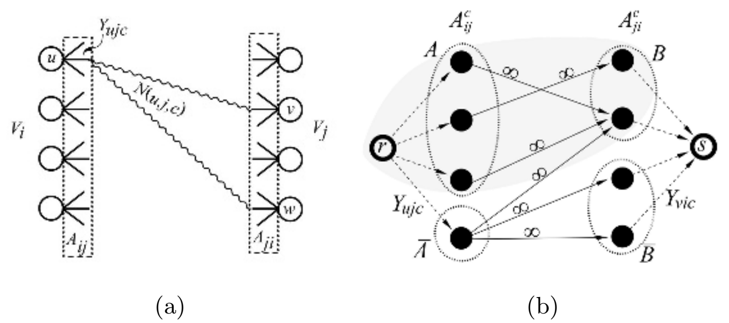


Fig. 2. (a) Mapping for the compatibility graph $\mathcal{C}_{ij}$. The two columns of circles represent nodes in V_i and V_j. Solid lines adjacent to each circle represent the Y_{ujc} or Y_{vic} variables associated with the node. A_{ij} and A_{ji} (dotted boxes) are the sets of these variables associated with the pair of graph parts i and j. The function $\mathcal{N}(u, j, c)$ maps a variable Y_{ujc} to a set of compatible Y_{vic} variables (squiggly lines). $\mathcal{N}(u, j, c)$ is shown assuming that v and w are the only nodes in V_j that have cost c with u. (b) Flow network $\mathcal{C}^c_{ij}$ between positions i and j. Nodes r and s are a source and sink. Each solid node corresponds to a Y variable. The edges between A_{ji} and A_{ij} have infinite capacity, while those entering s or leaving r have capacity equal to the value of the Y variable to which they are adjacent. The shading gives an $r - s$ cut.

Notice that the set of constraints (IP2c) is of the form (1), taking Q_{ij} to be the singleton set $\{(u, j, c)\}$.

Theorem 1. *If for every pair $i < j$, constraints of the form* (1) *are added to IP2 for each $Q \subseteq A_{ij}$ s.t. all triples in Q are of the same cost, the resulting LP relaxation LP2 has the same optimal solution as that of the relaxation LP1 of IP1.*

Proof. It is clear that the LP relaxation LP2 described in Theorem 1 is no stronger than LP1 as any solution to LP1 can be converted to a solution of LP2 by making the node variable weights the same and putting the weight of edge variables X_{uv} onto Y_{ujc} and Y_{vic}, where $w_{uv} = c$. This solution to LP2 will satisfy all the constraints in the theorem, and be of the same objective value.

The rest of the proof will involve showing that for any feasible solution for LP2, there is a feasible solution for LP1 with the same objective value, thereby demonstrating that the optimal solution to LP2 is not weaker than the optimal solution to LP1. In particular, fix a solution (X, Y) to LP2 with objective value γ. We need to show that for any feasible distribution of weights on the Y variables a solution to LP1 can be found with objective value γ.

In order to reconstruct a solution $\hat{X}$ for LP1 of objective value γ, we will set $\hat{X}_u = X_u$, using the values of the node variables X_u in the optimal solution to LP2. We must assign values to $\hat{X}_{uv}$ to complete the solution. Recall the compatibility graph $\mathcal{C}_{ij}$. Because all edges in $\mathcal{C}_{ij}$ are between nodes of the same cost, $\mathcal{C}_{ij}$ is really $|D|$ disjoint bipartite graphs $\mathcal{C}^c_{ij}$, one for each cost. Let $A^c_{ij} \cup A^c_{ji}$

be the node set for the subgraph $\mathcal{C}_{ij}^c$ for cost c. Each edge in a subgraph $\mathcal{C}_{ij}^c$ corresponds to one edge in the graph G underlying LP1. Conversely, each edge in G corresponds to exactly one edge in one of the $\mathcal{C}_{ij}^c$ graphs (if edge $\{u, v\}$ has cost c_1, it corresponds to an edge in $\mathcal{C}_{ij}^{c_1}$). We will thus proceed by assigning values to the edges in the various $\mathcal{C}_{ij}^c$, and this will yield values for the $\hat{X}_{uv}$.

If $y(A) := \sum_{(u,j,c)\in A} Y_{ujc}$, by the sets of constraints (IP2a) and (IP2b), $y(A_{ij}) = y(A_{ji}) = 1$. Since the constraints (1) are included with $Q = A_{ij}^c$ for each cost c, by the pigeonhole principle, $y(A_{ij}^c) = y(A_{ji}^c)$ for every cost c. Thus, for each subgraph $\mathcal{C}_{ij}^c$, the weight placed on the left half equals the weight placed on the right half. We will consider each induced subgraph $\mathcal{C}_{ij}^c$ separately.

We modify $\mathcal{C}_{ij}^c$ as follows to make it a capacitated flow network. Direct the edges of $\mathcal{C}_{ij}^c$ so that they go from A_{ij}^c to A_{ji}^c, and set the capacities of these edges to be infinite. Add source and sink nodes $\{r, s\}$ and edges directed from r to each node in A_{ij}^c and edges from each node in A_{ji}^c to s. Every edge adjacent to r and s is also adjacent to some node representing a Y variable; put capacities on these edges equal to the value of the adjacent Y variable (see Fig. 2(b)).

The desired solution to LP1 can be found if the weight of the nodes (Y variables) in each compatibility subgraph can be spread over the edges. That is, a solution to LP1 of weight γ can be found if, for each pair (i, j) and each c, there is a flow of weight $y(A_{ij}^c)$ from r to s in the flow network. The assignment to $\hat{X}_{uv}$ will be the flow crossing the corresponding edge in the $\mathcal{C}_{ij}^c$ of appropriate cost. In the following lemma, we show that the set of constraints described in the theorem ensure that the minimum cut in the flow network is $\geq y(A_{ij}^c)$, and thus there is a flow of the required weight. The proof of this fact is quite similar to those of other flow feasibility problems found in [5], and we omit it here. Together with the lemma we have shown LP1 and LP2 to be equivalent. □

Lemma 2. *The minimum cut of the flow network described in the proof of Theorem 1 (and shown in Fig. 2(b)) is $y(A_{ij}^c)$.*

4.1 Separation Algorithm

Despite the exponential number of constraints, it is possible to solve LP2 in polynomial time by the ellipsoid algorithm [6] provided that there exists a separation algorithm that finds a violated constraint, if one exists, in polynomial time or reports that no constraints are violated. The next lemma gives such an algorithm, formalizing the intuition in the proof of Theorem 1, by which all constraints are satisfied in a compatibility graph only if a large enough maximum flow exists. Otherwise, the minimum cut identifies a violated constraint.

Theorem 2. *There is a polynomial-time algorithm that can find a violated constraint in LP2 or report that none exists.*

Proof. Because each constraint in (1) involves variables of a single cost, if (1) is violated for some set Q, then Q is a subset of an A_{ij}^c for some i, j, c, and so we can consider each subgraph $\mathcal{C}_{ij}^c$ independently. The proof of Theorem 1 shows that there is a violated constraint of the form (1) between i, j involving variables of cost c if and only if the maximum flow in C_{ij}^c is less than $y(A_{ij}^c)$. Thus, the minimum cut can be found for each triple i, j, c, and, if a triple i, j, c is found where the minimum cut is less than $y(A_{ij}^c)$, one knows that a violated constraint exists between positions i and j with $Q \subset A_{ij}^c$.

The minimum cut can then be examined to determine the violated constraint explicitly. Let $\{r\} \cup A \cup B$ be the minimum $r - s$ cut in C_{ij}^c, with $A \subseteq A_{ij}^c$ and $B \subseteq A_{ji}^c$. Such a cut is shaded in Fig. 2(b). Let m be the capacity of this cut, and assume, because we are considering a triple i, j, c that was identified as having a violated constraint, that $y(A_{ij}^c) > m$. For ease of notation let $\bar{A} = A_{ij}^c \setminus A$ and $\bar{B} = A_{ji}^c \setminus B$. Because $m < \infty$ there are no edges going from A to $\bar{B}$, and hence two things hold: (1) $m = y(B) + y(\bar{A})$ and (2) $\mathcal{N}(A) \subseteq B$, and therefore $y(\mathcal{N}(A)) \leq y(B)$. Chaining these facts together, we have

$$y(A) = y(A_{ij}^c) - y(\bar{A}) > m - y(\bar{A}) = y(B) \geq y(N(A)),$$

Thus, the set A is a set for which the constraint of the form (1) is violated. □

5 Computational Results

We apply our LP formulation to find binding sites for *E. coli* transcription factors, and we show that in practice our LP formulation results in significantly faster running times than the previous simpler linear program. Moreover, in order to demonstrate that our formulation of the motif finding problem results in biologically relevant solutions, we show that our approach identifies binding sites as well as a widely-used probabilistic technique [20].

Test Sets. We present results on identifying the binding sites of 39 *E. coli* transcription factors (see Table 1). We construct our data set from the data of [17, 14] in a fashion similar to [15]. In short, we remove all sites for sigma-factors, duplicate sites, as well as those that could not be unambiguously located in the genome. Data sets for all factors with only two sites remaining were discarded as uninteresting for motif finding; datasets for *ihf* and *crp* are omitted

Table 1. Sizes for the 39 problems considered: number of sequences (p), motif length (ℓ), and total number of nodes in the underlying graph (n)

TF	ℓ	p	n	TF	ℓ	p	n	TF	ℓ	p	n	TF	ℓ	p	n	TF	ℓ	p	n
ada	31	3	810	dnaA	15	8	2381	galR	16	7	2188	metJ	16	15	5754	phoB	22	14	4618
araC	48	6	1715	fadR	17	7	2122	gcvA	20	4	1234	metR	15	8	3312	purR	26	20	5856
arcA	15	13	4790	farR	10	3	873	glpR	20	11	3829	modE	24	3	934	soxS	35	13	4004
argR	18	17	5960	fis	35	18	5371	hipB	30	4	1084	nagC	23	6	1870	torR	10	4	2198
cpxR	15	9	2614	flhCD	31	3	810	hns	11	5	1485	narL	16	10	3301	trpR	24	4	1108
cspA	20	4	1410	fnr	22	12	3705	lexA	20	19	5554	ntrC	17	5	1516	tus	23	5	1390
cysB	40	3	783	fruR	16	11	4082	lrp	25	14	4090	ompR	20	9	3057	tyrR	22	17	5258
cytR	18	5	1695	fur	18	9	3182	malT	10	10	3410	oxyR	39	4	1048				

due to size considerations. For each transcription factor considered, we gather at least 300 base pairs of genomic sequence upstream of the transcription start sites of the regulated genes. In the cases where the binding site is located further upstream, we extend the sequence to include the binding site. This results in graphs with up to 20 parts and $5,960$ nodes. The motif length for each dataset was chosen based on the length of the consensus binding site, determined from other biological studies and ranging between 11 and 48. The transcription factors, the length of their binding site, and the number of DNA sequences considered are shown in Table 1.

Methodology. We first solve the LP relaxation of IP2. If the solution is not integral, we find and add violated constraints and re-solve. We have observed that certain classes of constraints of the form (1) are powerful in practice, and so we consider these first:

1. $Q_{ij} = A^c_{ij}$ for every $i < j, c$.
2. $Q_{ij} = \{(u, j, c) : c \in D\}$ for every $i < j, u \in V_i$.

In addition, we consider the above constraints with $i > j$. We iterate, adding all violated constraints of the above types and re-solving, until all such constraints are satisfied. While in theory this heuristic approach may lead to a solution that is not as tight as that of LP1, in all cases considered, we find that adding this particular set of constraints is sufficient for making LP2 as tight as LP1. Moreover, in practice, this heuristic approach will be faster than using the ellipsoid method [6] with our separation algorithm and, we show below, is usually faster than solving LP1.

LP1 was solved using two different simplex variants. In the first (`primal dualopt`), the primal problem was solved using the dual simplex algorithm. In the second (`dual primalopt`), the dual problem was solved using the primal simplex algorithm. LP2 was always solved using the dual simplex method applied to the primal problem so that we could use the optimal basis of the previous iteration as a starting point for the next, setting the dual variables for the added constraints to be basic. This strategy eliminates the need to re-solve using an arbitrary starting solution and provides a significant speedup.

The linear and integer programs were specified with Ampl and solved using CPLEX 7.1. All experiments were run on a public 1.2 GHz SPARC workstation using a single processor. All the timings reported are in CPU seconds. Any problem taking longer than five hours was aborted. Interestingly, only 3 of the 34 problems solvable in less than five hours were not integral. Since the problem is NP-complete, this is somewhat surprising. This suggests that handling non-integral cases may not be as pressing as one would think.

Performance of the LP relaxations. We solved LP1 and LP2 relaxations for the transcription factors listed in Table 1. Fig. 3 plots the running times, matrix sizes, defined as the number of constraints times the number of variables, and speed-up factors of LP2 over LP1. For five problems, each LP failed to find

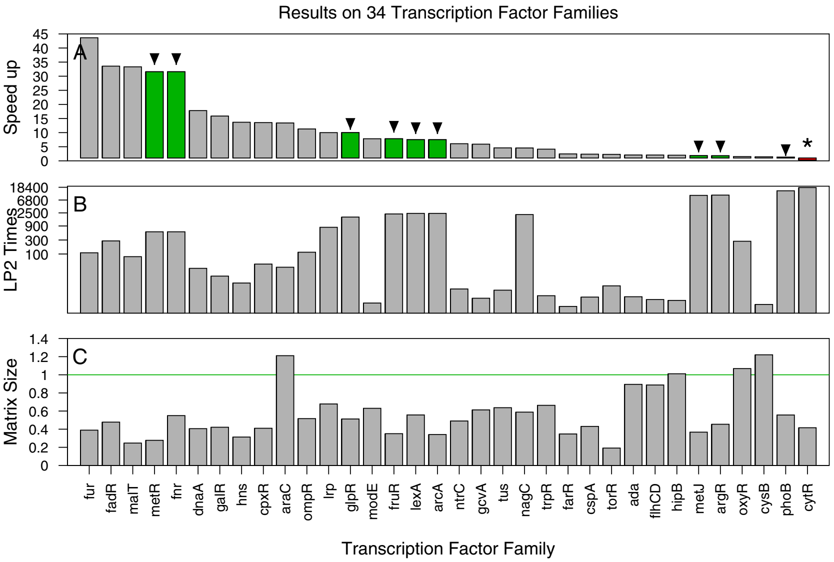


Fig. 3. (a) Speed-up factor of LP2 over LP1. A triangle indicates problems for which LP1 did not finish in less than five hours. An asterisk (far right) marks the problem for which LP2 did not finish in less than five hours, but LP1 did. (b) Running times in seconds for LP2 (log scale). (c) Ratio of matrix sizes for LP2 to LP1.

a solution in the allotted five hours; these are omitted from the figure. In most cases, the initial set of constraints was sufficient to get a solution at least as good as that obtained by LP1. Six problems required additional constraints to LP2 to make their solutions as tight. The problems *flhCD*, *torR*, and *hu* required two iterations of adding violated constraints, *ompR* required three, *oxyR* four, and *nagC* five. Running times reported in Fig. 3(b) are the sum of the initial solve times and of all the iterations. Fig. 3(c) plots (size of LP2)/(size of LP1). As expected, the size of the constraint matrix is typically smaller for LP2. While in four cases the matrix for LP2 is larger, often it is < 50% the size of the matrix for LP1.

When comparing the running times of LP2 with those of LP1, the speed-up factor is computed as $\min\{\texttt{primal dualopt LP1}, \texttt{dual primalopt LP1}\}/\texttt{LP2}$, that is, using the better running time for LP1. For all but one of the datasets, a significant speed-up when using LP2 is observed, and an order of magnitude speed-up is common, as shown in Fig. 3(a). For nine problems, while LP2 was solved, neither simplex variant completed in < 5 hours when solving LP1. For these problems, the timing for LP1 was set at five hours, giving a lower bound on the speed up. For one problem, *cytR*, the reverse was true and LP2 did not finish within five hours, while LP1 successfully solved the problem. For this dataset, the timing for LP2 was taken to be five hours, giving an upper bound.

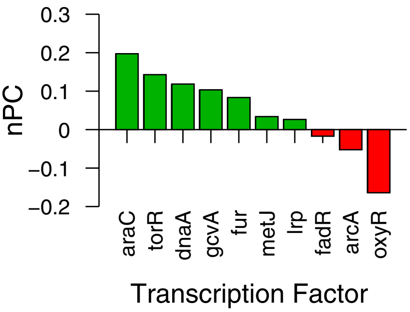


Fig. 4. Difference between *nPC* values obtained using the ILP approach and Gibbs Motif Sampler [20]; data sets with identical motifs are omitted. Bars above zero indicate that ILP performs better.

We also compared the performance of our approach, measured by the nucleotide performance coefficient (*nPC*) [21], in identifying existing transcriptionfactor binding sites to that of Gibbs Motif Sampler [20]. The *nPC* measures the degree of overlap between known and predicted motifs, and is defined as $nTP/(nTP + nFN + nFP)$, where *nTP*, *nFP*, *nTN*, *nFN* refer to nucleotide level true positives, false positives, true negatives and false negatives respectively. We compare the *nPC* values for the two methods in Fig. 4. Each bar in the chart measures the difference in *nPC* between the ILP approach and Gibbs Motif Sampler, omitting those transcription factor datasets for which the found motifs are identical. Of the 30 problems for which the integral optimal was found using LP2, the sum-of-pairwise hamming distances measure more accurately identifies the biologically known motif in seven cases, with *nPC* 0.11 better on average. In 20 cases, the two methods find equally good solutions. In the remaining 3 cases, Gibbs sampling does better, with *nPC* 0.08 better on average. Since the Gibbs sampling approaches have comparable performance to other stochastic motif finding methods [21] and most combinatorial methods are restricted by the lengths of the motifs considered, our ILP framework provides an effective alternative approach for identifying DNA sequence motifs.

6 Conclusions

We introduced a novel ILP for the motif finding problem that works well in practice. There are many interesting avenues for future work. While the underlying graph problem is similar to that of [4, 9], one central difference is that the edge weights satisfy the triangle inequality. In addition, edge weights in the graph are not independent, as each node represents a subsequence from a sliding window. Incorporating these features into the ILP may lead to further advances in computational methods for motif finding. It would also be useful to extend the basic formulation presented here to find multiple co-occurring or repeated motifs (as supported by many widely-used packages). Finally, we note that graph pruning and decomposition techniques (e.g., [16, 23]) may allow mathematical programming formulations to tackle problems of considerably larger size.

Acknowledgments. M.S. thanks the NSF for PECASE award MCB-0093399 and DARPA for award MDA972-00-1-0031.

References

1. Akutsu, T., Arimura, H., Shimozono, S. On approximation algorithms for local multiple alignment. RECOMB, 2000, pp. 1–7.
2. Bafna, V., Lawler, E., Pevzner P. A. Approximation algorithms for multiple alignment. Theoretical Computer Science 182: 233–244, 1997.
3. Bailey, T., Elkan, C. Unsupervised learning of multiple motifs in biopolymers using expectation maximization. Machine Learning 21: 51–80, 1995.
4. Chazelle, B., Kingsford, C., Singh M. A semidefinite programming approach to side-chain positioning with new rounding strategies, INFORMS J. on Computing 16: 380–392, 2004.
5. Cook, W., Cunningham, W., Pulleyblank, W., Schrijver, A. Combinatorial Optimization. Wiley-Interscience, New York, 1997.
6. Grötschel, M., Lovász, L., Schrijver, A. 1993. Geometric Algorithms and Combinatorial Optimization. Springer-Verlag, Berlin, Germany, 2nd edition.
7. Hertz, G., Stormo, G. Identifying DNA and protein patterns with statistically significant alignments of multiple sequences. Bioinf. 15: 563–577, 1999.
8. Kellis, M. Patterson, N., Endrizzi, M., Birren, B., Lander E. Sequencing and comparison of yeast species to identify genes and regulatory elements. Nature 423: 241–254, 2003.
9. Kingsford, C., Chazelle, B., Singh, M. Solving and analyzing side-chain positioning problems using linear and integer programming. Bioinf. 21: 1028–1039, 2005.
10. Lawrence, C., Altschul, S., Boguski, M., Liu, J., Neuwald, A., Wootton, J. Detecting subtle sequence signals: a Gibbs sampling strategy for multiple alignment. Science 262: 208–214, 1993.
11. Lee, T., Rinaldi, N., Robert, F., Odom, D., Bar-Joseph, Z., Gerber, G. *et al.* Transcriptional regulatory networks in *S. cerevisiae.* Science 298: 799–804, 2002.
12. Li, M., Ma, B., Wang, L. Finding similar regions in many strings. J. Computer and Systems Sciences 65(1): 73–96, 2002.
13. Marsan L., Sagot M.F. Algorithms for extracting structured motifs using a suffix tree with an application to promoter and regulatory site consensus identification. J. Comp. Bio. 7:345-362, 2000.
14. McGuire, A., Hughes, J., Church, G. Conservation of DNA regulatory motifs and discovery of new motifs in microbial genomes. Genome Res. 10: 744–757, 2000.
15. Osada, R., Zaslavsky, E., Singh, M. Comparative analysis of methods for representing and searching for transcription factor binding sites. Bioinf. 20: 3516–3525, 2004.
16. Pevzner, P., Sze, S. Combinatorial approaches to finding subtle signals in DNA sequences. ISMB, 2000, pp. 269–278.
17. Robison, K., McGuire, A., Church, G. A comprehensive library of DNA-binding site matrices for 55 proteins applied to the complete *Escherichia coli* K-12 Genome. J. Mol. Biol. 284: 241–254, 1998.
18. Schuler, G., Altschul, S., Lipman, D. A workbench for multiple alignment construction and analysis. Proteins 9(3): 180–190, 1991.
19. Tavazoie, S., Hughes, J., Campbell, M., Cho, R., Church, G. Systematic determination of genetic network architecture. Nat. Genetics 22(3): 281–285, 1999.

20. Thompson, W., Rouchka, E., Lawrence, C. Gibbs Recursive Sampler: finding transcription factor binding sites. Nucleic Acids Res. 31: 3580-3585, 2003.
21. Tompa, M., Li, N., Bailey, T., Church, G., De Moor, B., Eskin, E., *et al.* Assessing computational tools for the discovery of transcription factor binding sites. Nat. Biotech. 23: 137–144, 2005.
22. Wang, L., Jiang, T. On the complexity of multiple sequence alignment. J. Comp. Bio. 1: 337–348, 1994.
23. Zaslavsky, E., Singh, M. Combinatorial Optimization Approaches to Motif Finding. Submitted. Also available as Princeton University Computer Science Dept. Technical Report TR-728-05.

Local Alignment of RNA Sequences with Arbitrary Scoring Schemes

Rolf Backofen[1], Danny Hermelin[2,⋆],
Gad M. Landau[2,3], and Oren Weimann[4]

[1] Institute of Computer Science,
Albert-Ludwigs Universität Freiburg, Freiburg - Germany
backofen@informatik.uni-freiburg.de
[2] Department of Computer Science,
University of Haifa, Haifa - Israel
danny@cri.haifa.ac.il, landau@cs.haifa.ac.il
[3] Department of Computer and Information Science,
Polytechnic University, New York - USA
[4] Computer Science and Artificial Intelligence Laboratory,
MIT, Cambridge MA - USA
oweimann@mit.edu

Abstract. Local similarity is an important tool in comparative analysis of biological sequences, and is therefore well studied. In particular, the Smith-Waterman technique and its normalized version are two established metrics for measuring local similarity in strings. In RNA sequences however, where one must consider not only sequential but also structural features of the inspected molecules, the concept of local similarity becomes more complicated. First, even in global similarity, computing global sequence-structure alignments is more difficult than computing standard sequence alignments due to the bi-dimensionality of information. Second, one can view locality in two different ways, in the sequential or structural sense, leading to different problem formulations.

In this paper we introduce two sequentially-local similarity metrics for comparing RNA sequences. These metrics combine the global RNA alignment metric of Shasha and Zhang [16] with the Smith-Waterman metric [17] and its normalized version [2] used in strings. We generalize the familiar alignment graph used in string comparison to apply also for RNA sequences, and then utilize this generalization to devise two algorithms for computing local similarity according to our two suggested metrics. Our algorithms run in $\mathcal{O}(m^2 n \lg n)$ and $\mathcal{O}(m^2 n \lg n + n^2 m)$ time respectively, where $m \leq n$ are the lengths of the two given RNAs. Both algorithms can work with any arbitrary scoring scheme.

1 Introduction

Ribonucleic acids (RNAs) are polymers consisting of the four nucleotides Adenine, Cytosine, Guanine, and Uracil, which are linked together by their phosphodiester bonds. Bases which are part of the nucleotides form hydrogen bonds

⋆ Partially supported by the Israel Science Foundation grant 282/01.

M. Lewenstein and G. Valiente (Eds.): CPM 2006, LNCS 4009, pp. 246–257, 2006.

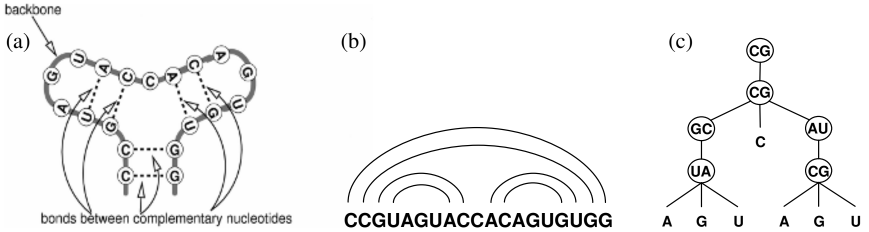


Fig. 1. Three different ways of viewing an RNA sequence. In (a), a schematic 2-dimensional description of an RNA folding. In (b), a linear representation of the RNA. In (c), the RNA as a rooted ordered tree.

within the same molecule leading to structure formation. These hydrogen bonds are referred to as base pairs, and the set of all base pairs is called the secondary structure of the RNA. The role of RNA in biological systems was largely underestimated for a long time. Today, RNA enjoys increasing attention due to recent developments such as the discovery of ribozymes (RNA-molecules with enzymatic properties), and the observation that non-coding RNA molecules play an enormous role in cell control. As an example, research on non-coding RNAs has been elected as the scientific breakthrough of 2002 by the readers of Science [6].

One major challenge of research on RNAs is to find common patterns since these suggest functional similarities in the inspected molecules. For this purpose, one has to investigate not only sequential features, but also structural features for the following reasons. First, a major fraction of the function of an RNA is determined by its secondary structure [15]. Second, it is known that the structure of an RNA is often more conserved than its sequence during evolution [4]. Thus, two RNA sequences with their corresponding secondary structure are aligned using both sequential and structural information for scoring the alignment.

There have been quite a few approaches for defining alignments in terms of RNAs. The first one is due to the seminal paper of Shasha and Zhang [16] which represented RNA sequences as rooted ordered trees, and defined editing operations on trees which correspond to editing operations on RNA sequences. In this way, an alignment of two RNA sequences corresponds to a sequence of editing operations on two corresponding trees, and any tree editing algorithm can be used to compute the optimal alignment of two RNAs. Furthermore, this approach allows base pairs to be considered as whole entities, meaning that one can require any base pair to either be deleted (resulting in a removal of two nucleotides) or be aligned against another base pair in the opposite RNA. Since [16], there have been attempts at extending either the set of edit operations on trees [1, 11], or the set of allowed RNA alignments [13], in order to model certain biological mutations that weaken and ultimately break bonds between base pairs. Usually, these extensions introduce an increase in the time complexities of the algorithms required to compute them.

In RNA sequences, as in many other biological applications, searching for local similarities is at least as important as determining global similarity. In contrast, most RNA sequence-structure alignment methods are global. To our knowledge, there are only a few exceptions for this, namely [3, 5, 7, 10, 18]. These can be divided roughly into two main categories, depending on the exact notion of locality under consideration. The first category of [3, 7, 18] defines locality in the structural sense, thus allowing large gaps in the sequences not to be considered as relevant in the alignment score. The second category of [5, 10] defines locality in the sequential sense, thus extending the well understood notion of locality in strings to RNA sequences.

In this paper we introduce two sequentially-local metrics for RNA local alignment. The first one is a natural extension of the Smith-Waterman metric used in strings [17]. The second one is a a normalized variant of the first metric, where one divides the alignment score of two local regions by the sum of their lengths. This metric was suggested for string comparison by [2], and was dealt also in [9].

Our Results: We give two algorithms for computing the optimal local alignment score of two RNA sequences of lengths m and n, $m \leq n$. The first algorithm computes in $\mathcal{O}(m^2 n \lg n)$ time the optimal local alignment score according to our extension of the Smith-Waterman metric. The second one computes the optimal normalized local alignment score in $\mathcal{O}(m^2 n \lg n + n^2 m)$ time. Both algorithms work with any arbitrary scoring scheme.

Roadmap: The rest of this paper is organized as follows. We next introduce notations and terminology that will be used throughout the paper. Following this, in Section 2, we discuss the notion of alignment for RNA sequences. In Section 3, we discuss local alignment and introduce two new local similarity metrics that we will be dealing with throughout the paper. Section 4 then describes an adaptation of the familiar alignment graph used in string comparison to an alignment graph for RNA sequences. This adapted graph is then used in Section 5 to design two algorithms that compute the local alignment score between a pair RNA sequences according to our two suggested metrics. Due to space limitations, all proofs are omitted from this version of the paper.

Notations: An RNA sequence $\mathcal{R}$ is an ordered pair (S, A), where $S = s_1 \cdots s_{|S|}$ is a string over the alphabet $\Sigma = \{A, C, G, U\}$, and $A \subseteq \{1, \ldots, |S|\} \times \{1, \ldots, |S|\}$ is the set of hydrogen bonds between bases of $\mathcal{R}$ (*i.e.* the secondary structure). Any base in $\mathcal{R}$ can bond with at most one other base, therefore we have $\forall\ (i'_1, i_1), (i'_2, i_2) \in A$, $i'_1 = i'_2 \Leftrightarrow i_1 = i_2$. Furthermore, following Zuker [19, 20], we assume a model where the bonds in A are *non crossing*, *i.e.* for any $(i'_1, i_1), (i'_2, i_2) \in A$, we cannot have $i'_1 < i'_2 < i_1 < i_2$ nor $i'_2 < i'_1 < i_2 < i_1$. We refer to a bond $(i', i) \in A$, $i' < i$, as an *arc*, and i' and i are referred to as the *left* and *right endpoints* of this arc. Also, we let $|\mathcal{R}|$ denote the number of nucleotides in $\mathcal{R}$, *i.e.* $|\mathcal{R}| = |S|$.

We will require a notion similar to that of a substring for RNA sequences. Therefore, for any $1 \leq i' \leq i \leq |\mathcal{R}|$, we let $\mathcal{R}[i', i] = (S[i', i], A[i', i])$, the *consecutive subsequence* of $\mathcal{R}$, be the RNA with $S[i',i] = S_{i'} \cdots S_i$ and $A[i',i] = A \cap \{i',\ldots,i\} \times \{i',\ldots,i\}$. If $(i',i) \in A$, then we say that arc (i',i) *wraps* $\mathcal{R}[i',i]$. Also, for convenience purposes, we slightly abuse notation and let $\mathcal{R}[i+1,i] = (\emptyset,\emptyset)$ denote the empty RNA for any $1 \leq i \leq |\mathcal{R}|$. Note that arcs of $\mathcal{R}$ with one endpoint in $\mathcal{R}[i',i]$ are absent in $\mathcal{R}[i',i]$. These arcs are said to be *broken* in $\mathcal{R}[i',i]$. A position $l \in \{i',...,i\}$ is considered an arc endpoint in $\mathcal{R}[i',i]$, even if it is an arc endpoint of an arc which is broken in $\mathcal{R}[i',i]$.

This paper deals with comparing two RNA sequences. We denote these two RNAs by $\mathcal{R}_1 = (S_1,A_1)$ and $\mathcal{R}_2 = (S_2,A_2)$ throughout the paper, and we set $|\mathcal{R}_1| = |S_1| = n$ and $|\mathcal{R}_2| = |S_2| = m$. Furthermore, we assume $m \leq n$.

2 RNA Alignment

As in the case of strings, RNA alignment is analogous to the edit distance of two RNAs, *i.e.* the minimum number of edit operations necessary in order to transform one RNA into the other [16]. The edit operations defined for RNA molecules are similar to those defined for strings, except that here we can perform editing operations on arcs as well as on unpaired nucleotides. The allowed edit operations are therefore insertion, deletion, and relabeling of arcs and nucleotides on either one of the given RNAs. Defining separate operations on arcs and unpaired nucleotides captures the notion of arcs and unpaired bases being different entities.

An *alignment* of $\mathcal{R}_1$ and $\mathcal{R}_2$ is another way of viewing a sequence of edit operations on these two RNAs. Formally, it is defined as follows:

Definition 1 (Alignment). *An* alignment $\mathcal{A}$ *of* $\mathcal{R}_1$ *and* $\mathcal{R}_2$ *is a subset of* $\{1,\ldots,n\} \cup \{-\} \times \{1,\ldots,m\} \cup \{-\}$ *satisfying the following conditions:*

- $(-,-) \notin \mathcal{A}$.
- $\forall (i,j) \in \mathcal{A} \cap \{1,\ldots,n\} \times \{1,\ldots,m\}$: i *and* j *appear exactly once in* $\mathcal{A}$.
- $\forall (i',j'),(i,j) \in \mathcal{A} \cap \{1,\ldots,n\} \times \{1,\ldots,m\} : i' < i \iff j' < j$. *That is, any two pairs in* $\mathcal{A}$ *are non-crossing.*
- $\forall (i,j) \in \mathcal{A} \cap \{1,\ldots,n\} \times \{1,\ldots,m\}$: i *is a left (resp. right) arc endpoint in* $\mathcal{R}_1 \iff j$ *is a left (resp. right) arc endpoint in* $\mathcal{R}_2$.
- $\forall (i',i) \in A_1, (j',j) \in A_2 : (i',j') \in \mathcal{A} \iff (i,j) \in \mathcal{A}$. *That is, the left endpoints of any pair of arcs are aligned against each other in* $\mathcal{A}$ *iff their right endpoints are also aligned against each other in* $\mathcal{A}$.

In terms of editing operations, a pair $(i,j) \in \mathcal{A} \cap \{1,\ldots,n\} \times \{1,\ldots,m\}$ corresponds to relabeling the ith nucleotide (unpaired or not) of $\mathcal{R}_1$ so it would match the jth nucleotide of $\mathcal{R}_2$, while pairs $(i,-)$ and $(-,j)$ corresponds to deleting the ith and jth nucleotides in $\mathcal{R}_1$ and $\mathcal{R}_2$. The first three conditions in the above definition require any position in $\mathcal{R}_1$ and $\mathcal{R}_2$ to be aligned, and that $(-,-) \notin \mathcal{A}$, since $(-,-)$ does not correspond to any valid edit operation. The next condition enforces the order of the subsequences to be preserved in $\mathcal{A}$, and the last two conditions restrict any arc to be either deleted or aligned against another arc in the opposite RNA.

Let $\Sigma' = \Sigma \cup \{-\}$. A *scoring scheme* $\delta = (\delta_1, \delta_2)$ for alignments of $\mathcal{R}_1$ and $\mathcal{R}_2$ is an ordered pair of two separate scoring functions $\delta_1 : \Sigma' \times \Sigma' \longrightarrow \mathbb{Z}$ and $\delta_2 : \Sigma'^2 \times \Sigma'^2 \longrightarrow \mathbb{Z}$, one which measures the quality of aligning a single unpaired nucleotide of $\mathcal{R}_1$ against another unpaired nucleotide of $\mathcal{R}_2$, and the other for measuring the quality of aligning pairs of arcs of the two RNA sequences. Hence $\delta_2((S_1[i'], S_1[i]), (-,-))$ denotes, for example, the deletion of the arc $(i', i) \in A_1$. We assume the scoring scheme is a similarity metric, and so a high score is given for aligning similar arcs or similar unpaired nucleotides, while different penalties are given in all other possible cases.

For brevity of notation, let us write $\delta_1(i,j)$ to denote the value $\delta_1(S_1[i], S_2[j])$ if $i, j \neq -$, and $\delta_1(S_1[i], -)$ (resp. $\delta_1(-, S_2[j])$) if j (resp. i) is the blank symbol '$-$'. Also, we write $\delta_2(i', i, j', j)$ instead of $\delta_2((i', i), (j', j))$. The *score* of an alignment $\mathcal{A}$ of $\mathcal{R}_1$ and $\mathcal{R}_2$ with respect to δ is given by:

$$\delta(\mathcal{A}) = \sum_{\substack{(i,j)\in\mathcal{A},\ i,j \text{ are not} \\ \text{arc endpoints}}} \delta_1(i,j) + \sum_{\substack{(i',j'),(i,j)\in\mathcal{A}, \\ (i',i)\in A_1 \wedge (j',j)\in A_2}} \delta_2(i', i, j', j).$$

Definition 2 ($OPT_\delta(\mathcal{R}'_1, \mathcal{R}'_2)$). *Given two RNA sequences $\mathcal{R}'_1$ and $\mathcal{R}'_2$ and a scoring scheme $\delta = (\delta_1, \delta_2)$, $OPT_\delta(\mathcal{R}'_1, \mathcal{R}'_2)$ denotes the highest score of any alignment of $\mathcal{R}'_1$ and $\mathcal{R}'_2$ with respect to δ.*

2.1 RNA Alignment Via Ordered Tree Editing

The non crossing formation formed by the arcs in both $\mathcal{R}_1$ and $\mathcal{R}_2$ conveniently allows representing these RNAs as rooted ordered trees [16]. Each arc (i, i') is identified with a set of ordered children which are all unpaired bases i'' such that $i < i'' < i'$, and arcs (l, l') such that $i < l < l' < i'$ (see Figure 1). In [16], Shasha and Zhang suggested an algorithm for computing the edit distance between two ordered trees. Their algorithm was later improved by Klein [14] to an $\mathcal{O}(m^2 n \lg n)$ algorithm, where $m \leq n$ denote the number of nodes in the two trees. (Recently, Demaine *et al.* [8] presented an $\mathcal{O}(m^2 n(1+\lg \frac{n}{m}))$ improvement to this algorithm. Using their algorithm improves the results of this paper to $\mathcal{O}(m^2 n(1 + \lg \frac{n}{m}))$ time for the Smith-Waterman metric, and $\mathcal{O}(n^2 m)$ for the normalized metric).

Not by chance, the edit operations defined for trees are analogous to the ones defined for RNA sequences. For this reason, any tree editing algorithm can be used to determine the global alignment score of two RNAs, with the slight modification that a penalty of ∞ is assigned for relabeling a node which corresponds to an unpaired nucleotide by a label corresponding to a base pair, and vise versa. Furthermore, as a side effect of the recursions used in [14, 16], these algorithms compute the optimal alignment between every pair of rooted subtrees of the two given trees, assuming this alignment matches the roots of the two subtrees. In our setting, this means that $\delta_2(i', i, j', j) + OPT_\delta(\mathcal{R}_1 [i'+1, i-1], \mathcal{R}_2[j+1, j-1])$ is computed between all pairs of arcs $(i', i) \in A_1$ and $(j', j) \in A_2$ in a single execution of either algorithms. The importance of this property will become apparent later on.

Definition 3 ($OPT_{\delta}^{arc}(\mathcal{R}_1[i',i],\mathcal{R}_2[j',j])$). *For a pair of arcs $(i',i) \in A_1$ and $(j',j) \in A_2$, we set $OPT_{\delta}^{arc}(\mathcal{R}_1[i',i],\mathcal{R}_2[j',j]) = \delta_2(i',i,j',j) + OPT_{\delta}(\mathcal{R}_1[i'+1,i-1],\mathcal{R}_2[j+1,j-1])$.*

3 Local Alignment

While the metric described in the previous section is suitable for measuring global similarity of two RNA sequences, in many applications two RNAs may not be very similar when both considered as a whole, but may contain many regions of high similarity. The goal is then to extract a pair of regions, one from each RNA, which admits a strong degree of similarity. This is known as *local similarity.* In the following section we introduce two metrics for measuring local similarity between RNA sequences. These metrics are extensions of the Smith-Waterman [17] and normalization [2] techniques used for strings.

A *local alignment* of $\mathcal{R}_1$ and $\mathcal{R}_2$, one which corresponds to a pair of contiguous regions in the RNAs, is an alignment of two consecutive subsequences $\mathcal{R}_1[i',i]$ and $\mathcal{R}_2[j',j]$. Note that the last condition of Definition 1 implies that any arc endpoint of a broken arc in each subsequence must be aligned with the blank symbol '$-$'. However, we need to distinguish between this situation and a deletion of an unpaired nucleotide. Therefore, we use $\delta_2(l',-,-,-)$ (resp. $\delta_2(-,l,-,-)$) to denote the cost of aligning the left (resp. right) endpoint of a broken edge (l',l) in $\mathcal{R}_1[i',i]$ against '$-$', and symmetrically, $\delta_2(-,-,l',-)$ and $\delta_2(-,-,-,l)$ are used to denote the costs of aligning the endpoints of a broken edge (l',l) in $\mathcal{R}_1[i',i]$ against '$-$'. Furthermore, we require that the total cost of aligning the left and right endpoints of a broken edge (in two different alignments) be equal to the cost of deleting this edge. That is, $\delta_2(l',-,-,-) + \delta_2(-,l,-,-) = \delta_2(l',l,-,-)$ and $\delta_2(-,-,l',-) + \delta_2(-,-,-,l) = \delta_2(-,-,l',l)$.

3.1 Standard Local Alignment

The well known Smith-Waterman [17] technique for computing local similarity between strings has been extensively studied in the literature. It is defined as the highest scoring alignment between any pair of substrings of the input strings. The simplicity of this definition has gained it wide applicability in many biological settings [12]. In our terms, it is defined by:

$$\max\left\{ OPT_{\delta}(\mathcal{R}_1[i',i],\mathcal{R}_2[j',j]) \;\middle|\; \begin{array}{l} 1 \leq i' \leq i \leq |\mathcal{R}_1|, \\ 1 \leq j' \leq j \leq |\mathcal{R}_2| \end{array} \right\}.$$

We refer to the Smith-Waterman metric as the *standard local alignment score* of $\mathcal{R}_1$ and $\mathcal{R}_2$. The computational problem corresponding to this metric is then defined as follows:

Definition 4 (The standard local alignment problem). *Given two RNA sequences $\mathcal{R}_1 = (S_1, A_1)$ and $\mathcal{R}_2 = (S_2, A_2)$, determine the standard local alignment score of $\mathcal{R}_1$ and $\mathcal{R}_2$.*

3.2 Normalized Local Alignment

According to [2], the Smith-Waterman technique has two weaknesses that make it non optimal as a local similarity measure. The first weakness is called the *mosaic effect*. This term describes the algorithm's inability to discard poorly conserved intermediate segments, although it can discard poor prefixes or suffixes of a segment. The second weakness is known as the *shadow effect*. This term describes the tendency of the algorithm to lengthen long alignments with a high score rather than shorter alignments with a lower score and a higher degree of similarity.

One way to overcome these weaknesses is to normalize the alignment score of two substrings by dividing it with their total length [2]. In our terms, the *normalized alignment score* of $\mathcal{R}_1$ and $\mathcal{R}_2$ is defined by:

$$\max\left\{ \frac{OPT_\delta(\mathcal{R}_1[i,i'],\mathcal{R}_2[j,j'])}{|\mathcal{R}_1[i,i']|+|\mathcal{R}_2[j,j']|} \;\middle|\; \begin{array}{l} 1 \le i \le i' \le |\mathcal{R}_1|, \\ 1 \le j \le j' \le |\mathcal{R}_2|, \\ OPT_\delta(\mathcal{R}_1[i,i'],\mathcal{R}_2[j,j']) \ge I \end{array} \right\}.$$

Where $I \in \mathbb{N}$ is an integer regulating the minimum score (before normalization) of solution alignments, predefined according to the application at hand. Note that this additional parameter is necessary for preventing trivial solutions (*e.g.* a single match) from being optimal.

Definition 5 (The normalized local alignment problem). *Given two RNA sequences $\mathcal{R}_1 = (S_1, A_1)$ and $\mathcal{R}_2 = (S_2, A_2)$, and an integer I, determine the normalized local alignment score of $\mathcal{R}_1$ and $\mathcal{R}_2$.*

4 An Alignment Graph for RNA Sequences

We next present an adaptation of the alignment graph that is used to describe string alignment [12], to an alignment graph that describes alignments of RNA sequences. Later, in Section 5, this adapted graph will be utilized for computing the local similarity score of $\mathcal{R}_1$ and $\mathcal{R}_2$ according to our two suggested metrics. We begin with a brief description of the alignment graph used for strings, and then proceed to explain in further detail the modifications necessary for our case.

Let S_1 and S_2 be two strings over any given alphabet, and δ_1 be a given scoring function over this alphabet. The *alignment graph* for S_1 and S_2 is a weighted directed graph with $(|S_1|+1)(|S_2|+1)$ vertices, each indexed by a distinct pair $(i,j) \in \{0,\ldots,|S_1|\} \times \{0,\ldots,|S_2|\}$. For each vertex (i,j), the alignment graph contains a directed edge from (i,j) to each of the vertices $(i,j+1)$, $(i+1,j)$, and $(i+1,j+1)$, provided these vertices exist. These edges are called the *horizontal*, *vertical*, and *diagonal* edges of (i,j) respectively, and their weights are given by $\delta_1(-,j)$, $\delta_1(i,-)$, and $\delta_1(i,j)$. In this way, the alignment graph captures the standard dynamic programming used in string alignment.

The central property of the alignment graph of S_1 and S_2 is that any path from say (i',j') to (i,j) corresponds to an alignment between $S_1[i'+1,i]$ and

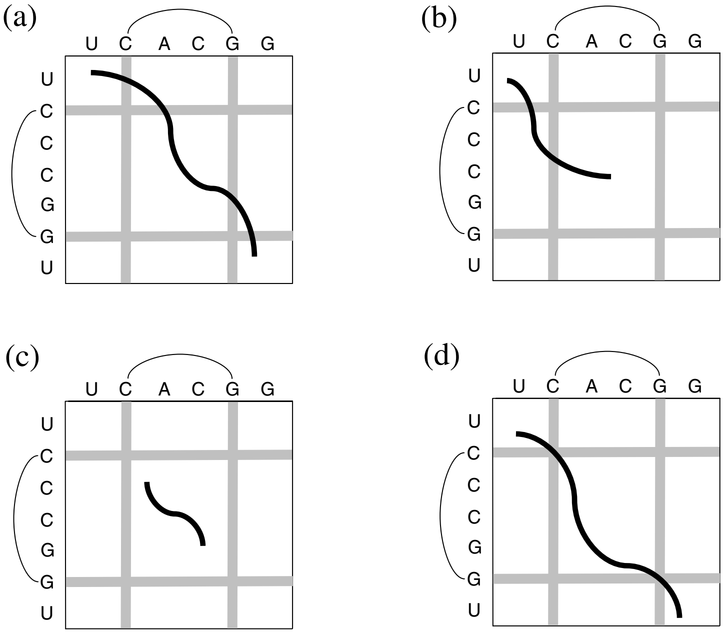


Fig. 2. Four different situations that occur when aligning RNA sequences

$S_2[j'+1, j]$ with a score which equals the total sum of weights of the edges in the path. Conversely, any alignment between $S_1[i'+1, i]$ and $S_2[j'+1, j]$ with score w has a corresponding w-weighted path in the alignment graph.

Theorem 1 ([12]). *An alignment of $S_1[i', i]$ and $S_2[j', j]$ has optimal score iff it corresponds to the heaviest path from (i', i) to (j', j).*

Let us now consider alignments of RNA sequences. The main difference when aligning RNA sequences is that now we must consider arcs and unpaired nucleotides as different entities which must be aligned separately. There are four different cases that we should each address accordingly:

- *Case 1.* Corresponds to arc deletions and is depicted in Figure 2(a). Note that the path in the figure deletes the left and right endpoints in both arcs of the RNAs, and therefore it corresponds to deleting the two arcs. Note that this would also be the case even if the path had passed through the diagonal edge that corresponds to aligning the right endpoints of the arcs.
- *Case 2.* Corresponds to alignments which break arcs and is depicted in Figure 2(b). Such alignments must be penalized accordingly.
- *Case 3.* Corresponds to alignments in areas which do not contain arcs. In contrast to the previous case, these types of alignments do not break any arcs, and therefore they should not be penalized.
- *Case 4.* Corresponds to alignments which align arcs against each other.

Note that there are also alignments which combine the first two cases, by deleting one arc and breaking the other.

We now turn to describe the necessary modifications for adapting the alignment graph to RNA sequence-structure alignment. We begin by focusing on the first three cases in the example above. The last case will be dealt with separately. For a given $i \in \{1, \ldots, n\}$, we refer to the set of all edges connecting a pair of vertices in $\{(i-1, j), (i, j) \mid 0 \leq j \leq m\}$ as the *ith row* of the alignment graph. Hence, the ith row corresponds to all edges that represent an edit operation which involves the ith position of $\mathcal{R}_1$. The *jth column*, $j \in \{1, \ldots, n\}$, is defined symmetrically to be the set of all edges connecting pairs of vertices in $\{(i, j-1), (i, j) \mid 0 \leq j \leq m\}$.

Consider some position i in $\mathcal{R}_1$, along with the ith row corresponding to this position in the alignment graph. If i is not an arc endpoint, then no modifications are necessary on this row, since all editing operations on i are equivalent to those in strings. Otherwise, when i is an endpoint, we wish to model the three cases discussed above. For this, we first remove all diagonal edges. This is done to ensure that i is not aligned against any position in $\mathcal{R}_2$, as we are only concerned with arc deletions at the moment. Following this, we set the weights of all vertical edges to the penalty of breaking the arc of which i is an endpoint of. If i a left endpoint, we set these weights to $\delta_2(i, -, -, -)$, and otherwise we set them to $\delta_2(-, i, -, -)$. This takes care of the second case described above. All other edges in the row, *i.e.* the horizontal edges, are left untouched. For a position j in $\mathcal{R}_2$ the modifications are symmetric. Here the weights of the horizontal edges are modified, while the vertical edges remain unmodified. We mention that all our modifications could be done on the scoring scheme (by adding additional letters to the alphabet which represent different types of arc endpoints) rather then on the alignment graph.

After applying the above modifications to each column and row which corresponds to arc endpoints, we obtain the *grid part* of our adapted alignment graph.

Lemma 1. *Paths in the grid part are in bijective correspondence with alignments of consecutive subsequences of $\mathcal{R}_1$ and $\mathcal{R}_2$ in which all arcs are deleted.*

What is left now, is to take care of alignments which align arcs against each other, *i.e.* the last case in the example above. Consider a path, as in Figure 2(d), that corresponds to an alignment which aligns $(i', i) \in A_1$ against $(j', j) \in A_2$. This path must pass through the nodes $(i'-1, j'-1)$ and (i, j) in the alignment graph (the two intersections of the shaded rows and columns). This means that this path consists of a prefix which ends at $(i'-1, j'-1)$, a middle part from $(i'-1, j'-1)$ to (i, j), and a suffix which starts at (i, j). As was explained in Section 2.1, the optimal score of the middle part is given by $OPT_{\delta}^{arc}(\mathcal{R}_1[i', i], \mathcal{R}_2[j', j])$, and it is computed in the preprocessing step. Therefore, if this path is optimal, its weight equals $OPT_{\delta}^{arc}(\mathcal{R}_1[i', i], \mathcal{R}_2[j', j])$ plus the combined weights of the suffix and prefix. We represent the middle part of any optimal path that aligns (i', i) against (j', j) by adding a single edge from $(i'-1, j'-1)$ to (i, j) in the alignment graph, and setting its weight to $OPT_{\delta}^{arc}(\mathcal{R}_1[i', i], \mathcal{R}_2[j', j])$ accordingly. We refer to this new edge as a *shortcut* edge, and we add such shortcut edges for each pair of arcs in $\mathcal{R}_1$ and $\mathcal{R}_2$.

Theorem 2. *An alignment of $\mathcal{R}_1[i'+1,i]$ and $\mathcal{R}_2[j'+1,j]$ is optimal iff it corresponds to the heaviest path from (i',i) to (j',j) in the alignment graph of $\mathcal{R}_1$ and $\mathcal{R}_2$.*

5 Local Alignment Algorithms

We next describe two algorithms for computing local similarity of $\mathcal{R}_1$ and $\mathcal{R}_2$ according to the two metrics defined in Section 3. For simplicity, we focus only on computing the score of an optimal alignment rather than computing an actual alignment. One can easily obtain the latter within the same time and space bounds in both algorithms.

As a consequence of Theorem 2, computing optimal local alignments of $\mathcal{R}_1$ and $\mathcal{R}_2$ reduces to computing locally optimal paths in the alignment graph of $\mathcal{R}_1$ and $\mathcal{R}_2$. Therefore, both our algorithms initially construct the alignment graph of $\mathcal{R}_1$ and $\mathcal{R}_2$, and then perform all computations on this graph. For any edge in the alignment graph, from say (i',j') to (i,j), we let $w((i',j'),(i,j))$ denote the weight of the edge. For any vertex (i,j) in the graph, we let $N_{in}(i,j)$ denote the set of vertices with an edge to (i,j), that is, the set of *in-neighbors* of (i,j).

5.1 Standard Local Alignment Algorithm

For computing the standard local alignment score of $\mathcal{R}_1$ and $\mathcal{R}_2$, we define $s(i,j)$ to be the weight of the heaviest path, including the empty one, that ends at vertex (i,j). Note that by Theorem 2, this value equals the highest scoring alignment achievable by any pair of consecutive substrings $\mathcal{R}_1[i',i]$ and $\mathcal{R}_2[j',j]$ with $i' \in \{1,\ldots,i\}$ and $j' \in \{1,\ldots,j\}$. Therefore, the maximum $s(i,j)$ over all $(i,j) \in \{1,\ldots,n\} \times \{1,\ldots,m\}$ equals the standard local alignment score of $\mathcal{R}_1$ and $\mathcal{R}_2$.

Lemma 2. *The recursion below correctly computes $s(i,j)$:*

$$s(i,j) = \max \begin{cases} s(i',j') + w((i',j'),(i,j)) \ \textit{where}\ (i',j') \in N_{in}(i,j) \\ 0 \end{cases}$$

Time Complexity: Using standard dynamic programming, once the alignment graph of $\mathcal{R}_1$ and $\mathcal{R}_2$ is constructed, we can compute $s(i,j)$ for every $i \in \{1,\ldots,n\}$ and $j \in \{1,\ldots,m\}$ in $\mathcal{O}(nm)$ time, since the in-degree of every vertex is at most three. The preprocessing step takes $\mathcal{O}(m^2 n \lg n)$ time [14], and therefore our suggested algorithm solves the standard local alignment problem in $\mathcal{O}(m^2 n \lg n + nm) = \mathcal{O}(m^2 n \lg n)$ time.

5.2 Normalized Local Alignment Algorithm

We next present an algorithm for computing the normalized local alignment score of $\mathcal{R}_1$ and $\mathcal{R}_2$. This algorithm works by computing all optimal (in terms of length) local alignments of any possible score, and which end at any possible pair of positions in $\mathcal{R}_1$ and $\mathcal{R}_2$. For this purpose, it is convenient to slightly abuse the graph-theoretic notion of path lengths, and define the *length* of any path from (i',j') to (i,j) in the alignment graph as the value $\Delta((i',j'),(i,j)) = j'-j+i'-i$ rather than the number of edges in this path. In other words, the length of any path from (i',j') to (i,j) is defined to be the combined lengths of the two consecutive subsequences $\mathcal{R}_1[i'+1,i]$ and $\mathcal{R}_2[j'+1,j]$.

For computing the normalized local alignment score of $\mathcal{R}_1$ and $\mathcal{R}_2$, we define $s^k(i,j)$ to be the length of the shortest path that ends at vertex (i,j) and has weight equal to k, or ∞ if no such path exists. Note that the normalized score of such a path is given by $k/s^k(i,j)$.

Lemma 3. *The recursion below correctly computes* $s^k(i,j)$*:*

$$s^k(i,j) = \min \left\{ s^{k'}(i',j') + \Delta((i',j'),(i,j)) \;\middle|\; \begin{array}{l} (i',j') \in N_{in}(i,j), \\ k' = k - w((i',j'),(i,j)) \end{array} \right\}.$$

Notice that if our scoring scheme contains values which are not constant, we could use a similar recursion in which the roles of lengths and scores are reversed. This is done by defining $s^k(i,j)$ to be the weight of the heaviest length k path that ends at vertex (i,j). The advantage of defining $s^k(i,j)$ as in the presentation above is that one can stop the computation once a satisfying solution is found.

Time Complexity: Let δ_{min} and δ_{max} be the minimum and maximum score of a single edit operation in our given scoring scheme $\delta = (\delta_1, \delta_2)$. Notice that $|(m+n)\delta_{max}|$ and $-|(m+n)\delta_{min}|$ are upper and lower bounds on the global alignment score of $\mathcal{R}_1$ and $\mathcal{R}_2$. Set $\hat{k} = |(m+n)\delta_{max}|$ and $\check{k} = -|(m+n)\delta_{min}|$. Using standard dynamic programming, once the alignment graph of $\mathcal{R}_1$ and $\mathcal{R}_2$ is constructed, we can compute $s^k(i,j)$ for every $i \in \{1,\ldots,n\}$, $j \in \{1,\ldots,m\}$, and $k \in \{\check{k},\ldots,\hat{k}\}$, in $\mathcal{O}(nm(\hat{k}-\check{k}))$ time, which is $\mathcal{O}(n^2m)$ assuming δ_{max} and δ_{min} are both constants. Also, The bounds on k follow from the integrality of the scoring scheme. Note that if either δ_{max} or δ_{min} are not constants, or if the scoring scheme is not integral, we can use the alternative definition of $s^k(i,j)$ given above to obtain the same complexity bounds. With a preprocessing stage of $\mathcal{O}(m^2 n \lg n)$ time [14], our suggested algorithm therefore solves the normalized local alignment problem in $\mathcal{O}(m^2 n \lg n + n^2 m)$ time.

References

1. J. Alliali and M-F. Sagot. A new distance for high level RNA secondary structure comparison. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 2(1):4–14, 2005.

2. A.N. Arslan, Ö. Eğecioğlu, and P.A. Pevzner. A new approach to sequence alignment: normalized sequence alignment. *Bioinformatics*, 17(4):327–337, 2001.
3. R. Backofen and S. Will. Local sequence-structure motifs in RNA. *Journal of Bioinformatics and Computational Biology (JBCB)*, 2(4):681–698, 2004.
4. P. Chartrand, X-H. Meng, R.H. Singer, and R.M. Long. Structural elements required for the localization of ASH1 mRNA and of a green fluorescent protein reporter particle *in vivo*. *Current Biology*, 9:333–336, 1999.
5. S. Chen, Z. Wang, and K. Zhang. Pattern matching and local alignment for RNA structures. In *international conference on Mathematics and Engineering Techniques in Medicine and Biological Sciences (METMBS)*, pages 55–61, 2002.
6. J. Couzin. Breakthrough of the year. Small RNAs make big splash. *Science*, 298(5602):2296–2297, 2002.
7. K.M. Currey, D. Sasha, B.A. Shapiro, J. Wang, and K. Zhang. An algorithm for finding the largest approxemately common substructure of two trees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(8):889–895, 1998.
8. E.D. Demaine, S. Mozes, B. Rossman, and O. Weimann. An $O(n^3)$-time algorithm for tree edit distance. Technical Report arXiv:cs.DS/0604037, Cornell University, 2006.
9. N. Efraty and G.M. Landau. Sparse normalized local alignment. In *15th Combinatorial Pattern Matching conference (CPM)*, pages 333–346, 2004.
10. R. Giegerich, M. Höchsmann, S. Kurtz, and T. Töller. Local similarity in RNA secondary structures. In *Computational Systems Bioinformatics (CSB)*, pages 159–168, 2003.
11. V. Guignon, C. Chauve, and S. Hamel. An edit distance between RNA stem-loops. In *12th Symposium on String Processing and Information Retrieval (SPIRE)*, pages 335–347, 2005.
12. D. Gusfield. *Algorithms on Strings, Trees, and Sequences. Computer Science and Computational Biology*. Press Syndicate of the University of Cambridge, 1997.
13. T. Jiang, G. Lin, B. Ma, and K. Zhang. A general edit distance between RNA structures. *Journal of Computational Biology*, 9(2):371–88, 2002.
14. P.N. Klein. Computing the edit-distance between unrooted ordered trees. In *6th European Symposium on Algorithms (ESA)*, pages 91–102, 1998.
15. P.B. Moore. Structural motifs in RNA. *Annual review of biochemistry*, 68:287–300, 1999.
16. D. Shasha and K. Zhang. Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal on Computing*, 18(6):1245–1262, 1989.
17. T.F. Smith and M.S. Waterman. The identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.
18. J. Wang and K. Zhang. Identiffying consensus of trees through alignment. *Information Sciences*, 126:165–189, 2000.
19. M. Zuker. On finding all suboptimal foldings of an RNA molecule. *Science*, 244(4900):48–52, 1989.
20. M. Zuker and P. Stiegler. Optimal computer folding of large RNA sequences using thermodynamics and auxiliary information. *Nucleic Acids Research*, 9(1):133–148, 1981.

An $O(n^{3/2}\sqrt{\log(n)})$ Algorithm for Sorting by Reciprocal Translocations

Michal Ozery-Flato and Ron Shamir

School of Computer Science, Tel-Aviv University, Tel Aviv 69978, Israel
{ozery, rshamir}@post.tau.ac.il

Abstract. We prove that sorting by reciprocal translocations can be done in $O(n^{3/2}\sqrt{\log(n)})$ for an n-gene genome. Our algorithm is an adaptation of the Tannier et. al algorithm for sorting by reversals. This improves over the $O(n^3)$ algorithm for sorting by reciprocal translocations given by Bergeron et al.

1 Introduction

In this paper we study the problem of sorting by reciprocal translocations (abbreviated SRT). *Reciprocal translocations* exchange non-empty tails between two chromosomes. Given two multi-chromosomal genomes A and B, the problem of SRT is to find a shortest sequence of reciprocal translocations that transforms A into B. SRT that was first introduced by Kececioglu and Ravi [7] and was given a polynomial time algorithm by Hannenhalli [3]. Bergeron, Mixtacki and Stoye [2] pointed to an error in Hannenhalli's proof of the reciprocal translocation distance formula and consequently in Hannenhalli's algorithm. They presented a new $O(n^3)$ algorithm, which to the best of our knowledge, is the only extant correct algorithm for SRT[1].

Reversals (or inversions) reverse the order and the direction of transcription of the genes in a segment inside a chromosome. Given two uni-chromosomal genomes π_1 and π_2, the problem of sorting by reversals (abbreviated SBR) is to find a shortest sequence of reversals that transforms π_1 into π_2. Tannier, Bergeron and Sagot [9] presented an elegant algorithm for SBR that can be implemented in $O(n^{3/2}\sqrt{log(n)})$ using a clever data structure by Kaplan and Verbin [6]. This is currently the fastest algorithm for SBR.

In this paper we prove that SRT can be solved in $O(n^{3/2}\sqrt{\log(n)})$ for an n-gene genome. Our algorithm for SRT is similar to the algorithm by Tannier et al [9] for SBR. The paper is organized as follows. The necessary preliminaries are given in Sect. 2. In Sect. 3 we give a linear time reduction from SRT to a simpler restricted subproblem. In Sect. 4 we prove the main theorem and present the algorithm for the restricted subproblem. In Sect. 5 we describe an $O(n^{3/2}\sqrt{\log(n)})$

[1] Li et al. [8] gave a linear time algorithm for computing the reciprocal translocation distance (without producing a shortest sequence). Wang et al. [10] presented an $O(n^2)$ algorithm for SRT. However, the algorithms in [8, 10] rely on an erroneous theorem of Hannenhali and hence provide incorrect results in certain cases.

M. Lewenstein and G. Valiente (Eds.): CPM 2006, LNCS 4009, pp. 258–269, 2006.

implementation of the algorithm. Due to space constraints, most proofs are omitted.

2 Preliminaries

This section provides a basic background for the analysis of SRT. It follows to a large extent the nomenclature and notation of [3, 5, 2]. In the model we consider, a *genome* is a set of chromosomes. A *chromosome* is a sequence of genes. A *gene* is identified by a positive integer. All genes in the genome are distinct. When it appears in a genome, a gene is assigned a sign of plus or minus. For example, the following genome consists of 8 genes in two chromosomes: $A = \{(1, -3, -2, 4, -7, 8), (6, 5)\}$. The *reverse* of a sequence of genes $I = (x_1, \ldots, x_l)$ is $-I = (-x_l, \ldots, -x_1)$. A *reversal* reverses a segment of genes inside a chromosome. Two chromosomes, X and Y, are *identical* if either $X = Y$ or $X = -Y$. Therefore, *flipping* chromosome X into $-X$ does not affect the chromosome it represents.

A *signed permutation* $\pi = (\pi_1, \ldots, \pi_n)$ is a permutation on the integers $\{1, \ldots, n\}$, where a sign of plus or minus is assigned to each number. If A is a genome with the set of genes $\{1, \ldots, n\}$ then any concatenation π_A of the chromosomes of A is a signed permutation of size n.

Let $X = (X_1, X_2)$ and $Y = (Y_1, Y_2)$ be two chromosomes, where X_1, X_2, Y_1, Y_2 are sequences of genes. A *translocation* cuts X into X_1 and X_2 and Y into Y_1 and Y_2 and exchanges segments between the chromosomes. It is called *reciprocal* if X_1,X_2, Y_1 and Y_2 are all non-empty. There are two ways to perform a translocation on X and Y. A *prefix-suffix* translocation switches X_1 with Y_2 resulting in: $(\underline{X_1}, X_2), (Y_1, \underline{Y_2}) \Rightarrow (-Y_2, X_2), (Y_1, -X_1)$. A *prefix-prefix* translocation switches X_1 with Y_1 resulting in: $(\underline{X_1}, X_2), (\underline{Y_1}, Y_2) \Rightarrow (Y_1, X_2), (X_1, Y_2)$. Note that we can mimic a prefix-prefix (respectively, prefix-suffix) translocation by a flip of one of the chromosomes followed by a prefix-suffix (respectively, prefix-prefix) translocation. As was demonstrated by Hannenhalli and Pevzner [4], a translocation on A can be simulated by a reversal on π_A in the following way: $(\ldots, X_1, \underline{X_2, \ldots, Y_1}, Y_2, \ldots) \Rightarrow (\ldots, X_1, \underline{-Y_1, \ldots, -X_2}, Y_2, \ldots)$. The type of translocation depends on the relative orientation of X and Y in π_A (and not on their order): if the orientation is the same, then the translocation is prefix-suffix, otherwise it is prefix-prefix. The segment between X_2 and Y_1 may contain additional chromosomes that are flipped and thus unaffected.

For a chromosome $X = (x_1, \ldots, x_k)$ define $Tails(X) = \{x_1, -x_k\}$. Note that flipping X does not change $Tails(X)$. For a genome A_1 define $Tails(A_1) = \bigcup_{X \in A_1} Tails(X)$. For example: $Tails(\{(1, -3, -2, 4, -7, 8), (6, 5)\}) = \{1, -8, 6, -5\}$. Two genomes A_1 and A_2 are *co-tailed* if $Tails(A_1) = Tails(A_2)$. In particular, two co-tailed genomes have the same number of chromosomes. Note that if A_2 was obtained from A_1 by performing a reciprocal translocation then $Tails(A_2) = Tails(A_1)$. Therefore, SRT is defined only for genomes that are co-tailed. For the rest of this paper the word "translocation" refers to a reciprocal translocation and we assume that the given genomes, A and B, are co-tailed.

2.1 The Cycle Graph

Let N be the number of chromosomes in A (equivalently, B). We shall always assume that both A and B contain genes $\{1, \ldots, n\}$. The *cycle graph* of A and B, denoted $G(A, B)$, is defined as follows. The set of vertices is $\bigcup_{i=1}^{n} \{i^0, i^1\}$. For every two genes, i and j, where j immediately follows i in some chromosome of A (respectively, B) add a black (respectively, grey) edge $(i, j) \equiv (out(i), in(j))$, where $out(i) = i^1$ if i has a positive sign in A (respectively, B) and otherwise $out(i) = i^0$, and $in(j) = j^0$ if j has a positive sign in A (respectively, B) and otherwise $in(j) = j^1$. There are $n - N$ black edges and $n - N$ grey edges in $G(A, B)$. A grey edge (i, j) is *external* if the genes i and j belong to different chromosomes of A, otherwise it is *internal*.

Every vertex in $G(A, B)$ has degree 2 or 0, where vertices of degree 0 (isolated vertices) belong to $Tails(A)$ (equivalently, $Tails(B)$). Therefore, $G(A, B)$ is uniquely decomposed into cycles with alternating grey and black edges. An *adjacency* is a cycle with two edges.

2.2 The Overlap Graph

Place the vertices of $G(A, B)$ along a straight line according to their order in π_A. Now, every grey edge can be associated with an interval of vertices of $G(A, B)$. Two grey edges *overlap* if the intersection of their intervals is not empty but none contains the other. The *overlap graph* of A and B w.r.t. π_A, denoted $OV(A, B, \pi_A)$, is defined as follows. The set of vertices is $\{(i_1, i_2) : (i_1, i_2)$ is a grey edge in $G(A, B)\}$. Two vertices are connected if their corresponding grey edges overlap. We shall use the word "component" for a connected component of the overlap graph. The set of components of $OV(A, B, \pi_A)$ can be computed in linear time using an algorithm by Bader, Moret and Yan [1].

A vertex in an overlap graph is *external* if its corresponding edge is external, otherwise it is *internal*. Note that the internal/external state of a vertex in $OV(A, B, \pi_A)$ does not depend on π_A (the partition of the chromosomes is known from A). A component of $OV(A, B, \pi_A)$ is *external* if at least one of the vertices in it is external, otherwise it is *internal*. A component is *trivial* if it corresponds to an adjacency and hence always internal. A vertex in the overlap graph is *oriented* if its corresponding edge connects two genes with different signs in π_A, otherwise it is *unoriented*.

The *span* of a component M is an interval of genes $I(M) = [i, j] \subset \pi_A$, where $i = arg \min\{\pi_A^{-1}(i_1), \pi_A^{-1}(i_2) \mid (i_1, i_2) \in M\}$ and $j = arg \max\{\pi_A^{-1}(j_1), \pi_A^{-1}(j_2) \mid (j_1, j_2) \in M\}$. Clearly, $I(M)$ is independent of π_A iff M is internal. Therefore, the set of internal components in $OV(A, B, \pi_A)$ is independent of π_A.

2.3 The Forest of Internal Components

$(M_1, \ldots, M_t)$ is a *chain* of components if $I(M_j)$ and $I(M_{j+1})$ overlap in exactly one gene for $j = 1, .., t - 1$. For a chain of components $C = (M_1, \ldots, M_t)$ define $I(C) = \bigcup_{j=1}^{t} I(M_j)$. The *forest of internal components*, denoted $F(A, B)$, is defined as follows. The vertices of $F(A, B)$ are *(i)* the non-trivial internal components and *(ii)* every maximal chain of internal components that contains at least one non-trivial component. Let M and C be two vertices in $F(A,B)$ where M corresponds to a component and C to a chain. $M \rightarrow C$ is an edge of $F(A,B)$ if $M \in C$. $C \rightarrow M$ is an edge of $F(A,B)$ if $I(C) \subset I(M)$ and $I(M)$ is minimal. We will refer to a component that is a leaf in $F(A,B)$ as simply a *leaf.*

2.4 The Reciprocal Translocation Distance

Let $c(A,B)$ denote the number of cycles in $G(A,B)$. Let $T(A,B)$ and $L(A,B)$ denote the number of trees and leaves in $F(A,B)$ respectively. Obviously $T(A,B) \leq L(A,B)$. Define $f_{\mathrm{r}}(A,B) = \begin{cases} 2 & \text{if } T(A,B)=1 \text{ and } L(A,B) \text{ is even} \\ 1 & \text{if } L(A,B) \text{ is odd} \\ 0 & \text{otherwise } (T(A,B) \neq 1 \text{ and } L(A,B) \text{ is even}) \end{cases}$

Theorem 1. *[2, 3] The reciprocal translocation distance between A and B is $d_{\mathrm{r}}(A,B) = n - N - c(A,B) + L(A,B) + f_{\mathrm{r}}(A,B)$*

Let Δc denote the change in the number of cycles after performing a translocation on A. Then $\Delta c \in \{-1,0,1\}$ [3]. A translocation is *proper* if $\Delta c = 1$ and *bad* if $\Delta c = -1$. A translocation ρ is *valid* if $d_{\mathrm{r}}(A \cdot \rho, B) = d_{\mathrm{r}}(A,B) - 1$. A translocation is *safe* if it does not create any new non-trivial internal component. As was demonstrated by Bergeron et al. [2] a safe translocation might be invalid if the set of leaves is not empty. However, if there are no leaves, then a safe proper translocation is necessarily valid. We define SRTNL as a special case of SRT when there are no leaves (i.e. $T(A,B) = L(A,B) = 0$).

3 A Linear Reduction of SRT to SRTNL

A translocation is bad iff it cuts two black edges, b_1 and b_2, that belong to different cycles [3]. Note that there are two bad translocations, either prefix-prefix or suffix-prefix, cutting the black edges b_1 and b_2. A leaf M is *eliminated* by performing a (bad) translocation that cuts one black edge incident to a grey edge in M and one black edge in another chromosome of A. Observe that in this case all the ancestors of M in $F(A,B)$ are eliminated as well. Let $L(X)$ denote the number of leaves in chromosome X. Let $N^{\mathrm{L}}(A,B)$ denote the number of chromosomes of A containing at least one leaf. A translocation ρ is *separating* if $N^{\mathrm{L}}(A,B) = 1$ but $N^{\mathrm{L}}(A \cdot \rho, B) > 1$. It is easy to see that a translocation is separating only if it cuts a black edge between two leaves.

Lemma 1. *[2] There is a sequence of safe proper translocations that sorts all external components (i.e., after performing the sequence, every edge in an external component becomes an adjacency).*

Lemma 2. *[2] Let $S = (\rho_1, \ldots, \rho_k)$ be a sequence of safe proper translocations that sorts all external components. If $N^{\mathrm{L}}(A,B) = 1$ but $T(A,B) > 1$ then S contains a separating translocation ρ_l. Moreover, $S' = \rho_1, \ldots, \rho_l$ is a sequence of valid translocations and $N^{\mathrm{L}}(A \cdot \rho_1 \cdots \rho_l, B) > 1$.*

Lemma 3. *[3] Suppose that the following conditions are satisfied: (i)* $N^{\mathrm{L}}(A, B) = 1$*, (ii)* $L(A, B) \geq 2$*, and (iii) either* $L(A, B)$ *is odd or* $T(A, B) = 1$*. Let* ρ *be a (prefix-prefix) translocation that eliminates the second leaf from the left in* A*. Then* ρ *is valid and if* $L(A \cdot \rho, B) \geq 2$ *then* $N^{\mathrm{L}}(A \cdot \rho, B) \geq 2$*.*

Lemma 4. *All the bad translocations in the algorithm in Fig. 1 are valid.*

```
(1) if $N^{\mathrm{L}} = 1$ and $L \geq 2$ :
    (a) if $T > 1$ and $L$ is even:
          (i) Solve SRTNL on the set of external components until $N^{\mathrm{L}} \neq 1$.
    (b) else: eliminate the second leaf from the left by a prefix-prefix translocation.
(2) Let $Q_1$ be a queue of the chromosomes containing exactly one leaf.
    Let $Q_2$ be a queue of the chromosomes containing more than one leaf.
(3) while $L > 0$ (Invariant: $L=1$ or $N^{\mathrm{L}} \geq 2$)
    (a) if $L = 1$: eliminate the single leaf by a prefix-prefix translocation.
    (b) else:
          (i) For $i = 1, 2$
              1. if $Q_2 \neq \emptyset$ then $X_i \leftarrow pop(Q_2)$, otherwise $X_i \leftarrow pop(Q_1)$.
              2. if $L(X_i) = 2$ then $l_i \leftarrow$ the second leaf from the left in $X_i$,
                 otherwise $l_i \leftarrow$ the single leaf in $X_i$.
         (ii) Eliminate $l_1$ and $l_2$ by a prefix-prefix translocation.
        (iii) For $i = 1, 2$: if $L(X_i) > 1$ then $push(X_i, Q_2)$. if $L(X_i) = 1$ then
              $push(X_i, Q_1)$.
(4) Solve SRTNL on $A$.
```

Fig. 1. A generic algorithm for solving SRT using an algorithm for SRTNL

The generic algorithm in Fig. 1 and the preceding lemmas imply:

Theorem 2. *SRT is linearly reducible to SRTNL.*

4 An Algorithm for SRTNL

In this section we present an algorithm for SRTNL. We first define an extension of the overlap graph and then prove the algorithm's correctness. Fig. 3 provides examples of the graphs used.

4.1 The Overlap Graph with Chromosomes

A chromosome X and an edge e *overlap* if X contains exactly one of the two endpoints of e. Hence, if edge e overlaps chromosome X of A then e must be an external grey edge. We define the *overlap graph with chromosomes*, $OVCH(A, B, \pi_A)$ based on $OV(A, B, \pi_A)$ as follows. We add to $OV(A, B, \pi_A)$ a vertex for each chromosome of A. In order to prevent confusion, we will refer to the new vertices as "chromosomes" and reserve the word "vertex" for the original vertices of $OV(A, B, \pi_A)$ (that correspond to edges). A vertex and a chromosome are connected if the corresponding grey edge overlaps the chromosome. There are no edges between chromosomes.

Let $H = OVCH(A, B, \pi_A)$ and let v be any vertex in H. Denote by $N(v) \equiv N(v, H)$ the set of vertices that are neighbors of v, including v itself (but not including chromosome neighbors). Denote by $CH(v) \equiv CH(v, H)$ the set of chromosomes that are neighbors of v in H. Hence if v is external then $|CH(v)| = 2$, otherwise $CH(v) = \emptyset$.

Every external grey edge e defines one proper translocation that cuts the black edges incident to e. (Out of the two possibilities of prefix-prefix or prefix-suffix translocations, exactly one would be proper). For an external vertex v denote by $\rho(v)$ the proper translocation that the corresponding grey edge defines on A. Two external vertices v_1 and v_2 in H are *equivalent* if they define the same translocation, i.e. $\rho(v_1) \equiv \rho(v_2)$. Let $H \cdot \rho(v) = OVCH(A \cdot \rho(v), B, \pi_A)$. Given two sets S_1 and S_2 define $S_1 \bigoplus S_2 = (S_1 \bigcup S_2) \setminus (S_1 \bigcap S_2)$.

Lemma 5. *Let v be an oriented external vertex in H. Then $H \cdot \rho(v)$ is obtained from H by the following operations. (i) Complement the subgraph induced by $N(v)$ and flip the orientation of every vertex in $N(v)$. (ii) For every vertex $u \in N(v)$ such that the endpoints of u and v share at least one common chromosome, update the edges between u and $CH(u) \bigcup CH(v)$ such that $CH(u) = CH(u) \bigoplus CH(v)$.*

Two overlap graphs with chromosomes are *equivalent* if one can be obtained from the other by a sequence of chromosome flips. For a chromosome X let $\rho(X)$ denote a flip of chromosome X in π_A. Let $H \cdot \rho(X) = OVCH(A, B, \pi_A \cdot \rho(X))$.

Lemma 6. *$H \cdot \rho(X)$ is obtained from H by complementing the subgraph induced by the set $\{u : X \in CH(u)\}$ and flipping the orientation of every vertex in it.*

4.2 The Main Theorem and Algorithm

In this section we give the main theorem and algorithm. Our algorithm is formally very similar to the algorithm for SBR presented in [9]. Instead of performing reversals on oriented edges in [9], we perform translocations on external edges. Despite of the great similarity between the algorithms our validity proof is completely new. We analyze an overlap graph with chromosomes of a multi-chromosomal genome, while [9] analyze the overlap graph of a uni-chromosomal genome. Like [9], we perform operations defined by oriented vertices (i.e. translocations). However, in our case these vertices must also be external. If an external vertex is unoriented, we can turn it into an oriented vertex by a flip of a chromosome. Hence, we consider two types of operations in our analysis.

A sequence of vertices $S = (v_1, \ldots, v_k)$ from H is *legal* if v_j is external in $H \cdot \rho(v_1) \cdots \rho(v_{j-1})$ for $j = 1, .., k$. For a legal sequence S define $\rho(S) = \rho(v_1) \cdots \rho(v_k)$. A legal sequence S is *total* if $H \cdot \rho(S)$ contains only trivial components. For H_1, an overlap graph with chromosomes, let $IN(H_1)$ and $EXT(H_1)$ denote the sets of vertices that are in non-trivial internal components and external components respectively. If S is a maximal legal sequence of vertices in H then $EXT(H \cdot \rho(S)) = \emptyset$. If in addition S is not total then $IN(H \cdot \rho(S)) \neq \emptyset$.

Theorem 3. *Let $S = (v_1, \ldots, v_k)$ be a maximal legal but not total sequence of vertices in H. Let $IN = IN(H \cdot \rho(S))$. Let v_l be the first vertex in S satisfying $IN(H \cdot \rho(v_1, \ldots, v_l)) = IN$, i.e. $\rho(v_l)$ is the last unsafe translocation in $\rho(S)$. Let $S_1 = (v_1, \ldots, v_{l-1})$ and $S_2 = (v_l, \ldots, v_k)$. Then every maximal sequence of vertices $S' = (w_1, \ldots, w_m)$ in IN that satisfies (i) (S_1, S') is legal and (ii) v_l is not an adjacency in $H \cdot \rho(S_1, S')$ also satisfies: (iii) S' is not empty and (iv) (S_1, S', S_2) is a maximal legal sequence. Moreover, all the translocations in $\rho(S_2)$ are safe.*

Proof. Let $v = v_l$, $H_0 = H \cdot \rho(S_1)$ and $IN_0 = EXT(H_0) \cap IN$. Then $IN_0 \neq \emptyset$ and none of the vertices in IN_0 is equivalent to v in H_0 (otherwise it would be an adjacency in $H \cdot \rho(S)$ and hence not in IN). Hence S' is not empty. Let $A_0 = A \cdot \rho(S_1)$ and $CH(v) = \{X, Y\}$. We choose π_0 to be a concatenation of the chromosomes in A_0 in which X and Y are the first two chromosomes. We can assume w.l.o.g. that $H = OVCH(A, B, \pi_0)$, hence $H_0 = OVCH(A_0, B, \pi_0)$. For $j = 1, .., m$ let $H_j = H_0 \cdot \rho(w_1, \ldots, w_j)$. Let $IN_j = EXT(H_j) \bigcap IN$. Then for $j = 1, \ldots, m$: *(i)* $w_j \in IN_{j-1}$ and *(ii)* w_j is not equivalent to v in H_{j-1}. Let $EXT = EXT(H_0 \cdot \rho(v))$. The following conditions hold for H_j when $j = 0$ (see Fig. 4-(a)):

(1) The subgraphs of $H_j \cdot \rho(v)$ and $H_0 \cdot \rho(v)$ that are induced by EXT are equivalent.
(2) Every $w \in IN_j$ satisfies: $CH(w) = CH(v) = \{X, Y\}$.
(3) If v is oriented then $N(v) \bigcap IN = IN_j$.
(4) All the possible edges exist between $N(v) \bigcap EXT$ and IN_j.
(5) There are no edges between $IN \setminus IN_j$ and vertices outside IN.
(6) There are no edges between $EXT \setminus N(v)$ and vertices outside EXT.

We shall prove below that in H_m v is external and that all the above conditions are satisfied. The first condition ensures that (S_1, S', S_2) is legal. The rest of the conditions ensure that $H_m \cdot \rho(v)$ satisfies: *(i)* there are no external vertices in IN and *(ii)*there are no edges between EXT and vertices outside EXT. Hence (S_1, S', S_2) is maximal and every translocation in $\rho(v_{l+1}, \ldots, v_k)$ is safe. $\rho(v_l)$ is safe in H_m since S' is maximal. Therefore, all the translocations in $\rho(S_2)$ are safe.

Assume that v is external in H_j and that the all above conditions hold for a certain j. Since these conditions are true for every graph that is equivalent to H_j we can assume that v is oriented. We now prove, using an induction on j, that these conditions are satisfied for every H_i, $i \in \{1, \ldots, m\}$ in which v is external, and that v is external in H_m.

Case 1: w_{j+1} is oriented in H_j. Let $H_{j+1} = H_j \cdot \rho(w_{j+1})$ (see Fig. 4-(b)). Then $IN_{j+1} = N(v, H_j) \bigoplus N(w_{j+1}, H_j)$. $IN_{j+1} \neq \emptyset$, otherwise v is an isolated internal vertex in H_{j+1} and hence equivalent to w_{j+1} in H_j. Hence $m \geq j + 2$.

Case 1.*a*: w_{j+2} is oriented in H_{j+1}. Let $H_{j+2} = H_{j+1} \cdot \rho(w_{j+2})$ (see Fig. 4-(c)). Clearly, v is external in H_{j+2}. Let $M = N(v, H_j) \bigcap EXT$. Then $N(w_{j+2}, H_{j+1})$

$\bigcap EXT = N(w_{j+1}, H_j) \bigcap EXT = M$. Hence the subgraphs of H_{j+2} and H_j that are induced by M are identical and the first condition is satisfied in H_{j+2}.

Case 1.*b*: w_{j+2} is unoriented in H_{j+1}. Let $H'_{j+1} = H_{j+1} \cdot \rho(X)$ (H'_{j+1} and H_{j+1} are equivalent) (see Fig. 4-(d)). Hence w_{j+2} is oriented in H'_{j+1}. Note that v is an internal vertex in H'_j. Let $M' = N(w_{j+1}, H'_{j+1}) \bigcap EXT$. Let $H_{j+2} = H'_{j+1} \cdot \rho(w_{j+2})$ (see Fig. 4-(e)). v is an oriented external vertex in H_{j+2} and $N(v, H_{j+2}) \bigcap EXT = M'$. Therefore, the two subgraphs of $H_{j+2} \cdot \rho(v)$ (see Fig. 4-(f)) and H'_{j+1} (see Fig. 4-(d)) that are induced by EXT are identical. The subgraphs of H_{j+1} and $H_j \cdot \rho(v)$ that are induced by EXT are also identical. Hence, the first condition is satisfied.

Looking at Figs. 4-(c) and 4-(e) it is easy to verify that the rest of the conditions are also satisfied for H_{j+2}.

Case 2: w_{j+1} is unoriented in H_j. We define the three subsets of vertices $M_1, M_2, M_3 \subset EXT$ in H_j as follows:

(1) M_1 is the set of neighbors of w_{j+1} (equivalently, v) that are either internal or external but does not overlap chromosome X.
(2) M_2 is the set of neighbors of w_{j+1} (equivalently, v) that overlap chromosome X. Hence $M_1 \bigcup M_2 = N(v, H_j) \bigcap EXT$.
(3) M_3 is the set of vertices that overlap chromosome X but are not neighbors of w_{j+1} (equivalently, v).

For an illustration of H_j see Fig. 4-(g). Let $H'_j = H_j \cdot \rho(X)$ (see Fig. 4-(h)). In H'_j: w_{j+1} is an oriented external vertex and is not a neighbor of v. Let $H_{j+1} = H'_j \cdot \rho(w_{j+1})$ (see Fig. 4-(i)). Obviously, v remains intact in H_{j+1}. Let $H'_{j+1} = H_{j+1} \cdot \rho(X)$ (see Fig. 4-(j)). Then, the subgraphs of $H'_{j+1} \cdot \rho(v)$ (see Fig. 4-(k)) and $H_j \cdot \rho(v)$ that are induced by M_1, M_2 and M_3 are equivalent (Compare the subgraph induced by EXT in H_j in Fig. 4 (g) with the subgraph induced by EXT in $H'_{j+1} \cdot \rho(v) \cdot \rho(X)$ in Fig. 4 (l)). Hence the first condition is satisfied. Looking at Fig. 4-(i), it is easy to verify that conditions (2)-(6) hold for H_{j+1}. □

The algorithm in Fig. 2 builds a sequence of translocations by a repeated application of Theorem 3. It greedily removes external edges from an allowed subset and performs the corresponding translocations (step (2).(a)). When the allowed subset contains only internal edges, the algorithm repeats the last translocations in a reverse order (thereby cancelling them) until another edge in the allowed subset becomes external (step (2).(b)). Every translocation in the algorithm is applied at most twice and so the algorithm performs at most $2n$ translocations.

5 An $\mathbf{O(n^{3/2}\sqrt{\log(n)})}$ Time Implementation of the Algorithm

The algorithm in Fig. 2 can be implemented in $O(n^2)$ time in a relatively simple manner. We provide below an $O(n^{3/2}\sqrt{\log(n)})$ algorithm. The implementation follows closely the ideas of [6] and [9].

```
(1) Let $V$ be the set of edges in $G(A, B)$ that are in non-trivial components.
    Set $S_1 = S_2 = \emptyset$
(2) while $V \neq \emptyset$:
    (a) while there exists an external edge $v \in V$ in $G(A, B)$:
          (i) Remove $v$ from $V$.
         (ii) if $v$ is not equivalent to the first element in $S_2$:
               1. Append $\rho(v)$ to $S_1$
               2. $A \leftarrow A \cdot \rho(v)$
    (b) while all the edges in $V$ are internal:
          (i) Let $\rho$ be the last translocation in $S_1$
         (ii) Remove $\rho$ from $S_1$
        (iii) Prepend $\rho$ to $S_2$
         (iv) $A \leftarrow A \cdot \rho$
(3) return $(S_1, S_2)$
```

Fig. 2. An algorithm for SRTNL

Assume w.l.o.g. that π_B is the identity permutation. Then every grey edge is of the form $(i, i+1)$. We identify a grey edge $(i, i+1)$ by i and refer to $(i+1)$ as the *remote end* of i. The data structure we use for maintaining the genome A is as follows.

1. A doubly linked list of $O(\sqrt{\frac{n}{\log(n)}})$ blocks. We partition π_A into continuous blocks such that the size of every block is at least $\frac{1}{2}\sqrt{n\log(n)}$ and at most $2\sqrt{n\log(n)}$.
2. A balanced search tree for every block. The tree contains the edges in the block ordered by the positions of their remote ends. We use balanced trees that support split and concatenate operations in logarithmic time, such as red-black trees or 2-4 trees. We use $T[v]$ to denote the subtree rooted at v and containing all its descendants.
3. An n-array of block pointers. The i^{th} entry in the array points to the block containing i.

We add the following fields to the above data structure.

1. For each edge we keep an external-bit. If the external-bit is *on* then the edge is external, otherwise it is internal.
2. For each block we keep the following fields: *(i)* a counter of external edges in V, *(ii)* a counter of chromosomes' left tails, and *(iii)* a reverse-flag. If the reverse-flag of a block is *on* then the order and signs of the elements in the block are reversed.
3. For every subtree $T[v]$ of each block's search tree we keep the following fields in its root v: *(i)* counters of external and internal edges in V, *(ii)* a direction-flip-flag and *(iii)* an external-flip-flag. If the external-flip-flag of a vertex v is *on* then in $T[v]$ the external-bits of all the elements are flipped and the counters of internal and external elements from V exchange their values. If the direction-flip-flag of a vertex v is *on* then in $T[v]$ the order of the elements is reversed.

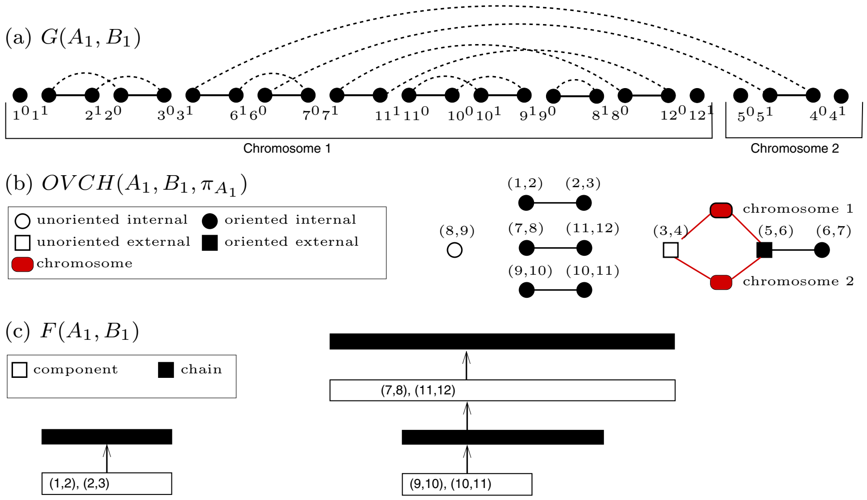


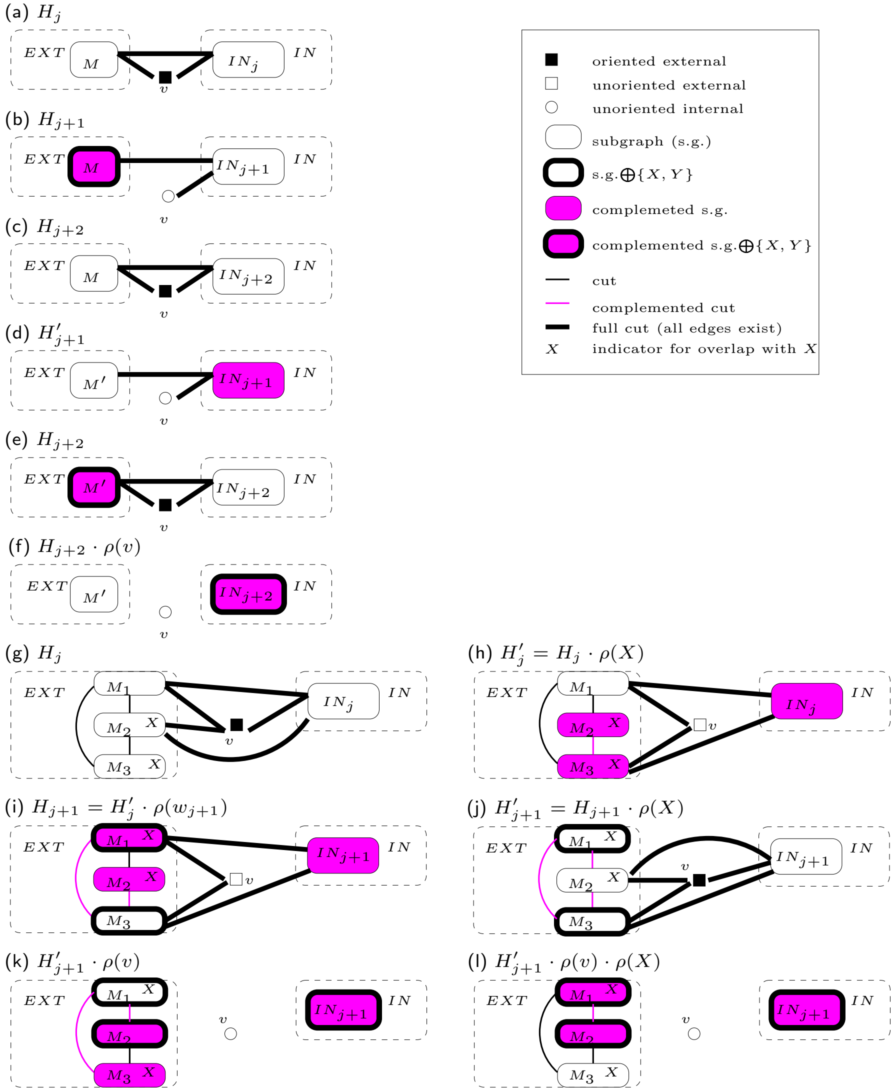
Fig. 3. Auxiliary graphs for $A_1 = \{(1, -2, 3, -6, 7, -11, 10, -9, -8, 12), (5, 4)\}$, $B_1 = \{(1, \ldots, 4), (5, \ldots, 12)\}$ ($\pi_{A_1} = (1, -2, 3, -6, 7, -11, 10, -9, -8, 12, 5, 4)$)

We can clear the direction-flip-flag of a node by reversing the order of its children and flipping the direction-flip-flag in each of them. We can clear the external-flip-flag in a node by exchanging the values of the counters of external and internal edges in V, flipping the external-flip-flag in each of its children and flipping the external-bit of the element residing at the node. One can view this procedure as "pushing down" the flags. An direction-flip-flag and an external-flip-flag that are *on* are "pushed down" whenever $T[v]$ is searched.

We implement the algorithm using the above data structures. A search for an external edge in V is done as follows. We traverse the list of blocks until we reach a block that contains external edges from V. We then search the tree of the block for an external edge i. We locate element $i+1$ (the remote end of edge i) using the n-array and a search of its block.

Let ρ be a translocation on A operating on the chromosomes $X = (X_1, X_2)$ and $Y = (Y_1, Y_2)$. Then ρ is performed in $O(\sqrt{n \log(n)})$ time as follows:

(1) Split at most six blocks so that each of the four segments X_1, X_2, Y_1 and Y_2 corresponds to a union of blocks. If ρ is a prefix-prefix translocation exchange the blocks of X_1 and Y_1. Otherwise, reverse the order and flip the reverse-flags of the blocks of X_2 and Y_1 and then exchange the blocks of X_2 and Y_1.
(2) We now have to modify the trees of each block to reflect the order and direction changes. This is done as follows. Traverse all the blocks and for each block:
    (a) Let T be the balanced search tree of the block. If ρ is a translocation on an edge i in V and i is contained in the block: decrease by 1 the counters of external edges in V of the block and of every node in T that contains i in its subtree.



(a) H_j

(b) H_{j+1}

(c) H_{j+2}

(d) H'_{j+1}

(e) H_{j+2}

(f) $H_{j+2} \cdot \rho(v)$

(g) H_j

(h) $H'_j = H_j \cdot \rho(X)$

(i) $H_{j+1} = H'_j \cdot \rho(w_{j+1})$

(j) $H'_{j+1} = H_{j+1} \cdot \rho(X)$

(k) $H'_{j+1} \cdot \rho(v)$

(l) $H'_{j+1} \cdot \rho(v) \cdot \rho(X)$

Fig. 4. Illustrations for the proof of Theorem 3

(b) Split T into at most seven subtrees such that each of the segments X_1, X_2, Y_1 and Y_2 has a corresponding subtree.

(c) If the block corresponds to a segment of X_1, X_2, Y_1 and Y_2 flip the external-flip-flag at the roots of two subtrees according to Table 1.

(d) If ρ is a prefix-prefix translocation, exchange the subtrees of X_1 and Y_1. Otherwise, exchange the subtrees of X_2 and Y_1 and flip the direction-flip-flags of both.

(e) Concatenate the seven subtrees into T.

Table 1. The subtrees for which the external-flip-flag is flipped as a function of translocation type and block type

Block	X_1	X_2	Y_1	Y_2
prefix-prefix	X_2, Y_2	X_1, Y_1	X_2, Y_2	X_1, Y_1
prefix-suffix	X_2, Y_1	X_1, Y_2	X_1, Y_2	X_2, Y_1

(3) If necessary, concatenate small blocks and split large blocks such that the size of each block is at least $\frac{1}{2}\sqrt{n\log(n)}$ and at most $2\sqrt{n\log(n)}$.

Theorem 4. *SRTNL can be solved in* $O(n^{3/2}\sqrt{\log(n)})$. □

References

1. D.A. Bader, B. M.E. Moret, and M. Yan. A linear-time algorithm for computing inversion distance between signed permutations with an experimental study. *Journal of Computational Biology*, 8(5):483–491, 2001.
2. A. Bergeron, J. Mixtacki, and J. Stoye. On sorting by translocations. *Journal of Computational Biology*, 13(2):567–578, 2006.
3. S. Hannenhalli. Polynomial algorithm for computing translocation distance between genomes. *Discrete Applied Mathematics*, 71:137–151, 1996.
4. S. Hannenhalli and P. Pevzner. Transforming men into mice (polynomial algorithm for genomic distance problems). In *Proc. FOCS'95*, pages 581–592, Los Alamitos, 1995. IEEE Computer Society Press.
5. H. Kaplan, R. Shamir, and R. E. Tarjan. Faster and simpler algorithm for sorting signed permutations by reversals. *SIAM Journal of Computing*, 29(3):880–892, 2000.
6. H. Kaplan and E. Verbin. Sorting signed permutations by reversals, revisited. *Journal of Computer and System Sciences*, 70(3):321–341, 2005.
7. J. D. Kececioglu and R. Ravi. Of mice and men: Algorithms for evolutionary distances between genomes with translocation. In *Proceedings of the 6th Annual Symposium on Discrete Algorithms*, pages 604–613, New York, NY, USA, January 1995. ACM Press.
8. G. Li, X. Qi, X. Wang, and B. Zhu. A linear-time algorithm for computing translocation distance between signed genomes. In *Proc. CPM*, pages 323–332, 2004.
9. E. Tannier, A. Bergeron, and M. Sagot. Advances on sorting by reversals. *to appear in Discrete Applied Mathematics.*
10. L. Wang, D. Zhu, X. Liu, and S. Ma. An $\mathrm{O}(\mathrm{n}^2)$ algorithm for signed translocation problem. In *Proc. APBC*, pages 349–358, 2005.

Longest Common Subsequences in Permutations and Maximum Cliques in Circle Graphs

Alexander Tiskin

Department of Computer Science, The University of Warwick,
Coventry CV4 7AL, United Kingdom
tiskin@dcs.warwick.ac.uk

Abstract. For two strings a, b, the longest common subsequence (LCS) problem consists in comparing a and b by computing the length of their LCS. In a previous paper, we defined a generalisation, called "the all semi-local LCS problem", for which we proposed an efficient output representation and an efficient algorithm. In this paper, we consider a restriction of this problem to strings that are permutations of a given set. The resulting problem is equivalent to the all local longest increasing subsequences (LIS) problem. We propose an algorithm for this problem, running in time $O(n^{1.5})$ on an input of size n. As an interesting application of our method, we propose a new algorithm for finding a maximum clique in a circle graph on n nodes, running in the same asymptotic time $O(n^{1.5})$. Compared to a number of previous algorithms for this problem, our approach presents a substantial improvement in worst-case running time.

1 Introduction

Given two strings a, b of lengths m, n respectively, the longest common subsequence (LCS) problem consists in comparing a and b by computing the length of their LCS. In [14], we defined a generalisation, called "the all semi-local LCS problem", where each string is compared against all substrings of the other string, and all prefixes of each string are compared against all suffixes of the other string. In the same paper we introduced a relatively simple geometric framework, allowing to represent the problem's output by a data structure of size $O(m+n)$, and to query an individual output length efficiently. We also proposed an efficient all semi-local LCS algorithm.

In this paper, we consider an important special case of string comparison, where each of the strings a, b consists of distinct characters. Without loss of generality, we may assume that $m = n$, and that both strings are permutations of a given totally ordered set of size n. The all semi-local LCS problem on permutations is equivalent to finding the length of the longest increasing subsequence (LIS) in every substring of a given string. We propose an algorithm for this problem, running in time $O(n^{1.5})$.

A related problem of computing the *complete* LIS in every substring of a *fixed size* is studied in papers [1, 6]. In particular, paper [6] gives an algorithm that runs in time proportional to the size of the output (i.e. the combined length of

M. Lewenstein and G. Valiente (Eds.): CPM 2006, LNCS 4009, pp. 270–281, 2006.

all the output LIS), which can be as high as $\Theta(n^2)$. In contrast, our algorithm only computes the lengths instead of complete LIS; however, this is done for substrings of *every possible size*.

A circle graph is defined as an intersection graph of a set of chords in a circle. It has long been known that the maximum clique problem on a circle graph on n nodes is solvable in polynomial time [9]. The best existing algorithms for this problem [13, 10, 12, 3] run in time $O(n^2)$ when the graph is dense. As an interesting application of our method, we propose a new algorithm for finding a maximum clique in a circle graph on n nodes, running in time $O(n^{1.5})$. This is a substantial improvement in running time for dense circle graphs.

2 Problem Statement and Notation

We consider strings of characters from a fixed finite alphabet, denoting string concatenation by juxtaposition. Given a string, we distinguish between its contiguous *substrings*, and not necessarily contiguous *subsequences*. Special cases of a substring are *a prefix* and *a suffix* of a string. For two strings $a = \alpha_1\alpha_2 \ldots \alpha_m$ and $b = \beta_1\beta_2 \ldots \beta_n$ of lengths m, n respectively, the *longest common subsequence (LCS) problem* consists in computing the LCS length of a and b, and the *longest increasing subsequence (LIS) problem* consists in computing the LIS length of a (assuming a given total order on the alphabet characters).

We consider a generalisation of the LCS problem, which we introduced in [14] as the *all semi-local LCS problem*. It consists in computing the LCS lengths on substrings of a and b as follows:

- the *all string-substring LCS problem*: a against every substring of b;
- the *all prefix-suffix LCS problem*: every prefix of a against every suffix of b;
- symmetrically, the *all substring-string LCS problem* and the *all suffix-prefix LCS problem*, defined as above but with the roles of a and b exchanged.

A string where all the characters are distinct will be called *a permutation*. The all string-substring LCS problem, when restricted to permutations, can be easily seen to be equivalent to the *all local LIS problem*, i.e. the problem of computing the LIS length in every substring of a.

For a string a, we denote by $\Sigma(a)$ the set of characters appearing in a at least once. For a set of characters S, we denote by a/S the substring of a obtained by deleting all characters not contained in S.

In addition to standard integer indices $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$, we use *odd half-integer* indices $\hat{\mathbb{Z}} = \{\ldots, -\frac{5}{2}, -\frac{3}{2}, -\frac{1}{2}, \frac{1}{2}, \frac{3}{2}, \frac{5}{2}, \ldots\}$. For two numbers i, j, we write $i \trianglelefteq j$ if $j - i \in \{0, 1\}$, and $i \lhd j$ if $j - i = 1$. We denote

$$[i : j] = \{i, i+1, \ldots, j-1, j\} \qquad \langle i : j \rangle = \left\{i + \tfrac{1}{2}, i + \tfrac{3}{2}, \ldots, j - \tfrac{3}{2}, j - \tfrac{1}{2}\right\}.$$

3 Problem Analysis

For completeness, in this section we restate (without motivation and proofs) the necessary definitions and results from [14]. In the following section, we give the

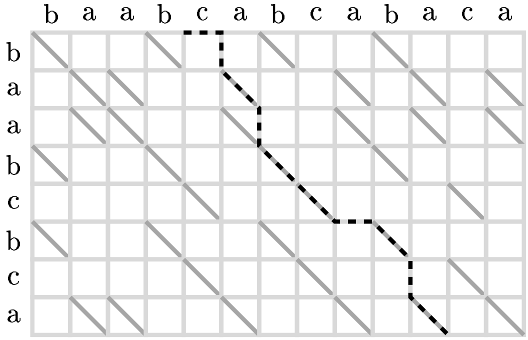


Fig. 1. A grid dag and a maximum-weight path

proof of the key lemma, a significant part of which was omitted from [14] due to space restrictions.

Definition 1. *Let $m, n \in \mathbb{N}$. A* grid dag *G is a weighted dag, defined on the set of nodes $v_{i,j}$, $i \in [0 : m]$, $j \in [0 : n]$. For all $i \in [1 : m]$, $j \in [1 : n]$:*

- *horizontal edge $v_{i,j-1} \to v_{i,j}$ and vertical edge $v_{i-1,j} \to v_{i,j}$ are both always present in G and have weight 0;*
- *diagonal edge $v_{i-1,j-1} \to v_{i,j}$ may or may not be present in G; if present, it has weight 1.*

Given an instance of the all semi-local LCS problem, its *corresponding grid dag* is an $m \times n$ grid dag, where the diagonal edge $v_{i-1,j-1} \to v_{i,j}$ is present, iff $\alpha_i = \beta_j$. Figure 1 shows the grid dag corresponding to strings a = “baabcbca”, b = “baabcabcabaca” (an example borrowed from [2]).

Definition 2. *Given an $m \times n$ grid dag G, its* extension *G^+ is an infinite weighted dag, defined on the set of nodes $v_{i,j}$, $i, j \in \mathbb{Z}$ and containing G as a subgraph. For all $i, j \in \mathbb{Z}$:*

- *horizontal edge $v_{i,j-1} \to v_{i,j}$ and vertical edge $v_{i-1,j} \to v_{i,j}$ are both always present in G^+ and have weight 0;*
- *when $i \in [1 : m]$, $j \in [1 : n]$, diagonal edge $v_{i-1,j-1} \to v_{i,j}$ is present in G^+ iff it is present in G; if present, it has weight 1;*
- *otherwise, diagonal edge $v_{i-1,j-1} \to v_{i,j}$ is always present in G^+ and has weight 1.*

An infinite dag that is an extension of some (finite) grid dag will be called an *extended grid dag.* When dag G^+ is the extension of dag G, we will say that G is the *core* of G^+. Relative to G^+, we will call the nodes of G *core nodes.*

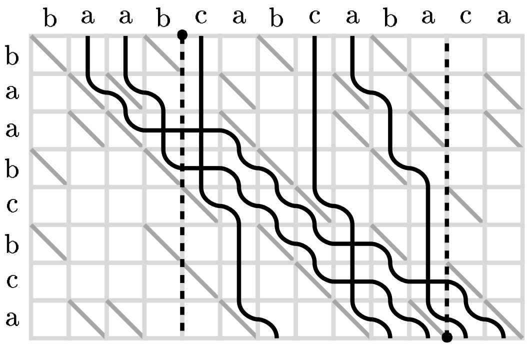


Fig. 2. A grid dag and some critical points

Definition 3. *Given an $m \times n$ grid dag G, its* extended score matrix *is an infinite matrix defined by*

$$A(i,j) = \max weight(v_{0,i} \rightsquigarrow v_{m,j}) \qquad\qquad i,j \in \mathbb{Z} \qquad\qquad (1)$$

where the maximum is taken across all paths between the given endpoints in the extension G^+. If $i = j$, we have $A(i,j) = 0$. By convention, if $j < i$, then we let $A(i,j) = j - i < 0$.

In Figure 1, the highlighted path has weight 5, and corresponds to the value $A(4,11) = 5$, equal to the LCS length of string a and substring $b' =$ "cabcaba".

Definition 4. *An odd half-integer point $(i,j) \in \hat{\mathbb{Z}}^2$ is called A-*critical*, if*

$$A\bigl(i+\tfrac{1}{2}, j-\tfrac{1}{2}\bigr) \lhd A\bigl(i-\tfrac{1}{2}, j-\tfrac{1}{2}\bigr) = A\bigl(i+\tfrac{1}{2}, j+\tfrac{1}{2}\bigr) = A\bigl(i-\tfrac{1}{2}, j+\tfrac{1}{2}\bigr)$$

Corollary 1. *Let $i, j \in \hat{\mathbb{Z}}$. For each i (respectively, j), there exists exactly one j (respectively, i) such that the point (i,j) is A-critical.*

Figure 2 shows the grid dag of Figure 1 along with some of the critical points. More precisely, it shows all the critical points (i,j), where $i, j \in \langle 0 : n \rangle$. Each such critical point is represented by a curve[1] originating between the nodes $v_{0,i-\frac{1}{2}}$ and $v_{0,i+\frac{1}{2}}$, and terminating between the nodes $v_{m,j-\frac{1}{2}}$ and $v_{m,j+\frac{1}{2}}$.

Definition 5. *Point (i_0, j_0)* dominates[2] *point (i,j), if $i_0 < i$ and $j < j_0$.*

Theorem 1. *For an arbitrary integer point $(i_0, j_0) \in \mathbb{Z}^2$, let $d_A(i_0, j_0)$ denote the number of (odd half-integer) A-critical points it dominates. We have*

$$A(i_0, j_0) = j_0 - i_0 - d_A(i_0, j_0)$$

[1] For the purposes of this illustration, the specific layout of the curves between their endpoints is not important.

[2] The standard definition of dominance requires $i < i_0$ instead of $i_0 < i$. Our definition is more convenient in the context of the LCS problem.

In Figure 2, critical points dominated by point $(4, 11)$ are represented by curves whose both endpoints fit between the two vertical lines, corresponding to index values $i = 4$ and $j = 11$. Note that there are exactly two such curves, and that $A(4, 11) = 11 - 4 - 2 = 5$.

Recall that outside the core, the structure of an extended grid graph is trivial: all possible diagonal edges are present in the non-core subgraph. This gives rise to an additional property: when $i < -m$ or $j > m + n$, point (i, j) is A-critical iff $j - i = m$. We will call such A-critical points *trivial.* It is easy to see that an A-critical point (i, j) is non-trivial, iff either both $v_{0,i-\frac{1}{2}}$ and $v_{0,i+\frac{1}{2}}$, or both $v_{m,j-\frac{1}{2}}$ and $v_{m,j+\frac{1}{2}}$, are core nodes.

Corollary 2. *There are exactly $m + n$ non-trivial A-critical points.*

Theorem 2. *For an extended score matrix A, there exists a data structure which*

- *has size $O\big((m + n)\log(m + n)\big)$;*
- *can be built in time $O\big((m + n)\log(m + n)\big)$, given the set of all non-trivial A-critical points;*
- *allows to query an individual element of A in time $O\big(\log(m + n)^2\big)$.*

The above theorem uses the *range tree*, a very simple data structure due to Bentley [4]. Time and memory asymptotics given in the theorem can be improved by using a more advanced data structure due to JaJa et al. [11]; however, Theorem 2 is sufficient for our current purposes.

4 Longest Common Subsequences in Permutations

Similarly to the efficient algorithm for the all semi-local LCS problem described in [14], we follow a divide-and-conquer approach in our new algorithm. String a is recursively partitioned into substrings. Consider a partitioning $a = a_1a_2$ into a concatenation of two substrings of length m_1, m_2, where $m_1 + m_2 = m$. Let A, B, C denote the extended score matrices for the all semi-local LCS problems comparing respectively a_1, a_2, a against b. In every recursive call our goal is, given matrices A, B, to compute matrix C efficiently. We call this procedure *merging.* All three matrices are assumed to be in the geometric representation introduced in Section 3.

By Theorem 1, matrices A, B, C can each be represented by the sets of respectively m_1+n, m_2+n, $m+n$ non-trivial critical points. Our new algorithm is based on a novel merging procedure introduced in [14].

Lemma 1. *Given subproblems with score matrices A, B, C as described above, the sets of A- and B-critical points can be merged into the set of C-critical points in time $O\big(m + n^{1.5}\big)$ and memory $O(m + n)$.*

For a function f and a predicate P defined on a variable i, notation “$\text{any}_{i:P(i)}f(i)$” will denote the value $f(i)$, where index i is chosen arbitrarily from the set

$\{i : P(i)\}$. This is analogous to the use of "$\min_{i:P(i)} f(i)$" to denote the minimum of a function on a given index set[3].

Proof (Lemma 1). Our goal is to compute the set of all non-trivial C-critical points. Without loss of generality, we may assume that $2m_1 = 2m_2 = m$, and that n is a power of 2 (otherwise, appropriate padding can be applied to the input). We will compute non-trivial C-critical points in two stages:

1. points $(i, k) \in \langle -m, -\frac{m}{2} \rangle \times \langle 0, \frac{m}{2} + n \rangle \cup \langle -\frac{m}{2}, n \rangle \times \langle \frac{m}{2} + n, m + n \rangle$;
2. points $(i, k) \in \langle -\frac{m}{2}, n \rangle \times \langle 0, \frac{m}{2} + n \rangle$.

It is easy to see that every non-trivial C-critical point (i, j) is computed in either the first or the second stage. Informally, each C-critical point in the first stage is obtained as a direct combination of an A-critical and a B-critical point, exactly one of which is trivial. All A-critical and B-critical points remaining in the second stage are non-trivial, and determine collectively the remaining C-critical points. However, in the second stage the direct one-to-one relationship between C-critical points and pairs of A- and B-critical points need not hold.

We now give a formal description of both stages of the algorithm.

First stage. Let $i \in \langle -m, -\frac{m}{2} \rangle$, $j = i + \frac{m}{2}$. Recall that (i, j) is a trivial A-critical point. It is straightforward to check that for all k, (i, k) is C-critical, iff (j, k) is B-critical. Therefore, all $m/2$ C-critical points in $\langle -m, -\frac{m}{2} \rangle \times \langle 0, \frac{m}{2} + n \rangle$ can be found in time $O(m + n)$. Analogously, all $m/2$ C-critical points in $\langle -\frac{m}{2}, n \rangle \times \langle \frac{m}{2} + n, m + n \rangle$ can also be found in time $O(m + n)$. The overall memory cost of the first stage is $O(m + n)$.

Second stage. By Definition 3 and Theorem 1, computing all non-trivial C-critical points is equivalent to determining the set of values

$$d_C(i, k) = \min_j \bigl(d_A(i, j) + d_B(j, k) \bigr) \tag{2}$$

where

$$i \in [-m : n] \qquad j \in \bigl[-\tfrac{m}{2} : \tfrac{m}{2} + n \bigr] \qquad k \in [0 : m + n]$$

However, in this stage, we only need to consider

$$i \in \bigl[-\tfrac{m}{2} : n \bigr] \qquad j \in [0 : n] \qquad k \in \bigl[0 : \tfrac{m}{2} + n \bigr]$$

Observe that none of the A- (respectively, B-) critical points considered in the first stage can be dominated by a point (i, j) (respectively, (j, k)) in the current range of i, j, k. Hence, critical points considered in the first stage cannot contribute to the current stage. We can therefore simplify the problem by eliminating all half-integer indices $i \in \langle -m : n \rangle$, $j \in \bigl\langle -\frac{m}{2} : \frac{m}{2} + n \bigr\rangle$, $k \in \langle 0 : m + n \rangle$, that correspond to triples i, j, k considered in the first stage. This results in m half-integer

[3] In fact, "min" (or "max") can always be used instead of "any" on a finite index set; however, such usage could be misleading when "any" happens to be sufficient.

index values being removed from each of the three ranges. We now renumber the remaining indices i, k, so that their new range becomes $i, k \in \langle 0 : n \rangle$. The index order is preserved by the renumbering, so the dominance relation is not affected. We already have $j \in \langle 0 : n \rangle$ after elimination, therefore index j need not be renumbered. The index elimination and renumbering can be done in time $O(m)$.

In the new index numbering, we will refer to the remaining critical points as $\bar{A}$-, $\bar{B}$- and $\bar{C}$-critical. Let $d_{\bar{A}}$, $d_{\bar{B}}$, $d_{\bar{C}}$, denote the values defined analogously to d_A, d_B, d_C by the remaining critical points in the new index ranges[4]. From (2), we now have

$$d_{\bar{C}}(i,k) = \min_j \bigl(d_{\bar{A}}(i,j) + d_{\bar{B}}(j,k)\bigr) \qquad i,j,k \in [0:n] \tag{3}$$

We proceed by partitioning the square index pair range $\langle 0 : n \rangle^2$ recursively into regular half-sized square blocks. For each block, we establish the number of $\bar{C}$-critical points contained in it, and perform further recursive partitioning of the block as long as this number is greater than 0.

Consider an $h \times h$ block

$$\langle i_0 - h : i_0 \rangle \times \langle k_0 : k_0 + h \rangle$$

The $\bar{C}$-critical points in this block will be determined by $\bar{A}$-critical points in $\langle i_0 - h : i_0 \rangle \times \langle 0 : n \rangle$, and $\bar{B}$-critical points in $\langle 0 : n \rangle \times \langle k_0 : k_0 + h \rangle$. We call such $\bar{A}$- and $\bar{B}$-critical points *relevant*. For the current block, there are exactly h relevant $\bar{A}$-critical and exactly h relevant $\bar{B}$-critical points.

For any $j \in [0 : n]$, let $\delta_{\bar{A}}(j)$ (respectively, $\delta_{\bar{B}}(j)$) denote the number of relevant $\bar{A}$-critical (respectively, $\bar{B}$-critical) points in $\langle i_0 - h : i_0 \rangle \times \langle 0 : j \rangle$ (respectively, $\langle j : n \rangle \times \langle k_0 : k_0 + h \rangle$):

$$\delta_{\bar{A}}(j) = d_{\bar{A}}(i_0 - h, j) - d_{\bar{A}}(i_0, j) \qquad \delta_{\bar{B}}(j) = d_{\bar{B}}(j, k_0 + h) - d_{\bar{B}}(j, k_0)$$

Sequence $\delta_{\bar{A}}$ is non-strictly monotonically increasing from $\delta_{\bar{A}}(0) = 0$ to $\delta_{\bar{A}}(n) = h$. Sequence $\delta_{\bar{B}}$ is non-strictly monotonically decreasing from $\delta_{\bar{B}}(0) = h$ to $\delta_{\bar{B}}(n) = 0$.

As the block size h gets smaller, sequences $\delta_{\bar{A}}$, $\delta_{\bar{B}}$ contain fewer and fewer distinct values. We represent these sequences compactly by storing, for every $d \in [-h : h]$, the values

$$\Delta_{\bar{A}}(d) = \text{any}\, \delta_{\bar{A}}(j) \qquad \Delta_{\bar{B}}(d) = \text{any}\, \delta_{\bar{B}}(j)$$
$$M(d) = \min\bigl(d_{\bar{A}}(i_0, j) + d_{\bar{B}}(j, k_0)\bigr)$$

where "any" and "min" are taken across all $j : \delta_{\bar{A}}(j) - \delta_{\bar{B}}(j) = d$. When the set of such j is empty, the corresponding values $\Delta_{\bar{A}}(d)$, $\Delta_{\bar{B}}(d)$, $M(d)$ are undefined

[4] The procedure of the second stage can also be carried out directly on values d_A, d_B, d_C across the original index ranges, without performing the first stage and the subsequent index elimination and renumbering. However, in this case the time cost of merging will increase from $O\bigl(m + n^{1.5}\bigr)$ to $O\bigl((m+n)^{1.5}\bigr)$.

and omitted from further computations. Sequence $\varDelta_{\bar{A}}$ is non-strictly monotonically increasing from $\varDelta_{\bar{A}}(-h) = 0$ to $\varDelta_{\bar{A}}(h) = h$ (ignoring the undefined values). Sequence $\varDelta_{\bar{B}}$ is non-strictly monotonically decreasing from $\varDelta_{\bar{B}}(-h) = h$ to $\varDelta_{\bar{B}}(h) = 0$ (again ignoring the undefined values). Sequences $\varDelta_{\bar{A}}$, $\varDelta_{\bar{B}}$ can be computed in time $O(h)$ by a single scan of the set of relevant $\bar{A}$- and $\bar{B}$-critical points. Sequence M is computed at the top level of recursion in time $O(n)$ by a scan of all $\bar{A}$- and $\bar{B}$-critical points (all of which are relevant at the top recursion level). In lower levels of recursion, sequence M is recomputed in time $O(h)$ by a procedure that will be described below. From sequences $\varDelta_{\bar{A}}$, $\varDelta_{\bar{B}}$, M, the following values can be found in time $O(h)$:

$$\begin{aligned}
&d_{\bar{C}}(i_0, k_0) = \min M(d)\\
&d_{\bar{C}}(i_0 - h, k_0) = \min\bigl(\varDelta_{\bar{A}}(d) + M(d)\bigr)\\
&d_{\bar{C}}(i_0, k_0 + h) = \min\bigl(M(d) + \varDelta_{\bar{B}}(d)\bigr)\\
&d_{\bar{C}}(i_0 - h, k_0 + h) = \min\bigl(\varDelta_{\bar{A}}(d) + M(d) + \varDelta_{\bar{B}}(d)\bigr)
\end{aligned}$$

where "min" is taken across all $d \in [-h : h]$ for which $\varDelta_{\bar{A}}(d)$, $\varDelta_{\bar{B}}(d)$, $M(d)$ are defined. The number of $\bar{C}$-critical points in the current block can then be determined as

$$d_{\bar{C}}(i_0 - h, k_0 + h) - d_{\bar{C}}(i_0 - h, k_0) - d_{\bar{C}}(i_0, k_0 + h) + d_{\bar{C}}(i_0, k_0)$$

If the above value is non-zero, the recursion proceeds by partitioning the current block of size h into four subblocks of size $h/2$. The sets of relevant $\bar{A}$- and $\bar{B}$-critical points are split accordingly, each into two subsets (not necessarily of equal size). Let $i'_0 \in \bigl\{i_0, i_0 - \frac{h}{2}\bigr\}$, $k'_0 \in \bigl\{k_0, k_0 + \frac{h}{2}\bigr\}$, and consider each of the four half-sized subblocks $\bigl\langle i'_0 - \frac{h}{2} : i'_0\bigr\rangle \times \bigl\langle k'_0, k'_0 + \frac{h}{2}\bigr\rangle$. Let $\delta'_{\bar{A}}$, $\delta'_{\bar{B}}$, M' denote the sequences defined for the current subblock analogously to sequences $\delta_{\bar{A}}$, $\delta_{\bar{B}}$, M for the parent block. For every $d \in [-h : h]$, let

$$\varDelta^*_{\bar{A}}(d) = \operatorname{any} \delta'_{\bar{A}}(j) \qquad \varDelta^*_{\bar{B}}(d) = \operatorname{any} \delta'_{\bar{B}}(j)$$

where "any" is taken across all $j : \delta_{\bar{A}}(j) - \delta_{\bar{B}}(j) = d$. When the set of such j is empty, $\varDelta^*_{\bar{A}}(d)$, $\varDelta^*_{\bar{B}}(d)$ are undefined. Sequence $\varDelta^*_{\bar{A}}$ is non-strictly monotonically increasing from $\varDelta^*_{\bar{A}}(-h) = 0$ to $\varDelta^*_{\bar{A}}(h) = h/2$ (ignoring the undefined values). Sequence $\varDelta^*_{\bar{B}}$ is non-strictly monotonically decreasing from $\varDelta^*_{\bar{B}}(-h) = h/2$ to $\varDelta^*_{\bar{B}}(h) = 0$ (again ignoring the undefined values). Similarly to $\varDelta_{\bar{A}}$, $\varDelta_{\bar{B}}$, sequences $\varDelta^*_{\bar{A}}$, $\varDelta^*_{\bar{B}}$ can be computed in time $O(h)$ by a single scan of the set of relevant $\bar{A}$- and $\bar{B}$-critical points. In each of the four subblocks, values $M'(d')$ for all $d' \in \bigl[-\frac{h}{2} : \frac{h}{2}\bigr]$ can now be obtained from sequence M by

$$\begin{aligned}
&M'(d') = \min M(d) && \text{for } i'_0 = i_0,\ k'_0 = k_0\\
&M'(d') = \min\bigl(\varDelta^*_{\bar{A}}(d) + M(d)\bigr) && \text{for } i'_0 = i_0 - \tfrac{h}{2},\ k'_0 = k_0\\
&M'(d') = \min\bigl(M(d) + \varDelta^*_{\bar{B}}(d)\bigr) && \text{for } i'_0 = i_0,\ k'_0 = k_0 + \tfrac{h}{2}\\
&M'(d') = \min\bigl(\varDelta^*_{\bar{A}}(d) + M(d) + \varDelta^*_{\bar{B}}(d)\bigr) && \text{for } i'_0 = i_0 - \tfrac{h}{2},\ k'_0 = k_0 + \tfrac{h}{2}
\end{aligned}$$

where "min" is taken across all $d : \Delta^*_{\bar{A}}(d) - \Delta^*_{\bar{B}}(d) = d'$ for which $\Delta^*_{\bar{A}}(d)$, $\Delta^*_{\bar{B}}(d)$, $M(d)$ are defined. Note that sequence M' is obtained purely from the sequences $\delta'_{\bar{A}}$, $\delta'_{\bar{B}}$ and M; in particular, evaluation of functions $d_{\bar{A}}$, $d_{\bar{B}}$ is not required. For each of the four subblocks, every value $M(d)$ contributes to exactly one value $M'(d')$, therefore the above computation can be done in time $O(h)$.

The base of the recursion is $h = 1$. At this point, we establish all 1×1 blocks containing a $\bar{C}$-critical point, which is equivalent to establishing the $\bar{C}$-critical points themselves. The merging is completed.

The recursion tree has maximum degree 4, height $\log n$, and n leaves corresponding to non-trivial $\bar{C}$-critical points.

Consider the top-to-middle levels of the recursion tree. As we move down from the top to the middle level, in each level the maximum number of nodes increases by a factor of 4, and the maximum amount of computational work per node decreases by a factor of 2. Hence, the maximum amount of work per level increases in geometric progression, and is dominated by the middle level $\frac{\log n}{2}$.

Consider the middle-to-bottom levels of the recursion tree. Since the tree has n leaves, each level contains at most n nodes. As we move down from the middle to the bottom level, in each level the maximum amount of computational work per node still decreases by a factor of 2. Hence, the maximum amount of work per level decreases in geometric progression, and is again dominated by the middle level $\frac{\log n}{2}$.

Thus, the computational work in the whole recursion tree is dominated by the maximum amount of work in the middle level $\frac{\log n}{2}$. This level has at most n nodes, each requiring at most $O(n)/2^{\frac{\log n}{2}} = O\bigl(n^{1/2}\bigr)$ work. Therefore, the overall computation cost of the recursion is at most $n \cdot O\bigl(n^{1/2}\bigr) = O\bigl(n^{1.5}\bigr)$.

The main recursion tree can be evaluated depth-first, so that the overall memory cost is dominated by the top level of the main recursion, running in memory $O(n)$.

In summary, the first stage takes time and memory $O(m + n)$. The second stage takes time and memory $O(m + n)$ for index elimination and renumbering, and then time $O(n^{1.5})$ and memory $O(n)$ for the recursion. Therefore, we have the total time and memory cost as claimed. □

We now describe our new algorithm for the all semi-local LCS problem on permutations. In contrast with the algorithm of [14], which works by partitioning both input strings recursively in alternate order, here we only need to partition the first input string. Instead of partitioning the second string, we reduce it in every recursive step by removing "redundant" characters not appearing in the corresponding part of the first string.

Algorithm 1 (All semi-local LCS in permutations).
Input: permutations a, b of length n over a set of n characters.
Output: all semi-local LCS matrix on a, b, represented by $2n$ non-trivial critical points.
Description. Without loss of generality, we assume that n is a power of 2. The computation proceeds recursively, partitioning string a into a concatenation $a = a_1 a_2$ of two strings of length $n/2$. Each of the strings a_1, a_2 is a permutation over a set of $n/2$ characters.

Let the matrices A, B, C be defined as above. Each of the matrices A, B can be represented by $3n/2$ critical points. Note that for all $i \in \langle 0 : n \rangle$, point (i, i) is A-critical, iff $\beta_{i+\frac{1}{2}} \notin \Sigma(a_1)$. There are exactly $n/2$ such critical points. The remaining n A-critical points can be obtained by solving recursively the all semi-local LCS problem on strings a_1 and $b' = b/\Sigma(a_1)$, both of which are permutations over character set $\Sigma(a_1) = \Sigma(b')$ of size $n/2$. Similarly, $n/2$ B-critical points can be obtained immediately, and the remaining n B-critical points can be obtained by solving recursively the all semi-local LCS problem on strings a_2 and $b'' = b/\Sigma(a_2)$.

Given a current partitioning, the corresponding sets of critical points are merged by Lemma 1. Note that we now have two nested recursions: the main recursion of the algorithm, and the inner recursion of Lemma 1.

The base of the main recursion is $n = 1$.

Cost analysis. Consider the main recursion tree. The computational work in the tree is dominated by the top recursion level. In that level, we sort the input strings a, b in time $O(n \log n)$, after which the substrings b', b'' can easily be computed in linear time. The merging of score matrices by Lemma 1 takes time $O(n^{1.5})$, and therefore dominates the rest of the computation.

The main recursion tree can be evaluated depth-first, so that the overall memory cost is dominated by the top level of the main recursion, running in memory $O(n)$. □

5 Maximum Cliques in Circle Graphs

A *circle graph* [8] is defined as the *intersection graph* of a set of chords in a circle, i.e. the graph where nodes correspond to the chords, and two nodes are adjacent iff the corresponding chords intersect. The *interval model* of a circle graph is obtained by cutting the circle at an arbitrary point and laying it out on a line, so that the chords become intervals. The original circle graph is isomorphic to the *overlap graph* of its interval model, i.e. the graph where nodes correspond to the intervals, and two nodes are adjacent iff the corresponding intervals intersect but do not contain one another.

It has long been known that many problems which are NP-hard for general graphs are solvable in polynomial time on circle graphs. It is also known that the maximum clique and the maximum independent set problems on a circle graph are related to string comparison problems (see e.g. [3]). As an interesting application of our new method for permutation string comparison, we propose a new algorithm for finding a maximum clique in a circle graph.

As the input, the algorithm takes an interval model of a circle graph G on n nodes. Without loss of generality, we may assume that the set of interval endpoints is $[1 : 2n]$. The interval model is represented by a permutation $a = \alpha_1 \ldots \alpha_{2n}$ of size $2n$, where for each left (respectively, right) endpoint $i \in [1 : 2n]$, α_i is the corresponding right (respectively, left) endpoint. Note that for all $i < j$, an interval with left endpoint i does not contain an interval with left endpoint j, iff $\alpha_i < \alpha_j$. Various alternative representations of interval models (e.g. the ones used in [13, 3]) can be converted to this representation in linear time.

In the interval model, a clique corresponds to a set of pairwise intersecting intervals, none of which contains another interval from the set. Recall that intervals in the line satisfy the *Helly property*: if all intervals in a set intersect pairwise, then they all intersect at a common point. In our context, we only need to consider odd half-integer indices $\langle 1 : 2n \rangle$ as intersection points.

Consider a clique in G. Let $k + \frac{1}{2}$, where $k \in [1 : 2n - 1]$, be a common intersection point of the intervals representing the clique, guaranteed to exist by the Helly property. Let $id = (1, 2, \ldots, 2n)$ denote the identity permutation. From the observations above, it follows that the clique corresponds to a common subsequence of a prefix of a of length k and a suffix of id of length $2n - k$. Consequently, the maximum clique can be found by solving the all prefix-suffix LCS problem, which is one of the constituents of the all semi-local LCS problem.

Algorithm 2 (Maximum clique in circle graph).
Input: interval model of circle graph G, represented by string a of size $2n$.
Output: maximum-size clique of G, represented by the set of (say) left endpoints of the corresponding intervals.
Description. We run Algorithm 1 on the input permutation a and the identity permutation id, obtaining the set of $4n$ non-trivial critical points. We then build the data structure of Theorem 2 for querying semi-local LCS lengths of a, id. Let $a^{(k)}$ (respectively, $id_{(k)}$) denote the prefix of a of length k (respectively, the suffix of id of length $2n - k$). For each $k \in [1 : 2n - 1]$, we query the LCS length of $a^{(k)}$ and $id_{(k)}$. The maximum of the $2n - 1$ returned values gives the size of the maximum clique in G, and the corresponding value $k + \frac{1}{2}$ gives a common intersection point of the clique intervals. The intervals in the clique can now be obtained by running a standard LIS algorithm (see e.g. [7, 5]) on string $a^{(k)}/\Sigma(id_{(k)})$.
Cost analysis. The cost of running Algorithm 1 is $O(n^{1.5})$. The combined cost of all the prefix-suffix queries is $O\big(n(\log n)^2\big)$. The cost of running the final LIS algorithm is $O(n \log n)$. The resulting total running time is $O(n^{1.5})$. □

6 Conclusions

We have proposed an efficient algorithm for the all semi-local LCS problem on permutations, running in time $O(n^{1.5})$. As a consequence, our algorithm provides a significant worst-case improvement over existing algorithms for the maximum clique problem in a circle graph. Several output-sensitive algorithms exist for the latter problem. Therefore, our method has an advantage only when the input circle graph is dense, resulting in a large maximum clique. It is possible that our method can be extended to provide an improved output-sensitive algorithm for sparse circle graphs.

Another interesting question is whether our method can be extended to other related problems on circle graphs, in particular the weighted clique problem and the maximum independent set problem, or to other types of graphs, e.g. interval and circular-arc graphs.

Acknowledgement

The author thanks Gad Landau for fruitful discussion.

References

1. M. H. Albert, A. Golynski, A. M. Hamel, A. López-Ortiz, S. S. Rao, and M. A. Safari. Longest increasing subsequences in sliding windows. *Theoretical Computer Science*, 321:405–414, 2004.
2. C. E. R. Alves, E. N. Cáceres, and S. W. Song. An all-substrings common subsequence algorithm. *Electronic Notes in Discrete Mathematics*, 19:133–139, 2005.
3. A. Apostolico, M. J. Atallah, and S. E. Hambrusch. New clique and independent set algorithms for circle graphs. *Discrete Applied Mathematics*, 36:1–24, 1992.
4. J. L. Bentley. Multidimensional divide-and-conquer. *Communications of the ACM*, 23(4):214–229, 1980.
5. S. Bespamyatnikh and M. Segal. Enumerating longest increasing subsequences and patience sorting. *Information Processing Letters*, 76:7–11, 2000.
6. E. Chen, H. Yuan, and L. Yang. Longest increasing subsequences in windows based on canonical antichain partition. In *Proceedings of ISAAC*, volume 3827 of *Lecture Notes in Computer Science*, pages 1153–1162, 2005.
7. E. W. Dijkstra. Some beautiful arguments using mathematical induction. *Acta Informatica*, 13:1–8, 1980.
8. S. Even and A. Itai. Queues, stacks and graphs. In *Theory of Machines and Computations*, pages 71–86. Academic Press, 1971.
9. F. Gavril. Algorithms for a maximum clique and a maximum independent set of a circle graph. *Networks*, 3:261–273, 1973.
10. W.-L. Hsu. Maximum weight clique algorithms for circular-arc graphs and circle graphs. *SIAM Journal on Computing*, 14(1):224–231, 1985.
11. J. JaJa, C. Mortensen, and Q. Shi. Space-efficient and fast algorithms for multidimensional dominance reporting and counting. In *Proceedings of the 15th ISAAC*, volume 3341 of *Lecture Notes in Computer Science*, pages 558–568, 2004.
12. S. Masuda, K. Nakajima, T. Kashiwabara, and T. Fujisawa. Efficient algorithms for finding maximum cliques of an overlap graph. *Networks*, 20:157–171, 1990.
13. D. Rotem and J. Urrutia. Finding maximum cliques in circle graphs. *Networks*, 11:269–278, 1981.
14. A. Tiskin. All semi-local longest common subsequences in subquadratic time. In *Proceedings of CSR*, volume 3967 of *Lecture Notes in Computer Science*, pages 352–363, 2006.

A Simpler Analysis of Burrows-Wheeler Based Compression

Haim Kaplan, Shir Landau, and Elad Verbin

School of Computer Science, Tel Aviv University, Tel Aviv, Israel
{haimk, landaush, eladv}@post.tau.ac.il

Abstract. In this paper we present a new technique for worst-case analysis of compression algorithms which are based on the Burrows-Wheeler Transform. We deal mainly with the algorithm purposed by Burrows and Wheeler in their first paper on the subject [6], called BW0. This algorithm consists of the following three steps: 1) Compute the Burrows-Wheeler transform of the text, 2) Convert the transform into a sequence of integers using the move-to-front algorithm, 3) Encode the integers using Arithmetic code or any order-0 encoding (possibly with run-length encoding).

We prove a strong upper bound on the worst-case compression ratio of this algorithm. This bound is significantly better than bounds known to date and is obtained via simple analytical techniques. Specifically, we show that for any input string s, and $\mu > 1$, the length of the compressed string is bounded by $\mu \cdot |s|H_k(s) + \log(\zeta(\mu)) \cdot |s| + g_k$ where H_k is the k-th order empirical entropy, g_k is a constant depending only on k and on the size of the alphabet, and $\zeta(\mu) = \frac{1}{1^\mu} + \frac{1}{2^\mu} + \ldots$ is the standard zeta function. As part of the analysis we prove a result on the compressibility of integer sequences, which is of independent interest.

Finally, we apply our techniques to prove a worst-case bound on the compression ratio of a compression algorithm based on the Burrows-Wheeler transform followed by distance coding, for which worst-case guarantees have never been given. We prove that the length of the compressed string is bounded by $1.7286 \cdot |s|H_k(s) + g_k$. This bound is *better* than the bound we give for BW0.

1 Introduction

In 1994, Burrows and Wheeler [6] introduced the Burrows-Wheeler Transform (BWT), and two new lossless text-compression algorithms that are based on this transform. Following [15], we refer to these algorithms as BW0 and BW0$_{RL}$. A well known implementation of these algorithms is bzip2 [18]. This program typically shrinks an English text to about 20% of its original size while gzip only shrinks to about 26% of the original size (see Table 1 and also [1] for detailed results). In this paper we refine and tighten the analysis of BW0. For this purpose we introduce new techniques and statistical measures. We believe these techniques may be useful for the analysis of other compression algorithms, and in predicting the performance of these algorithms in practice.

M. Lewenstein and G. Valiente (Eds.): CPM 2006, LNCS 4009, pp. 282–293, 2006.

The algorithm BW0 compresses the input text s in three steps.

1. Compute the Burrows-Wheeler Transform, $\hat{s}$, of s. We elaborate on this stage shortly.[1]
2. Transform $\hat{s}$ to a string of integers $\dot{s} = \text{MTF}(\hat{s})$ by using the move to front algorithm. This algorithm maintains the symbols of the alphabet in a list and encodes the next character by its index in the list (see Section 2).
3. Encode the string $\dot{s}$ of integers by using an order-0 encoder, to obtain the final bit stream $\text{BW0}(s) = \text{ORDER0}(\dot{s})$. An order-0 encoder assigns a unique bit string to each integer independently of its context, such that we can decode the concatenation of these bit strings. Common order-0 encoders are Huffman code or Arithmetic code.

The algorithm BW0_{RL} performs an additional run-length encoding (RLE) procedure between steps 2 and 3. See [6, 15] for more details on BW0 and BW0_{RL}, including the definition of run-length encoding which we omit here.

Next we define the Burrows-Wheeler Transform (BWT). Let n be the length of s. We obtain $\hat{s}$ as follows. Add a unique end-of-string symbol '\$' to s. Place all the cyclic shifts of the string $s\$$ in the rows of an $(n+1) \times (n+1)$ conceptual matrix. One may notice that each row and each column in this matrix is a permutation of $s\$$. Sort the rows of this matrix in lexicographic order ('\$' is considered smaller than all other symbols). The permutation of $s\$$ found in the last column of this sorted matrix, with the symbol '\$' omitted, is the Burrows-Wheeler Transform, $\hat{s}$. See an example in Figure 1. Although it may not be obvious at first glance, BWT is an invertible transformation, given that the location of '\$' prior to its omission is known to the inverting procedure. In fact, efficient methods exist for computing and inverting $\hat{s}$ in linear time (see for example [16]).

The BWT is effective for compression since in $\hat{s}$ characters with the same context[2] appear consecutively. This is beneficial since if a reasonably small context tends to predict a character in the input text s, then the string $\hat{s}$ will show local similarity – that is, symbols will tend to recur at close vicinity.

Therefore, if s is say a text in English, we would expect $\hat{s}$ to be a string with symbols recurring at close vicinity. As a result $\dot{s} = \text{MTF}(\hat{s})$ is an integer string which we expect to contain many small numbers. (Note that by "integer string" we mean a string over an integer alphabet). Furthermore, the frequencies of the integers in $\dot{s}$ are skewed, and so an order-0 encoding of $\dot{s}$ is likely to be short. This, of course, is an intuitive explanation as to why BW0 "should" work on *typical* inputs. As we discuss next, our work is in a worst-case setting, which means that we give upper bounds that hold for *any* input. These upper bounds are relative to statistics which measure how "well-behaved" our input string is. An interesting question which we try to address is which statistics actually capture the compressibility of the input text.

[1] For compatibility with other definitions, we actually need to compute the BWT of s in reversed order, that is from right to left. This does not change our results and does not effect the compression ratio significantly (see [10] for a discussion on this), so we ignore this point from now on.

[2] The context of length k of a character is the string of length k preceding it.

```
mississippi$        $ mississipp i
ississippi$m        i $mississip p
ssissippi$mi        i ppi$missis s
sissippi$mis        i ssippi$mis s
issippi$miss        i ssissippi$ m
ssippi$missi   ⟹   m ississippi $
sippi$missis        p i$mississi p
ippi$mississ        p pi$mississ i
ppi$mississi        s ippi$missi s
pi$mississip        s issippi$mi s
i$mississipp        s sippi$miss i
$mississippi        s sissippi$m i
```

Fig. 1. The Burrows-Wheeler transform for the string $s = mississippi$. The matrix on the right has the rows sorted in lexicographic order. $\hat{s}$ is the last column of the matrix, i.e. *ipssmpissii*, and we need to store the index of the symbol '\$', i.e. 6, to be able to get the original string.

Introductory Definitions. minus .1em Let s be the string which we compress, and let Σ denote the alphabet (set of symbols in S). Let $n = |s|$, and $h = |\Sigma|$. Let n_σ be the number of occurrences of the symbol σ in s. Let Σ^k denote the set of strings of length k over Σ. Let $\Sigma^* = \bigcup_{k \geq 0} \Sigma^k$ denote the set of all (finite) strings over Σ. For a compression algorithm A we denote by A(s) the output of A on a string s. The zeroth order empirical entropy of the string s is defined as $H_0(s) = \sum_{i=0}^{h-1} \frac{n_i}{n} \log \frac{n}{n_i}$. (All logarithms in the paper are to the base 2. In this context, we define $0 \log(n/0) = 0$, for any n). For any word $w \in \Sigma^k$, let w_s denote the string consisting of the characters following all occurrences of w in s. The value $H_k(s) = \frac{1}{n} \sum_{w \in \Sigma^k} |w_s| \, H_0(w_s)$ is called the k-th order empirical entropy of the string s. In [15] these terms, as well as BWT, are discussed in greater depth.

We also use the *zeta function*, $\zeta(\mu) = \frac{1}{1^\mu} + \frac{1}{2^\mu} + \ldots$, and the *truncated zeta function* $\zeta_h(\mu) = \frac{1}{1^\mu} + \ldots + \frac{1}{h^\mu}$. We denote by $[h]$ the integers $\{0, \ldots, h-1\}$.

History and Motivation. Define the *compression ratio* of a compression algorithm to be the average number of bits it produces per character in s. It is well known that the zeroth order empirical entropy of a string s, $H_0(s)$, is a lower bound on the compression ratio of any order-0 compressor [7, 12]. Similarly, the k-th order empirical entropy of a string s, denoted by $H_k(s)$, gives a lower bound on the compression ratio of any encoder, that to encode a character x, is allowed to use only the k characters preceding x (the context of length k of x). For this reason the compression ratio of compression algorithms is traditionally compared to $H_k(s)$, for various values of k. Another widely used statistic is $H_k^*(s)$, called the *modified* k-th order empirical entropy of s. This statistic is slightly larger than H_k, yet it still provides a lower bound on the bits-per-character ratio of any encoder that is based on a context of k characters. We do not define H_k^* here, as we present bounds only in terms of H_k. See [15] for more details on H_k^*.

In 1999, Manzini [15] gave the first worst-case upper bounds on the compression ratio of several BWT-based algorithms. In particular, Manzini bounded the total bit-length of the compressed text BW0(s) by the expression

$$8 \cdot nH_k(s) + (0.08 + C_{\text{ORDER0}}) \cdot n + \log n + g'_k \ . \tag{1}$$

for any $k \geq 0$. Here C_{ORDER0} is a small constant, defined in Section 2, which depends on the parameters of the Order-0 compressor which we are using, and $g'_k = h^k(2h \log h + 9)$ is a constant that depends only on k and h. Manzini also proved an upper bound of $5 \cdot nH^*_k(s) + g''_k$ on the bit-length of $\text{BW0}_{RL}(s)$, where g''_k is another constant that depends only on k and h.

In 2004, Ferragina, Giancarlo, Manzini and Sciortino [10] introduced a BWT-based compression booster. They show a compression algorithm such that the bit-length of its output is bounded by

$$1 \cdot nH_k(s) + C_{\text{ORDER0}} n + \log n + g'''_k \ . \tag{2}$$

(This algorithm follows from a general compression boosting technique. For details see [10]). As mentioned above this result is optimal, up to the $C_{\text{ORDER0}} n + \log n + g'''_k$ term. The upper bounds of this algorithm and its variants based on the same techniques are theoretically strictly superior to those in [15] and to those that we present here. However, implementations of the algorithm of [10] by the authors and another implementation by Manzini [14], give the results summarized in Table 1. These empirical results surprisingly imply that while the algorithm of [10] is optimal with respect to nH_k in a worst-case setting, its compression ratio in practice is comparable with that of algorithms with weaker worst-case guarantees. This seems to indicate that achieving good bounds with respect to H_k does not necessarily guarantee good compression results in practice. This was the starting point of our research. We looked for tight bounds on the length of the compressed text, possibly in terms of statistics of the text that might be more appropriate than H_k.

We define a new statistic of a text s, which we call the *local entropy* of s, and denote it by LE(s). This statistic was implicitly considered by Bentley et al. [4], and by Manzini [15]. We also define $\widehat{\text{LE}}(s) = \text{LE}(\hat{s})$. That is the statistic $\widehat{\text{LE}}(s)$ is obtained by first applying the Burrows-Wheeler transform to s and then computing the statistic LE of the result. These statistics are theoretically oriented and we find their importance to be two-fold. First they may highlight potential weaknesses of existing compression algorithms and thereby mark the way to invent better compression algorithms. Second, they may be useful in understanding current algorithms and proving better worse-case upper bounds for them.

Our Results. In this paper we tighten the analysis of BW0 and give a tradeoff result that shows that for any constant $\mu > 1$ and for any k, the length of the compressed text is upper-bounded by the expression

$$\mu \cdot nH_k(s) + (\log \zeta(\mu) + C_{\text{ORDER0}}) \cdot n + \log n + \mu g_k \ . \tag{3}$$

Here $g_k = 2k \log h + h^k \cdot h \log h$. In particular, for $\mu = 1.5$ we obtain the bound $1.5 \cdot nH_k(s) + (1.5 + C_{\text{ORDER0}}) \cdot n + \log n + 1.5g_k$. For $\mu = 4.45$ we get the bound

Table 1. Results (in bytes) of running various compressors on the non-binary files from the Canterbury Corpus [1]. The gzip results are taken from [1]. The column BW0 shows the results of our implementation (in C++) of the BW0 algorithm, using Huffman encoding as the order-0 compressor. The column marked [14] gives results from a preliminary implementation of the booster-based compression algorithm supplied to us by Manzini. The columns Booster(HC) and Booster(RHC) are our implementations of the compression booster of [10]. Ferragina et al. [10] suggest two methods to implement it: One using the algorithm HC, and one using the algorithm RHC (the interested reader is referred to [10]).

File Name	size	gzip	bzip2	BW0	[14]	Booster(HC)	Booster(RHC)
alice29.txt	152089	54181	43196	48915	47856	74576	79946
asyoulik.txt	125179	48819	39569	44961	42267	59924	61757
cp.html	24603	7965	7632	8726	8586	16342	16342
fields.c	11150	3122	3039	3435	3658	10235	10028
grammar.lsp	3721	1232	1283	1409	1369	2297	2297
lcet10.txt	426754	144562	107648	127745	116861	166043	177682
plrabn12.txt	481861	194551	145545	168311	154950	172471	183855
xargs.1	4227	1748	1762	1841	1864	2726	2726

$4.45 \cdot nH_k(s) + (0.08 + C_{\text{ORDER0}}) \cdot n + \log n + 4.45g_k$, thus surpassing Manzini's upper bound (1). Our proof is considerably simpler than Manzini's proof of (1).

We prove this bound using two basic observations on the statistic LE that we define. Thereby we bypass some of the technical hurdles in the analysis of [15]. Our analysis actually proves a considerably stronger result. We show that the size of the compressed text is bounded by

$$\mu \cdot \text{LE}(\hat{s}) + (\log \zeta(\mu) + C_{\text{ORDER0}}) \cdot n + \log n \ . \tag{4}$$

Empirically, this seems to give estimates which are quite close to the actual compression, as seen in Table 2.

In order to get our upper bounds we prove in Section 3 a result on compression of integer sequences, which may be of independent interest.

Here is an overview of the rest of the paper.

1. We prove a result on compressibility of integer sequences in Section 3.
2. We define the statistic $\widehat{\text{LE}}$ in Section 2 and show its relation to H_k in Section 4.
3. We use the last two contributions to give a simple proof of the bound (3) in Section 4.
4. We give a tighter upper bound for BW0 for the case that we are working over an alphabet of size 2 in Section 4.1.
5. We outline a further application of our techniques to prove a worst-case bound on the compression of a different BWT-based compressor, which runs BWT, then the so-called distance-coder (see [5, 2]), and finally an order-0 encoder. The upper bounds proved are strictly superior to those proved for BW0. This can be found in Section 5.

Table 2. Results of computing various statistics on the non-binary files from the Canterbury Corpus [1]. Numbers are in "bits" since the theoretical bounds in the literature are customarily calculated in bits. The column denoted by $H_0(\dot{s})$ gives the result of the algorithm BW0 assuming an optimal order-0 compressor. The final three columns show the bounds given by the Equations (4), (3), and (1). The difference between the column of $H_0(\dot{s})$ and the column marked (4) shows that our bound (4) is quite tight in practice. It should be noted that in order to get the bound of (4) we needed to minimize the expression in Equation (4) over μ. To get the bound of (3) and (1) we needed to calculate the values of the corresponding equations for all k and pick the best one.

File Name	size	$H_0(\dot{s})$	$LE(\hat{s})$	(4)	(3)	(1)
alice29.txt	1216712	386367	144247	396813	766940	2328219
asyoulik.txt	1001432	357203	140928	367874	683171	2141646
cp.html	196824	67010	26358	69857.6	105033.2	295714
fields.c	89200	24763	8855	25713	43379	119210
grammar.lsp	29768	9767	3807	10234	16054	45134
lcet10.txt	3414032	805841	357527	1021440	1967240	5867291
plrabn12.txt	3854888	1337475	528855	1391310	2464440	8198976
xargs.1	33816	13417	5571	13858	22317	64673

Due to space limitation, many proofs are omitted. They can found in the full version of the paper.

2 Preliminaries

Our analysis does not use the definitions of H_k and BWT directly. Instead, it uses the following observation of Manzini [15], that $H_k(s)$ is equal to a linear combination of H_0 of parts of $\hat{s}$, the Burrows-Wheeler transform of s.

Proposition 1 ([15]). *Let $\tilde{s}$ be the string obtained from $\hat{s}$ by deleting the occurrences in $\hat{s}$ of the first k characters of s. (Note that these characters can appear in arbitrary positions of $\hat{s}$). There is a partition $\tilde{s} = \tilde{s}_1 \ldots \tilde{s}_t$, with $t \leq h^k$, such that:*

$$|s|\, H_k(s) = \sum_{i=1}^{t} |\tilde{s}_i|\, H_0(\tilde{s}_i) \ . \tag{5}$$

Now we define the move to front (MTF) transformation, which was introduced in [4]. MTF encodes the character $s[i] = \sigma$ with an integer equal to the number of distinct symbols encountered since the previous occurrence of σ in s. More precisely, the encoding maintains a list of the symbols ordered by recency of occurrence. When the next symbol arrives, the encoder outputs its current rank and moves it to the front of the list. Therefore, a string over the alphabet Σ is transformed to a string over $[h]$ (note that the length of the string does not change).[3]

[3] In order to completely determine the encoding we must specify the status of the recency list at the beginning of the procedure. Here and in the future we usually ignore this fact.

$\textsc{mtf}$ has the property that if the input string has high local similarity, that is if symbols tend to recur at close vicinity, then the output string will consist mainly of small integers. We define the *local entropy* of a string s to be $\textsc{le}(s) = \sum_{i=1}^{n} \log(\textsc{mtf}(s)[i] + 1)$. That is, $\textsc{le}$ is the sum of the logarithms of the move-to-front values plus 1. For example, for a string "aabb" and initial list where 'b' is first and 'a' is second, $\textsc{le}(s) = 2$ because the $\textsc{mtf}$ values of the second a and the second b are 0, and the $\textsc{mtf}$ values of the first a and the first b are 1. We define $\widehat{\textsc{le}}(s) = \textsc{le}(\hat{s})$.

Note that $\textsc{le}(s)$ is the number of bits one needs to write the sequence of integers $\textsc{mtf}(s)$ in binary. Optimistically, this is the size we would like to compress the text to. Of course, one cannot decode the integers in $\textsc{mtf}(s)$ from the concatenation of their binary representations as these representations are of variable lengths.

The statistics $H_0(s)$ and $H_k(s)$ are normalized in the sense that they represent lower bounds on the *bits-per-character* rate attainable for compressing s, which we call the *compression ratio*. However, for our purposes it is more convenient to work with un-normalized statistics. Thus we define our new statistic $\textsc{le}$ to be un-normalized. We define the statistics nH_0 and nH_k to be the un-normalized counterparts of the original statistics, i.e. $(nH_0)(s) = n \cdot H_0(s)$ and $(nH_k)(s) = n \cdot H_k(s)$.

Let $f : \Sigma^* \rightarrow \mathbb{R}^+$ be an (un-normalized) statistic on strings, for example f can be nH_k or $\textsc{le}$.

Definition 2. *A compression algorithm A is called (μ, C)-f-competitive if for every string s it holds that $|A(s)| \leq \mu f(s) + Cn + o(n)$, where $o(n)$ denotes a function $g(n)$ such that* $\lim_{n\rightarrow\infty} \frac{g(n)}{n} = 0$.

Throughout the paper we refer to an algorithm called "$\textsc{order}0$". By this we mean any order-0 algorithm, which is assumed to be a $(1, C_{\textsc{order}0})$-nH_0-competitive algorithm. For example, $C_{\textsc{Huffman}} = 1$ and $C_{\textsc{Arithmetic}} \approx 10^{-2}$ for a specific time-efficient implementation of Arithmetic code [17, 19]. Furthermore, one can implement arithmetic code so that it is $(1, 0)$-nH_0-competitive, that is the bit-length of the compressed text is bounded by $nH_0(s) + O(\log n)$. Thus we can use $C_{\textsc{order}0} = 0$ in our equations. This implementation of arithmetic coding is interesting theoretically, but is not time-efficient in practice.

We will often use the following inequality, derived from Jensen's inequality:

Lemma 3. *For any $k \geq 1$, $x_1, \ldots, x_k > 0$ and $y_1, \ldots, y_k > 0$ it holds that* $\sum_{i=1}^{k} y_i \log x_i \leq \left(\sum_{i=1}^{k} y_i\right) \cdot \log\left(\frac{\sum_{i=1}^{k} x_i y_i}{\sum_{i=1}^{k} y_i}\right)$.

In particular this inequality implies that if one wishes to maximize the sum of logarithms of k elements under the constraint that the sum of these elements is S, then one needs to pick all of the elements to be equal to S/k.

3 Optimal Results on Compression with Respect to SL

In this section we look at a string s of length n over the alphabet $[h]$. We define the *sum of logarithms statistic*: $\text{SL}(s) = \sum_{i=1}^{n} \log(s[i]+1)$. Note that $\text{LE}(s) = \text{SL}(\text{MTF}(s))$. We show that in a strong sense the best SL-competitive compression algorithm is an order-0 compressor. In the end of this section we show how to get good LE-competitive and $\widehat{\text{LE}}$-competitive compression algorithms.

The problem we deal with in this section is related to the problem of universal encoding of integers. In the problem of universal encoding of integers [9, 4] the goal is to find a prefix-free encoding for integers, $U : \mathbb{Z}^+ \to \{0,1\}^*$, such that for every $x \geq 0$, $|U(x)| \leq \mu \log(x+1) + C$. A particularly nice solution for this is the Fibonacci encoding [3, 11], for which $\mu = \log_\phi 2 \simeq 1.4404$ and $C = 1 + \log_\phi \sqrt{5} \simeq 2.6723$ (here ϕ is the golden ratio, $\phi = (1+\sqrt{5})/2$). An additional solution for this problem was proposed by Elias [9]. This is an optimal solution, in the sense described in [13]. For more information on universal encoding of integers see the (somewhat outdated) survey paper [13].

Clearly a universal encoding scheme with parameters μ and C gives an (μ, C)-SL-competitive compressor. However, in this section we get a better competitive ratio, taking advantage of the fact that our goal is to encode a long sequence over $[h]$, while allowing an $o(n)$ additive term.

An Optimal (μ, C)-SL-Competitive Algorithm. We show, using a technique based on Lemma 3, that the algorithm ORDER0 is $(\mu, \log\zeta(\mu) + C_{\text{ORDER0}})$-SL-competitive *for any* $\mu > 1$. In fact, we prove a somewhat stronger theorem:

Theorem 4. *For any constant $\mu > 0$, the algorithm* ORDER0 *is $(\mu, \log\zeta_h(\mu) + C_{\text{ORDER0}})$-SL-competitive.*

Proof. Let s be a string of length n over alphabet $[h]$. Clearly it suffices to prove that for any constant $\mu > 0$, $nH_0(s) \leq \mu\text{SL}(s) + n\log\zeta_h(\mu)$.

From the definition of H_0 it follows that $nH_0(s) = \sum_{i=0}^{h-1} n_i \log\frac{n}{n_i}$, and from the definition of SL we get that $\text{SL}(s) = \sum_{j=1}^{n}\log(s[j]+1) = \sum_{i=0}^{h-1} n_i \log(i+1)$. Therefore, it suffices to prove that $\sum_{i=0}^{h-1} n_i \log\frac{n}{n_i} \leq \mu\sum_{i=0}^{h-1} n_i\log(i+1) + n\log\zeta_h(\mu)$.

Pushing the μ into the logarithm and moving terms around we get that it suffices to prove that $\sum_{i=0}^{h-1} n_i \log\frac{n}{n_i(i+1)^\mu} \leq n\log\zeta_h(\mu)$.

Defining $p_i = \frac{n_i}{n}$ and dividing the two sides of the inequality by n we get that it suffices to prove that $\sum_{i=0}^{h-1} p_i \log\frac{1}{p_i(i+1)^\mu} \leq \log\zeta_h(\mu)$.

Using Lemma 3 we obtain that $\sum_{i=0}^{h-1} p_i \log\frac{1}{p_i(i+1)^\mu} = \sum_{\substack{0\leq i\leq h-1\\ p_i\neq 0}} p_i \log\frac{1}{p_i(i+1)^\mu}$

$$\leq \log\left(\sum_{\substack{0\leq i\leq h-1\\ p_i\neq 0}} p_i \frac{1}{p_i(i+1)^\mu}\right) = \log\left(\sum_{\substack{0\leq i\leq h-1\\ p_i\neq 0}} \frac{1}{(i+1)^\mu}\right) \leq \log\zeta_h(\mu). \qquad \square$$

In particular we get the following corollary.

Corollary 5. *For any constant* $\mu > 1$*, the algorithm* ORDER0 *is* $(\mu, \log\zeta(\mu) + C_{\text{ORDER0}})$*-*SL*-competitive.*

One also gets the following analogous result with respect to $\widehat{\text{LE}}$:

Corollary 6. *For any constant* $\mu > 0$*, the algorithm* BW0 *is* $(\mu, \log\zeta_h(\mu) + C_{\text{ORDER0}})$*-*$\widehat{\text{LE}}$*-competitive*

A Lower Bound for SL-Competitive Compression. Theorem 4 shows that for any $\mu > 0$ there exists an $(\mu, \log\zeta_h(\mu) + C_{\text{ORDER0}})$-SL-competitive algorithm. We now show that for any fixed values of μ and h there is no algorithm with a better competitive ratio. Note that the lower bound that we get does not include the constant C_{ORDER0}.

Theorem 7. *Let* $\mu > 0$ *be some constant. For any* $C < \log\zeta_h(\mu)$ *there is no* (μ, C)*-*SL*-competitive algorithm.*

4 The Entropy Hierarchy

In this section we show that the statistics nH_k and $\widehat{\text{LE}}$ form a hierarchy, which allows us to percolate upper bounds down and lower bounds up. Specifically, we show that for each k,

$$\widehat{\text{LE}}(s) \le nH_k(s) + O(1) \tag{6}$$

where the $O(1)$ term depends on k and h (recall that h is the size of the alphabet). The known entropy hierarchy is

$$\ldots \le nH_k(s) \le \ldots \le nH_2(s) \le nH_1(s) \le nH_0(s)\ . \tag{7}$$

Which in addition to (6) gives us:

$$\widehat{\text{LE}}(s) \ldots \lessapprox \ldots \le nH_k(s) \le \ldots \le nH_2(s) \le nH_1(s) \le nH_0(s)\ . \tag{8}$$

($O(1)$ additive terms are hidden in the last formula).

Thus any (μ, C)-$\widehat{\text{LE}}$-competitive algorithm is also (μ, C)-nH_k-competitive. To establish this hierarchy we need to prove two properties of LE: that it is at most $nH_0 + o(n)$, and that it is convex (in a sense which we will define).

Some of the following claims appear, explicitly or implicitly, in [4, 15]. We specify them in a form that would help to understand the rest of the analysis.

Manzini [15] gave the following corollary of a theorem of Bentley et al. [4].

Lemma 8 ([15], Lemma 5.4). LE$(s) \le nH_0(s) + h \log h$.

In addition, we need the following lemma about LE.

Lemma 9. *Let* s *be a string of length* n *and let* s' *be a string obtained by deleting exactly one character from* s*. Then* LE$(s) \le$ LE$(s') + 2\log h$.

We now prove that LE is a convex statistic. That is, one cannot gain much by stopping MTF in the middle and restarting it with a different recency list.

Lemma 10 (LE is a convex statistic, implicitly stated in [15]). *Let* $s = s_1 \dots s_t$. *Then* $\textsc{le}(s) \leq \sum_i \textsc{le}(s_i)$

(For the last lemma, and for the lemmas before it, we define the initial state of the recency list of LE as the worst possible state. If we define it differently, we might incur an additive term of $th \log h$ in the last lemma. This would not significantly change our bounds).

From these Lemmas one can derive the hierarchy result:

Theorem 11. *For any* $k \geq 0$ *and any string* s, $|s|\, H_k(s) \geq \widehat{\textsc{le}}(s) - 2k \log h - h^k \cdot h \log h$.

Using Corollary 6 together with Theorem 11 allows us to derive the main result of this paper:

Theorem 12. *For any* $k \geq 0$ *and for any constant* $\mu > 0$, *the algorithm* BW0 *is* $(\mu, \log \zeta_h(\mu) + C_{\textsc{order0}})$-$nH_k$*-competitive (on strings from an alphabet of size* h*).*

4.1 An Upper Bound and a Conjecture About BW0

In the case where the alphabet size is 2 we were able to prove an improved upper bound for BW0:

Theorem 13. BW0 *is* $(2, C_{\textsc{order0}})$-nH_0*-competitive for texts over an alphabet of size 2.*

We believe that this upper bound is true in the general setting. Specifically, we leave the following conjecture as an open problem.

Conjecture 1. BW0 *is* $(2, C_{\textsc{order0}})$-nH_k*-competitive.*

5 A $(1.7286, C_{\textbf{order0}})$-$nH_k$-Competitive Algorithm

In this section we analyze the BWT *with distance coding* compression algorithm, $\textsc{bw}_{\textsc{dc}}$. This algorithm was invented but not published by Binder (see [5, 2]), and is described in a paper of Deorowicz [8]. The distance coding procedure, DC, will be described shortly. The algorithm $\textsc{bw}_{\textsc{dc}}$ compresses the text by running the Burrows-Wheeler Transform, then the distance-coding procedure, and then an Order-0 compressor. It also adds to the compressed string auxiliary information consisting of the positions of the first and last occurrence of each character. In this section we prove that $\textsc{bw}_{\textsc{dc}}$ is $(1.7286, C_{\textsc{order0}})$-$nH_k$-competitive.

First we define a transformation called DIST: DIST encodes the character $s[i] = \sigma$ with an integer equal to the number of characters encountered since the previous occurrence of the symbol σ. Therefore, DIST is the same as MTF, but instead of counting the number of distinct symbols between two consecutive occurrences of σ, it counts the number of characters. In DIST we disregard the first occurrence of each symbol.

The transformation DC converts a text (which would be in our case the Burrows-Wheeler transform of the original text) to a sequence of integers by applying DIST to s and disregarding all zeroes.[4] It follows that DC produces one integer per block of consecutive occurrences of the same character σ. This integer is the distance to the previous block of consecutive occurrences of σ. It is not hard to see that from DC(s) and the auxiliary information we can recover s.

As a tool for our analysis, let us define a new statistic of texts, LD. The LD statistic is similar to LE, except that it counts all characters between two successive occurrences of a symbol, instead of disregarding repeating symbols. Specifically, $\text{LD}(s) = \sum_i \log(\text{DIST}(s)[i] + 1)$. For example, the LD value of the string "abbbab" is $\log 4 + \log 2 = 3$. From the definition of LD and DC, it is easy to see that $\text{SL}(\text{DC}(s)) = \text{LD}(s)$.

We can now state the following theorem.

Theorem 14. *For any $k \geq 0$ and for any constant $\mu > 1$, the algorithm* BW_{DC} *is $(\mu, \log(\zeta(\mu) - 1) + C_{\text{ORDER0}})$-$nH_k$-competitive.*

To prove Theorem 14 we follow the footsteps of the proof of Theorem 12 outlined in Sections 3 and 4, where we use DC instead of MTF, and LD instead of LE. The term $\log(\zeta(\mu) - 1)$ appears instead of $\log \zeta(\mu)$ because the summations in the proof of Theorem 4 now start at $i = 1$ instead of at $i = 0$ (this is because we omitted all zeroes, so all characters of the alphabet are in this case at least 1).

Let $\mu_0 \approx 1.7286$ be the real number such that $\zeta(\mu_0) = 2$. Substituting $\mu = \mu_0$ gives:

Corollary 15. *For any $k \geq 0$, the algorithm* BW_{DC} *is $(\mu_0, C_{\text{ORDER0}})$-$nH_k$-competitive.*

In the full version of this paper we also show that this approach cannot yield better results. Namely, we prove that for any $\mu < \mu_0$, there is no $(\mu, 0)$-LD-competitive algorithm, so we have a matching lower bound for Corollary 15. We show that this lower bound holds even if the alphabet size is 2. We leave as an open problem to determine whether there is a matching lower bound for Theorem 14, or it can be improved.

6 Conclusions and Further Research

We leave the following idea for further research: In this paper we prove that the algorithm BW0 is $(\mu, \log \zeta(\mu))$-$\widehat{\text{LE}}$-competitive. On the other hand, Ferragina et al. [10] show an algorithm which is $(1, 0)$-nH_k-competitive. A natural question to ask is whether there is an algorithm that achieves both ratios. Of course, one can just perform both algorithms and use the shorter result. But the question is whether a direct simple algorithm with such performance exists. We are also

[4] This is a simplified version of [8]. Our upper bound applies to the original version as well, since the original algorithm just adds a few more optimizations that may produce an even shorter compressed string.

curious as to whether the insights gained in this work can be used to produce a better BWT-based compression algorithm.

Acknowledgments. We would like to thank Nir Markus for his work on the implementations. We also thank Gadi Landau for helpful discussions and useful references. We thank Giovanni Manzini for sharing with us a preliminary implementation of the compression booster. The third author would like to thank Roberto Grossi and Adi Avidor for some insightful discussions.

References

[1] The canterbury corpus. http://corpus.canterbury.ac.nz.

[2] J. Abel. Web page about Distance Coding. http://www.data-compression.info/Algorithms/DC/.

[3] A. Apostolico and A. S. Fraenkel. Robust transmission of unbounded strings using fibonacci representations. *IEEE Transactions on Information Theory*, 33(2): 238–245, 1987.

[4] J. L. Bentley, D. D. Sleator, R. E. Tarjan, and V. K. Wei. A locally adaptive data compression scheme. *Communications of the ACM*, 29(4):320–330, 1986.

[5] E. Binder. Distance coder. Usenet group comp.compression, 2000.

[6] M. Burrows and D. J. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, Palo Alto, California, 1994.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*, chapter 16.3, pages 385–392. MIT Press and McGraw-Hill, 2001.

[8] S. Deorowicz. Second step algorithms in the Burrows-Wheeler compression algorithm. *Software - Practice and Experience*, 32(2):99–111, 2002.

[9] P. Elias. Universal codeword sets and representation of the integers. *IEEE Trans. on Information Theory*, 21(2):194–203, 1975.

[10] P. Ferragina, R. Giancarlo, G. Manzini, and M. Sciortino. Boosting textual compression in optimal linear time. *Journal of the ACM*, 52:688–713, 2005.

[11] A. S. Fraenkel and S. T. Klein. Robust universal complete codes for transmission and compression. *Discrete Applied Mathematics*, 64(1):31–55, 1996.

[12] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[13] D. A. Lelewer and D. S. Hirschberg. Data compression. *ACM Computing Surveys*, 19(3):261–296, 1987.

[14] G. Manzini. Personal communication.

[15] G. Manzini. An analysis of the Burrows-Wheeler transform. *Journal of the ACM*, 48(3):407–430, 2001.

[16] G. Manzini and P. Ferragina. Engineering a lightweight suffix array construction algorithm. *Algorithmica*, 40:33–50, 2004.

[17] A. Moffat, R. M. Neal, and I. H. Witten. Arithmetic coding revisited. *ACM Trans. Inf. Syst.*, 16(3):256–294, 1998.

[18] J. Seward. bzip2, a program and library for data compression. http://www.bzip.org/.

[19] I. H. Witten, R. M. Neal, and J. G. Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540, 1987.

Statistical Encoding of Succinct Data Structures

Rodrigo González[⋆] and Gonzalo Navarro[⋆⋆]

Department of Computer Science, University of Chile
{rgonzale, gnavarro}@dcc.uchile.cl

Abstract. In recent work, Sadakane and Grossi [SODA 2006] introduced a scheme to represent any sequence $S = s_1 s_2 \ldots s_n$, over an alphabet of size σ, using $nH_k(S) + O(\frac{n}{\log_\sigma n}(k \log \sigma + \log \log n))$ bits of space, where $H_k(S)$ is the k-th order empirical entropy of S. The representation permits extracting any substring of size $\Theta(\log_\sigma n)$ in constant time, and thus it completely replaces S under the RAM model. This is extremely important because it permits converting any succinct data structure requiring $o(|S|) = o(n \log \sigma)$ bits in addition to S, into another requiring $nH_k(S) + o(n \log \sigma)$ (overall) for any $k = o(\log_\sigma n)$. They achieve this result by using Ziv-Lempel compression, and conjecture that the result can in particular be useful to implement compressed full-text indexes.

In this paper we extend their result, by obtaining the same space and time complexities using a simpler scheme based on statistical encoding. We show that the scheme supports appending symbols in constant amortized time. In addition, we prove some results on the applicability of the scheme for full-text self-indexing.

1 Introduction

Recent years have witnessed an increasing interest on succinct data structures, motivated mainly by the growth over time on the size of textual information. This has triggered a search for less space-demanding data structures bounded by the entropy of the original text. Their aim is to represent the data using as little space as possible, yet efficiently answering queries on the represented data. Several results exist on the representation of sequences [11, 16], trees [13, 3, 4], graphs [13], permutations and functions [12, 14], texts [5, 7, 6, 9], etc.

Several of those *succinct* data structures are built over a sequence of symbols $S[1, n] = s_1 s_2 \ldots s_n$, from an alphabet A of size σ, and require only $o(|S|) = o(n \log \sigma)$ additional bits in addition to S itself (S requires $n \log \sigma$ bits[1]). A more ambitious goal is a *compressed* data structure, which takes overall space proportional to the compressed size of S and still is able to recover any substring of S and manipulate the data structure.

A very recent result by Sadakane and Grossi [18] gives a tool to convert *any* succinct data structure on sequences into a compressed data structure. More

[⋆] Work supported by Mecesup Grant UCH 0109, Chile.

[⋆⋆] Work supported by a grant from Yahoo! Research Latin America.

[1] In this paper log stands for $\log_2$.

M. Lewenstein and G. Valiente (Eds.): CPM 2006, LNCS 4009, pp. 294–305, 2006.

precisely, they show that S can be encoded using $nH_k(S) + O(\frac{n}{\log_\sigma n}(k \log \sigma + \log \log n))$ bits of space[2], where $H_k(S)$ is the k-th order empirical entropy of S [10]. ($H_k(S)$ is a lower bound to the space achieved by any statistical compressor based on k-th order modeling.) Their structure permits retrieving any substring of S of $\Theta(\log_\sigma n)$ symbols in constant time. Under the RAM model of computation this is equivalent to having S in explicit form.

In particular, for sufficiently small $k = o(\log_\sigma n)$, the space is $H_k(S) + o(n \log \sigma)$. Any succinct data structure that requires $o(n \log \sigma)$ bits in addition to S can thus be replaced by a compressed data structure requiring $nH_k(S) + o(n \log \sigma)$ bits overall, where any access to S is replaced by an access to the novel structure. Their scheme is based on Ziv-Lempel encoding.

In this paper we show how the same result can be achieved by much simpler means. We present an alternative scheme based on semi-static k-th order modeling plus statistical encoding, just as a normal semi-static statistical compressor would process S. By adding some extra structures, we are able of retrieving any substring of S of $\Theta(\log_\sigma n)$ symbols in constant time. Although any statistical encoder works, we obtain the best results (matching exactly those of [18]) using Arithmetic encoding [1]. Furthermore, we show that we can append symbols to S without changing the space complexity, in constant amortized time per symbol.

In addition, we study the applicability of this technique to full-text self-indexes. Compressed self-indexes replace a text $T[1,n]$ by a structure requiring $O(nH_0(T))$ or $O(nH_k(T))$ bits of space. In order to provide efficient pattern matching over T, many of those structures [5, 15, 6] achieve space proportional to $nH_k(T)$ by first applying the Burrows-Wheeler Transform [2] over T, $S[1,n] = bwt(T)$, and then struggling to represent S in efficient form. An additional structure of $o(|S|)$ bits gives the necessary functionality to implement the search. One could thus apply the new structure over S, so that the overall structure requires $nH_k(S) + o(|S|)$ bits. Yet, the relation between $H_k(S)$ and $H_k(T)$ remains unknown. In this paper we move a step forward by proving a positive result: $H_1(S) \leq H_k(T) \log \sigma + o(1)$ for small $k = o(\log_\sigma n)$. Thus we can, for example, achieve essentially the same result of the Run-Length FM-Index [9] just by using the new structure on S, without the involved techniques they use.

Several indexes, however, compress $S = bwt(T)$ by means of a *wavelet tree* [7] on S, $wt(S)$. This is a balanced tree storing several binary sequences. Each such sequence B can be represented using $|B|H_0(B)$ bits of space. If we call $nH_0(wt(S))$ the overall resulting space, it turns out that $nH_0(wt(S)) = nH_0(S)$. A natural idea advocated in [18] is to use a k-th order representation for the binary sequences B, yielding space $nH_k(wt(S))$. Thus the question about the relationship between $H_k(wt(S))$ and $H_k(S)$ is raised. In this paper we exhibit examples where either is larger than the other. In particular, we show that when moving from $wt(S)$ to S, the k-th order entropy grows at least by a factor of $\Theta(\log k)$.

[2] The term $k \log \sigma$ appears as k in [18], but this is a mistake [17]. The reason is that they take from [8] an extra space of the form $\Theta(kt+t)$ as stated in Lema 2.3, whereas the proof in Theorem A.4 gives a term of the form $kt \log \sigma + \Theta(t)$.

2 Background and Notation

Hereafter we assume that $S[1,n] = S_{1,n} = s_1 s_2 \ldots s_n$ is the sequence we wish to encode and query. The symbols of S are drawn from an alphabet $A = \{a_1, \ldots, a_\sigma\}$ of size σ. We write $|w|$ to denote the length of sequence w.

Let $B[1,n]$ be a binary sequence. Function $rank_b(B,i)$ returns the number of times b appears in the prefix $B[1,i]$. Function $select_b(B,i)$ returns the position of the i-th appearance of b within sequence B. Both *rank* and *select* can be computed in constant time using $o(n)$ bits of space in addition to B [11].

2.1 The *k*-th Order Empirical Entropy

The empirical entropy resembles the entropy defined in the probabilistic setting (for example, when the input comes from a Markov source). However, the empirical entropy is defined for any string and can be used to measure the performance of compression algorithms without any assumption on the input [10].

The empirical entropy of k-th order is defined using that of zero-order. This is defined as

$$H_0(S) \;=\; -\sum_{a \in A} \frac{n_S^a}{n} \log_2\left(\frac{n_S^a}{n}\right) \qquad (1)$$

with n_S^a the number of occurrences of symbol a in sequence S. This definition extends to $k > 0$ as follows. Let A^k be the set of all sequences of length k over A. For any string $w \in A^k$, called a context of size k, let w_S be the string consisting of the concatenation of characters following w in S. Then, the k-th order empirical entropy of S is

$$H_k(S) \;=\; \frac{1}{n} \sum_{w \in A^k} |w_S| H_0\left(w_S\right). \qquad (2)$$

The k-th order empirical entropy captures the dependence of symbols upon their context. For $k \geq 0$, $nH_k(S)$ provides a lower bound to the output of any compressor that considers a context of size k to encode every symbol of S. Note that the uncompressed representation of S takes $n \log \sigma$ bits, and that $0 \leq H_k(S) \leq H_{k-1}(S) \leq \ldots \leq H_1(S) \leq H_0(S) \leq \log \sigma$.

Note that a semi-static k-th order *modeler* that yields the probabilities $p_1, p_2, \ldots, p_n$ for the symbols $s_1, s_2, \ldots, s_n$, will actually determine $p_i \approx P(s_i | s_{i-k} \ldots s_{i-1})$ using the formula $p_i = \frac{n_{w_S}^{s_i}}{|w_S|}$, where $w = s_{i-k} \ldots s_{i-1}$. It is not hard to see, by grouping all the terms with the same w in the summation [10, 7], that

$$-\sum_{i=k+1}^{n} p_i \log p_i \;=\; nH_k(S). \qquad (3)$$

2.2 Statistical Encoding

We are interested in the use of semi-static statistical encoders in this paper. Thus, we are given a k-th order modeler as described above, which will yield the probabilities $p_1, p_2, \ldots, p_n$ for each symbol in S, and we will encode the successive symbols of S trying to use $-p_i \log p_i$ bits for s_i. If we reach exactly $-p_i \log p_i$ bits, the overall number of bits produced will be $nH_k(S) + O(k \log n)$, according to Eq. (3).

Different encoders provide different approximations to the ideal $-p_i \log p_i$ bits. The simplest encoder is probably Huffman coding [1], while the best one, from the point of view of the number of bits generated, is Arithmetic coding [1].

Given a statistical encoder E and a semi-static modeler over sequence $S[1, n]$ yielding probabilities $p_1, p_2, \ldots, p_n$, we call $E(S)$ the bitwise output of E for those probabilities, and $|E(S)|$ its bit length. We call $f_k(E, S) = |E(S)| - (-\sum_{1 \leq i \leq n} p_i \log p_i)$ the extra space in bits needed to encode S using E, on top of the entropy of the model. For example, the wasted space of Huffman encoding is bounded by 1 bit per symbol, and thus $f_k(\text{Huffman}, S) < |S|$ (tighter bounds exist but are not useful for this paper [1]). On the other hand, Arithmetic encoding approaches $-p_i \log p_i$ as closely as desired, requiring only at most two extra bits to terminate the whole sequence [1, Section 5.2.6 and 5.4.1]. Thus $f_k(\text{Arithmetic}, S) \leq 2$. Again, we can relate the model entropy of $p_1, p_2, \ldots, p_n$ with the empirical entropy of S using Eq. (3), achieving that, say, Arithmetic coding encodes S using at most $nH_k(S) + O(k \log n) + 2$ bits.

Arithmetic coding essentially expresses S using a number in $[0, 1)$ which lies within a range of size $P = p_1 \cdot p_2 \cdots p_n$. We need $-\log P = -\sum \log p_i$ bits to distinguish a number within that range (plus two extra bits for technical reasons). Thus each new symbol s_i, which appears within its context np_i times, requires $-\log p_i$ bits to be encoded. This totalizes $-n \sum p_i \log p_i + 2$ bits.

There are usually some limitations to the near-optimality achieved by Arithmetic coding in practice [1]. One is that many bits are required to manipulate P, which can be cumbersome. This is mainly alleviated by emitting the most significant bits of the final number as soon as they are known, and thus scaling the remainder of the number again to the range $[0, 1)$ (that is, dropping the emitted bits from our number). Still, some symbols with very low probability may require many bits. To simplify matters, fixed precision arithmetic is used to approximate the real values, and this introduces a very small (yet linear) inefficiency in the coding. In our case, we never run into this problem because, as seen later, we do not encode any sequence that requires more than $\frac{\log n}{2}$ bits. As soon as those bits are not precise enough to represent the encoding, we switch to plain symbol-wise encoding.

Another limitation applies to adaptive encoding, where some kind of aging technique is used to let the model forget symbols that have appeared many positions away in the sequence. In our case this does not apply, as we use semi-static encoding. Finally, we notice that we run into no efficiency problems at all at decoding time, as we will use the $\frac{\log n}{2}$-bit compressed stream as an index to a precomputed table that will directly yield the uncompressed symbols.

2.3 Implementing Succinct Full-Text Self-indexes

A *succinct full-text index* provides fast search functionality using a space proportional to that of the text itself. A less space-demanding index, in particular, using space proportional to that of the compressed text is known as a *compressed full-text index.*Those indexes that contain sufficient the information to recreate the original text are known as *self-indexes.* An example of the latter is the FM-index family [5, 6, 9] based on the Burrows-Wheeler Transform (BWT) [2]. The BWT of a text T, $T^{bwt} = bwt(T)$, is a reversible transformation from strings to strings. For this paper, it is enough to say that T^{bwt} is a permutation of the characters of T which is easier to compress by local optimization methods [10].

Full-text indexes need essentially to perform *symbol rank* queries over T^{bwt}: $Occ_c(T^{bwt}, i)$ is the number of occurrences of character c in $T^{bwt}[1, i]$. This can be done in constant time for very small alphabets [5], but to handle larger alphabets [6] a tool called the *wavelet tree* [7] of $S = T^{bwt}$ is used.

Given a sequence $S[1, n]$ the wavelet tree $wt(S)$ [7] built on S is a perfect binary tree of height $\lceil \log \sigma \rceil$, built on the alphabet symbols, such that the root represents the whole alphabet and each leaf represents a distinct alphabet symbol. If a node v represents alphabet symbols in the range $A^v = [i, j]$, then its left child v_l represents $A^{v_l} = [i, \frac{i+j}{2}]$ and its right child v_r represents $A^{v_r} = [\frac{i+j}{2} + 1, j]$. We associate to each node v the subsequence S^v of S formed by the characters in A^v. However, sequence S^v is not really stored at the node. Instead, we store a bit sequence B^v telling whether characters in S^v go left or right, that is, $B^v_i = 1$ iff $S^v_i \in A^{v_r}$.

The wavelet tree of S requires $nH_0(S) + O(n \log\log n / \log_\sigma n)$ bits of space.

3 A New Entropy-Bound Succinct Data Structure

Given a sequence $S[1, n]$ over an alphabet A of size σ, we encode S into a compressed data structure S' within entropy bounds. To perform all the original operations over S under the RAM model, it is enough to allow extracting any $b = \frac{1}{2} \log_\sigma n$ consecutive symbols of S, using S', in constant time.

3.1 Data Structures for Substring Decoding

We describe our data structure to represent S in essentially $nH_k(S)$ bits, and to permit the access of any substring of size $b = \lfloor \frac{1}{2} \log_\sigma n \rfloor$ in constant time. This structure is built using any statistical encoder E as described in Section 2.2.

Structure. We divide S into blocks of length $b = \lfloor \frac{1}{2} \log_\sigma n \rfloor$ symbols. Each block will be represented using at most $b' = \lfloor \frac{1}{2} \log n \rfloor$ bits (and hopefully less). We define the following sequences indexed by block number $i = 0, \ldots, \lfloor n/b \rfloor$:

- $S_i = S[bi + 1, b(i + 1)]$ is the sequence of symbols forming the i-th block of S.
- $C_i = S[bi - k + 1, bi]$ is the sequence of symbols forming the k-th order context of the i-th block (a dummy value is used for C_0).
- $E_i = E(S_i)$ is the encoded sequence for the i-th block of S, initializing the k-th order modeler with context C_i.
- $\ell_i = |E_i|$ is the size in bits of E_i.

- $\tilde{E}_i = \begin{cases} S_i \text{ if } \ell_i > b' \\ E_i \text{ otherwise} \end{cases}$, is the shortest sequence among E_i and S_i.
- $\tilde{\ell}_i = |\tilde{E}_i| \leq \min(b', \ell_i)$ is the size in bits of $\tilde{E}_i$.

The idea behind $\tilde{E}_i$ is to ensure that no encoded block is longer than b' bits (which could happen if a block contains many infrequent symbols). These special blocks are encoded explicitly.

Our compressed representation of S stores the following information:

- $W[0, \lfloor n/b \rfloor]$: A bit array such that
$$W[i] = \begin{cases} 0 \text{ if } \ell_i > b' \\ 1 \text{ otherwise} \end{cases},$$
with the additional $o(n/b)$ bits to answer rank queries over W in constant time [11].
- $C[1, rank(W, \lfloor n/b \rfloor)]$: $C[rank(W, i)] = C_i$, that is, the k-th order context for the i-th block of S iff $\ell_i \leq b'$, with $1 \leq i \leq \lfloor n/b \rfloor$.
- $U = \tilde{E}_0 \tilde{E}_1 \ldots \tilde{E}_{\lfloor n/b \rfloor}$: A bit sequence obtained by concatenating all the variable-length $\tilde{E}_i$.
- $T : A^k \times 2^{b'} \longrightarrow 2^b$: A table defined as $T[\alpha, \beta] = \gamma$, where α is any context of size k, β represents any encoded block of b' bits at most, and γ represents the decoded form of β, truncated to the first b symbols (as less than the b' bits will be usually necessary to obtain the b symbols of the block).
- Information to answer where each $\tilde{E}_i$ starts within U. We group together every $c = \lceil \log n \rceil$ consecutive blocks to form superblocks of size $\Theta(\log^2 n)$ and store two tables:
  - $R_g[0, \lfloor n/(bc) \rfloor]$ contains the absolute position of each superblock.
  - $R_l[0, \lfloor n/b \rfloor]$ contains the relative position of each block with respect to the beginning of its superblock.

3.2 Substring Decoding Algorithm

We want to retrieve $q = S[i, i + b - 1]$ in constant time. To achieve this, we take the following steps:

1. We calculate $j = i$ div b and $j' = (i + b - 1)$ div b.
2. We calculate $h = j$ div c, $h' = (j+1)$ div c and $u = U[R_g[h] + R_l[j], R_g[h'] + R_l[j + 1] - 1]$, then
   - if $W[j] = 0$ then we have $S_j = u$.
   - if $W[j] = 1$ then we have $S_j = T[C[rank(W, j)], u']$, where u' is u padded with $b' - |u|$ dummy bits.

   We note that $|u| \leq b'$ and thus it can be manipulated in constant time.
3. If $j' \neq j$ then we repeat Step 2 for $j' = j + 1$ and obtain $S_{j'}$. Then, $q = S_j[i - jb + 1, b]\ S_{j'}[1, i - jb]$ is the solution.

Lemma 1. *For a given sequence $S[1, n]$ over an alphabet A of size σ, we can access any substring of S of b symbols in $O(1)$ time using the data structures presented in Section 3.1.*

3.3 Space Requirement

Let us now consider the storage size of our structures.

- We use the constant-time solution to answer the rank queries [11] over W, totalizing $\frac{2n}{\log_\sigma n}(1+o(1))$ bits.
- Table C requires at most $\frac{2n}{\log_\sigma n}k\log\sigma$ bits.
- Let us consider table U. $|U| = \sum_{i=0}^{\lfloor n/b \rfloor}|\tilde{E}_i| \leq \sum_{i=0}^{\lfloor n/b \rfloor}|E_i| = nH_k(S) + O(k\log n) + \sum_{i=0}^{\lfloor n/b \rfloor} f_k(E, S_i)$, which depends on the statistical encoder E used. For example, in the case of Huffman coding, we have $f_k(\text{Huffman}, S_i) < b$, and thus we achieve $nH_k(S) + O(k\log n) + n$ bits. For the case of Arithmetic coding, we have $f_k(\text{Arithmetic}, S_i) \leq 2$, and thus we have $nH_k(S) + O(k\log n) + \frac{4n}{\log_\sigma n}$ bits, as described in Section 2.2.
- The size of T is $\sigma^k 2^{b'} b\log\sigma = \sigma^k\ n^{1/2}\ \frac{\log n}{2}$ bits.
- Finally, let us consider tables R_g and R_l. Table R_g has $\lceil n/(bc)\rceil$ entries of size $\lceil\log n\rceil$, totalizing $\frac{2n}{\log_\sigma n}$ bits. Table R_l has $\lceil n/b\rceil$ entries of size $\lceil\log(b'c)\rceil$, totalizing $\frac{4n\log\log n}{\log_\sigma n}$ bits.

By considering that any substring of $\Theta(\log_\sigma n)$ symbols can be extracted in constant time by applying $O(1)$ times the procedure of Section 3.2, we have the final theorem.

Theorem 1. *Let $S[1,n]$ be a sequence over an alphabet A of size σ. Our data structure uses $nH_k(S) + O(\frac{n}{\log_\sigma n}(k\log\sigma + \log\log n))$ bits of space for any $k < (1-\epsilon)\log_\sigma n$ and any constant $0 < \epsilon < 1$, and it supports access to any substring of S of size $\Theta(\log_\sigma n)$ symbols in $O(1)$ time.*

Note that, in our scheme, the size of T can be neglected only if $k < (\frac{1}{2}-\epsilon)\log_\sigma n$, but this can be pushed as close to 1 as desired by choosing $b = \frac{1}{s}\log_\sigma n$ for constant $s \geq 2$.

Corollary 1. *The previous structure takes space $nH_k(S) + o(n\log\sigma)$ if $k = o(\log_\sigma n)$.*

These results match exactly those of [18], once one corrects their k to $k\log\sigma$ as explained. Note that we are storing some redundant information that can be eliminated. The last characters of block S_i are stored both within $\tilde{E}_i$ and as C_{i+1}. Instead, we can choose to explicitly store the first k characters of *all* blocks S_i, and encode only the remaining $b-k$ symbols, $S_i[k+1, b]$, either in explicit or compressed form. This improves the space in practice, but in theory it cannot be proved to be better than the scheme we have given.

4 Supporting Appends

We can extend our scheme to support appending symbols, maintaining the same space and query complexity, with each appended symbol having constant amortized cost. Assume our current static structure holds n symbols. We use a buffer of $n' = n/\log n$ symbols where we store symbols explicitly. When the buffer is full we use our entropy-bound data structure (EBDS, Section 3) to represent those n' symbols and then we empty the buffer. We repeat this until we have $\log n$ EBDS. At this moment we reencode all the structures plus our original n symbols, generating a new single EBDS, and restart the process with $2n$ symbols.

Data structures. We describe the additional structures needed to append symbols to the EBDS.

- $BF[1, n']$ is the sequence of at most $n' = n/\log n$ uncompressed symbols.
- AP_i is the i-th EBDS, with $0 \leq i \leq N$. $N \leq \log n$ is the number of EBDS we currently have. We call AS_i the sequence AP_i represents. AP_0 is the original EBDS. So $|AS_0| = n$ and $|AS_i| = n/\log n$, $i > 0$.

Substring decoding algorithm. We want to retrieve $q = S[i, i + b - 1]$. To achieve this, we take the following steps:

- We algebraically calculate the indexes $0 \leq t \leq t' \leq N+1$ where the positions i (for t) and $i + b - 1$ (for t') belong; $N + 1$ represents BF. The case when part of q belongs to BF is trivially solved because the symbols are explicitly represented in BF.
- If $t = t'$ we obtain q as in Section 3.2. Otherwise, we calculate the local indexes t_{off} and t'_{off} where q starts in structure AP_t and finishes in $AP_{t'}$, respectively. We decode q_1 as the last $n' - t_{off} + 1 \leq b$ symbols of AP_t and q_2 as the first $t'_{off} \leq b$ symbols of $AP_{t'}$. Finally, we obtain $q = q_1 q_2$.

Construction time. after we reencode everything we have that $n/2$ symbols have been reencoded once, $n/4$ symbols twice, $n/8$ symbols 3 times and so on. The total number of reencodings is $\sum_{i\geq 1} n\frac{i}{2^i} = 2n$. On the other hand, we are using a semi-static statistical encoder, which takes $O(1)$ time to encode each symbol. Thus each symbol has a worst-case amortized appending cost of $O(1)$.

Space requirement. Let us now consider the storage of the appended structures.

- Table BF requires $n/\log_\sigma n$ bits
- Each AP_i is an EBDS, using $|AS_i|H_k(AS_i) + O(\frac{|AS_i|}{\log_\sigma |AS_i|}(k \log \sigma + \log\log |AS_i|))$ bits of space.

Lemma 2. *The space requirement of all AP_i, for $0 \leq i \leq N$, is $\sum_{i=0}^{\log n} |AP_i| \leq |S\ AS_1 \ldots AS_N| H_k(S\ AS_1 \ldots AS_N) + O(\frac{n}{\log_\sigma n}(k \log \sigma + \log\log n))$ $+O(\sigma^{k+1} \log^2 n) + O(k \log^2 n)$ bits, where $n = |S| \leq |S\ AS_1 \ldots AS_N|/2$.*

Proof. Consider summing any two entropies (recall Eqs. (1) and (2)).

$$
\begin{aligned}
&|AS_1|H_k(AS_1) + |AS_2|H_k(AS_2) = \\
&\quad = \textstyle\sum_{w \in A^k} |w_{AS_1}|H_0(w_{AS_1}) + \sum_{w \in A^k} |w_{AS_2}|H_0(w_{AS_2}) \\
&\quad \leq \textstyle\sum_{w \in A^k} \left(\log \binom{|w_{AS_1}|}{|n^{a_1}_{AS_1}|,|n^{a_2}_{AS_1}|,\ldots,|n^{a_\sigma}_{AS_1}|} + \log \binom{|w_{AS_2}|}{|n^{a_1}_{AS_2}|,|n^{a_2}_{AS_2}|,\ldots,|n^{a_\sigma}_{AS_2}|} \right) + \\
&\qquad O(\sigma^{k+1} \log n) \\
&\quad \leq \textstyle\sum_{w \in A^k} \log \binom{|w_{AS_1}|+|w_{AS_2}|}{|n^{a_1}_{AS_1}|+|n^{a_1}_{AS_2}|,|n^{a_2}_{AS_1}|+|n^{a_2}_{AS_2}|,\ldots,|n^{a_\sigma}_{AS_1}|+|n^{a_\sigma}_{AS_2}|} + O(\sigma^{k+1} \log n) \\
&\quad \leq |AS_1AS_2|H_k(AS_1AS_2) + O(\sigma^{k+1} \log n) + O(k \log n)
\end{aligned}
$$

where $O(\sigma^{k+1} \log n)$ comes from the relationship between the zero-order entropy and the combinatorials, and $O(k \log n)$ comes from considering the symbols in the border between AS_1 and AS_2. Note that $\sigma^{k+1} \log n = o(n)$ if $k < (1 - \epsilon) \log_\sigma n$. Then the lemma follows by adding up the $N \leq \log n$ EBDSs.

Theorem 2. *The structure of Theorem 1 supports appending symbols in constant amortized time and retains the same space and query time complexities, being n the current length of the sequence.*

5 Application to Full-Text Indexing

In this section we give some positive and negative results about the application of the technique to full-text indexing, as explained in the Introduction. We have a text $T[1, n]$ over alphabet A and wish to compress a transformed version X of T with our technique. Then, the question is how does $H_k(X)$ relate to $H_k(T)$.

5.1 The Burrows-Wheeler Transform

The Burrows-Wheeler Transform, $S = bwt(T)$, is used by many compressed full-text self-indexes [5, 6, 9]. We have introduced it in Section 2.3.

We show that there is a relationship between the k-th order entropy of a text T and the first order entropy of $S = bwt(T)$. For this sake, we will compress S with a first-order compressor, whose output size is an upper bound to $nH_1(S)$.

A *run* in S is a maximal substring formed by a single letter. Let $rl(S)$ be the number of runs in S. In [9] they prove that $rl(S) \leq nH_k(T) + \sigma^k$ for any k. Our first-order encoder exploits this property, as follows:

- If $i > 1$ and $s_i = s_{i-1}$ then we output bit 0.
- Otherwise we output bit 1 followed by s_i in plain form ($\log \sigma$ bits).

Thus we encode each symbol of S by considering only its preceding symbol. The total number of bits is $n + rl(S) \log \sigma \leq n(1 + H_k(S) \log \sigma + \frac{\sigma^k \log \sigma}{n})$. The latter term is negligible for $k < (1 - \epsilon) \log_\sigma n$, for any $0 < \epsilon < 1$. On the other hand, the total space obtained by our first-order encoder cannot be less than $nH_1(S)$. Thus we get our result:

Lemma 3. *Let $S = bwt(T)$, where $T[1, n]$ is a text over an alphabet of size σ. Then $H_1(S) \leq 1 + H_k(T) \log \sigma + o(1)$ for any $k < (1 - \epsilon) \log_\sigma n$ and any constant $0 < \epsilon < 1$.*

We can improve this upper bound if we use Arithmetic encoding to encode the 0 and 1 bits that distinguish run heads. Their zero-order probability is $p = H_k(T) + \frac{\sigma^k}{n}$, thus the 1 becomes $-p \log p - (1-p)\log(1-p) \le 1$. Likewise, we can encode the run heads s_i up to their zero-order entropy. These improvements, however, do not translate into clean formulas.

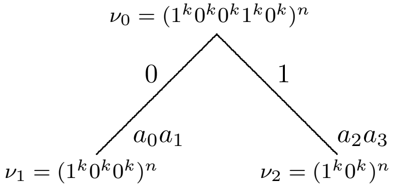
This shows, for example, that we can get (at least) about the same results of the Run-Length FM-Index [9] by compressing $bwt(T)$ using our structure.

5.2 The Wavelet Tree

Several FM-Index variants [9, 6] use wavelet trees to represent $S = bwt(T)$, while others [7] use them for other purposes. As explained in Section 2.3, $wt(S)$ is composed of several binary sequences. By compressing each such sequence B to $|B|H_0(B)$ bits, one achieves $nH_0(S)$ bits overall. The natural question is, thus, whether we can prove any bound on the overall space if we encode sequences B to $|B|H_k(B)$ bits. We present next two negative examples.

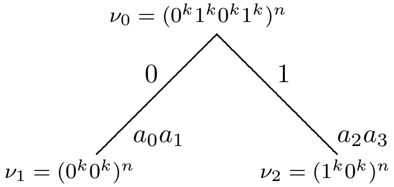
- First we show a case where $H_k(S) < H_k(wt(S))$. We choose $S = (a_3^k a_1^k a_0^k a_2^k a_0^k)^n$, then

$wt(S) =$



  Let us compute $H_k(S)$ according to Section 2.1. Note that $H_0(w_S) = 0$ for all contexts except $w = a_0^k$, where $w_S = a_2(a_3a_2)^{n-1}\$$, being "\$" a sequence terminator. Thus $|w_S| = 2n$ and $H_0(w_S) = -\frac{n}{2n}\log\frac{n}{2n} - \frac{n-1}{2n}\log\frac{n-1}{2n} - \frac{1}{2n}\log\frac{1}{2n} = 1 + O(\frac{\log n}{n})$. Therefore $H_k(S) \simeq \frac{2}{5k}$.
  On the other hand, $H_k(wt(S)) = \sum_{i=0}^{2} H_k(\nu_i) \simeq \underbrace{\frac{2}{5k}\log k}_{\nu_0} + \underbrace{\frac{1}{3k} + \frac{\log k}{3k}}_{\nu_1}$, as $H_k(\nu_2) \simeq 0$.
  Therefore, in this case, $H_k(S) < H_k(wt(S))$, by a $\Theta(\log k)$ factor.
- Second, we show a case where $H_k(S) > H_k(wt(S))$. Now we choose $S = (a_0^k a_3^k a_0^k a_2^k)^n$, then

$wt(S) =$



  In this case, $H_k(S) \simeq \frac{2}{4k}$ and $H_k(wt(S)) = \sum_{i=0}^{2} H_k(\nu_i) = O(\frac{\log n}{n})$. Thus $H_k(S) > H_k(wt(S))$ by a factor of $\Theta(n/(k \log n))$.

Lemma 4. *The ratio between the k-th order entropy of the wavelet tree representation of a sequence S, $H_k(wt(S))$, and that of S itself, $H_k(S)$, can be at least $\Omega(\log k)$. More precisely, $H_k(wt(S))/H_k(S)$ can be $\Omega(\log k)$ and $H_k(S)/H_k(wt(S))$ can be $\Omega(n/(k \log n))$.*

What is most interesting is that $H_k(wt(S))$ can be $\Theta(\log k)$ times larger than $H_k(S)$. We have not been able to produce a larger gap. Whether $H_k(wt(S)) = O(H_k(S) \log k)$ remains open.

6 Conclusions

We have presented a scheme based on k-th order modeling plus statistical encoding to convert any succinct data structure on sequences into a compressed data structure. This structure permits retrieving any string of S of $\Theta(\log_\sigma n)$ symbols in constant time. This is an alternative to the first work achieving the same result [18], which is based on Ziv-Lempel compression. We also show how to append symbols to the original sequence within the same space complexity and with constant amortized cost per appended symbol. This method also works on the structure presented in [18].

We also analyze the behavior of this technique when applied to full-text self-indexes, as advocated in [18]. Many compressed self-indexes achieve space proportional to $nH_k(T)$ by first applying the Burrows-Wheeler Transform [2] over T, $S[1, n] = bwt(T)$. In this paper, we show a relationship between the entropies of $H_1(S)$ and $H_k(T)$. More precisely, $H_1(S) \leq H_k(T) \log \sigma + o(1)$ for small $k = o(\log_\sigma n)$. On the other hand, several indexes represent $S = bwt(T)$ as a wavelet tree [7] on S, $wt(S)$. We show in this paper that $H_k(wt(S))$ can be at least $\Theta(\log k)$ times larger than $H_k(S)$. This means that, by applying the new technique to compress wavelet trees, we have no guarantee of compressing the original sequence more than $n \min(H_0(S), O(H_k(T) \log k))$. Yet, we do have guarantees if we compress S directly.

There are several future challenges on k-th order entropy-bound data structures: (i) making them fully dynamic (we have shown how to append symbols); (ii) better understanding how the entropies evolve upon transformations such bwt or wt; (iii) testing them in practice.

Acknowledgment. We thank K. Sadakane and R. Grossi for providing us article [18] and for confirming the correctness of Footnote 2.

References

1. T. Bell, J. Cleary, and I. Witten. *Text compression.* Prentice Hall, 1990.
2. M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.
3. P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Structuring labeled trees for optimal succinctness, and beyond. In *Proc. 46st FOCS*, 2005.

4. P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Compressing and searching XML data via two zips. In *Proc. 15th WWW'06*, 2006.
5. P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proc. 41st FOCS*, pages 390–398, 2000.
6. P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. An alphabet-friendly FM-index. In *Proc. 11th SPIRE*, LNCS 3246, pages 150–160. Springer, 2004. Extended version to appear in *ACM TALG*.
7. R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proc. 14th SODA*, pages 841–850, 2003.
8. R. Kosaraju and G. Manzini. Compression of low entropy strings with Lempel-Ziv algorithms. *SIAM Journal on Computing*, 29(3):893–911, 1999.
9. V. Mäkinen and G. Navarro. Succinct suffix arrays based on run-length encoding. *Nordic Journal of Computing*, 12(1):40–66, 2005.
10. G. Manzini. An analysis of the Burrows-Wheeler transform. *Journal of the ACM*, 48(3):407–430, 2001.
11. I. Munro. Tables. In *Proc. 16th FSTTCS*, LNCS v. 1180, pages 37–42, 1996.
12. I. Munro, R. Raman, V. Raman, and S. Rao. Succinct representations of permutations. In *Proc. 30th ICALP*, pages 345–356, 2003.
13. I. Munro and V. Raman. Succinct representation of balanced parentheses, static trees and planar graphs. In *Proc. 38th FOCS*, pages 118–126, 1997.
14. I. Munro and S. S. Rao. Succinct representations of functions. In *Proc. 31th ICALP*, pages 1006–1015, 2004.
15. G. Navarro. Indexing text using the Ziv-Lempel trie. *Journal of Discrete Algorithms (JDA)*, 2(1):87–114, 2004.
16. R. Raman, V. Raman, and S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. 13th SODA*, pages 233–242, 2002.
17. K. Sadakane and R. Grossi. Personal communication, 2005.
18. K. Sadakane and R. Grossi. Squeezing succinct data structures into entropy bounds. In *Proc. 17th SODA*, pages 1230–1239, 2006.

Dynamic Entropy-Compressed Sequences and Full-Text Indexes

Veli Mäkinen[1,⋆] and Gonzalo Navarro[2,⋆⋆]

[1] Department of Computer Science, University of Helsinki, Finland
vmakinen@cs.helsinki.fi
[2] Department of Computer Science, University of Chile
gnavarro@dcc.uchile.cl

Abstract. Given a sequence of n bits with binary zero-order entropy H_0, we present a dynamic data structure that requires $nH_0 + o(n)$ bits of space, which is able of performing *rank* and *select*, as well as inserting and deleting bits at arbitrary positions, in $O(\log n)$ worst-case time. This extends previous results by Hon et al. [ISAAC 2003] achieving $O(\log n/ \log\log n)$ time for *rank* and *select* but $\Theta(\text{polylog}(n))$ amortized time for inserting and deleting bits, and requiring $n + o(n)$ bits of space; and by Raman et al. [SODA 2002] which have constant query time but a static structure. In particular, our result becomes the *first* entropy-bound dynamic data structure for *rank* and *select* over bit sequences.

We then show how the above result can be used to build a dynamic full-text self-index for a collection of texts over an alphabet of size σ, of overall length n and zero-order entropy H_0. The index requires $nH_0 + o(n \log \sigma)$ bits of space, and can count the number of occurrences of a pattern of length m in time $O(m \log n \log \sigma)$. Reporting the *occ* occurrences can be supported in $O(occ \log^2 n \log \sigma)$ time, paying $O(n)$ extra space. Insertion of text to the collection takes $O(\log n \log \sigma)$ time per symbol, which becomes $O(\log^2 n \log \sigma)$ for deletions. This improves a previous result by Chan et al. [CPM 2004]. As a consequence, we obtain an $O(n \log n \log \sigma)$ time construction algorithm for a compressed self-index requiring $nH_0 + o(n \log \sigma)$ bits *working space* during construction.

1 Introduction and Related Work

The study of compressed data structures aims to represent classical structures like trees, graphs, text indexes, etc., in the smallest possible space without challenging the functionality of the structure; the original operations should be supported efficiently without decompressing the whole structure.

One of the most commonly appearing structures are the *rank* and *select* dictionaries for bit vectors: $rank(A, i)$ gives the number of bits set up to position i in bit vector $A = a_1 a_2 \cdots a_n$, $a_k \in \{0, 1\}$; $select(A, j)$ is the inverse, giving the position i containing the j-th bit set in A. We study the dynamic version of these dictionaries, where one can *insert* or *delete* a bit at any position.

⋆ Funded by the Academy of Finland under grant 108219.

⋆⋆ Partially Funded by Fondecyt Grant 1-050493, Chile.

M. Lewenstein and G. Valiente (Eds.): CPM 2006, LNCS 4009, pp. 306–317, 2006.

Dynamic *rank* and *select* dictionaries have been studied before [13, 9], as a special case of so-called *Searchable Partial Sums with Indels* problem. The best current result [9] requires $n + o(n)$ bits of space, $O(\log_b n)$ time for *rank* and *select*, and $O(b)$ amortized time for *insert* and *delete*, for $b = \Omega(\text{polylog}(n))$.

In this paper we improve some aspects of this result by achieving $O(\log n)$ worst-case time complexity for all the operations, over a data structure that requires $nH_0+o(n)$ bits of space, where $0 \leq H_0 \leq 1$ is the binary zero-order entropy of A. This space has been previously achieved only for static data structures [14], with constant time for *rank* and *select* but no support for updates. Ours is the *first* entropy-bound dynamic data structure answering *rank* and *select* queries. Moreover, our result works under weaker assumptions on the RAM model than the previous results on dynamic settings.

The indexed string matching problem is that of, given a long text $T[1, n]$ over an alphabet Σ of size σ, building a data structure called *full-text index* on it, to solve two types of queries: (a) Given a short pattern $P[1, m]$ over Σ, *count* the occurrences of P in T; (b) *locate* those *occ* positions in T. There are several classical full-text indexes requiring $O(n \log n)$ bits of space which can answer counting queries in $O(m \log \sigma)$ time (like suffix trees [1]) or $O(m + \log n)$ time (like suffix arrays [11]). Both locate each occurrence in constant time once the counting is done. Similar complexities are obtained with modern compressed data structures [6, 8, 7], requiring space $nH_k(T) + o(n \log \sigma)$ bits (for some small k), where $H_k(T) \leq \log \sigma$ is the k-th order empirical entropy of T.[1] These indexes are often called *entropy-compressed self-indexes* refering to their space requirement and to their ability to work without the text.

The main building block in entropy-compressed self-indexes is function *rank*, or more precisely, its generalization to non-binary sequences: $rank_c(A, i)$ counts the number of times symbol c appears in a given sequence A up to position i. Our dynamic entropy-compressed binary *rank* structure can be extended into a dynamic entropy-compressed *symbol rank* structure using *wavelet trees* [8]. This dynamic structure takes $nH_0 + o(n \log \sigma)$ bits of space, where H_0 is the empirical zero-order entropy of the sequence. It supports the same operations as binary *rank* with $O(\log \sigma)$ slowdown in queries. Plugging this structure in the dynamic self-index of Chan, Hon, and Lam [4], we obtain a dynamic entropy-compressed self-index occupying $nH_0 + o(n \log \sigma)$ bits on a text collection of overall length n. Our structure can count the number of occurrences of a pattern of length m in time $O(m \log n \log \sigma)$. Insertion of a text to the collection takes $O(\log n \log \sigma)$ time per symbol. Deletion takes $O(\log^2 n \log \sigma)$ time. These operations are $O(\log \sigma)$ times slower than with the original index of Chan et al., but we obtain a significant space saving: Their index takes $O(n\sigma)$ bits while ours takes $O(n \log \sigma)$ bits in general.

As a consequence, we obtain an $O(n \log n \log \sigma)$ time construction algorithm for a compressed self-index called *succinct suffix array* (SSA) [10] requiring $nH_0 + o(n \log \sigma)$ bits *working space* during construction (the same as the final structure). This is the first construction algorithm for a *FM-index* [6] variant,

[1] In this paper log stands for $\log_2$.

whose working space depends on the entropy. For another self-index called *LZ-index* [12], there is a recent entropy-bound construction algorithm [2].

2 Definitions

To simplify notation, we ignore roundings. When refering to number of bits, we use simply $\log n$ to refer to $\lfloor(\log n) + 1\rfloor$. That is, $\log\log n$ bits means actually $\lfloor(\log\lfloor(\log n)+1\rfloor)+1\rfloor$ bits. Similarly $(\log n)/2$ is the integer nearest to $\lfloor(\log n)+1\rfloor/2$, and so on.

We assume our sequence $A = a_1 \dots a_n$ to be drawn from an alphabet $\{0, 1, \dots \sigma - 1\}$. Let n_c denote the number of occurrences of symbol c in A, i.e., $n_c = |\{i \mid a_i = c\}|$. Then the zero-order *empirical entropy* is defined as $H_0(A) = \sum_{0 \le c < \sigma} \frac{n_c}{n} \log \frac{n}{n_c}$.

We assume a random access machine with word size w; typical arithmetic operations on w-bit integers are assumed to take constant time. We make the standard assumption that $\log n = \Theta(w)$ (in the full version, we show that this can be weakened to $\log n = O(w)$ without changing the results).

We study the following problems:

The *Dynamic Sequence with Indels* problem is to maintain a (virtual) sequence $A = a_1 \dots a_n$, $a_i \in \{0, 1, \dots, \sigma - 1\}$, supporting the operations:

- $rank_c(A, i)$ returns the number of occurrences of symbol c in $a_1 \cdots a_i$;
- $select_c(A, j)$ returns the index i containing j-th occurrence of c;
- $insert(A, c, i)$ inserts $c \in \{0, 1, \dots \sigma - 1\}$ between a_i and a_{i+1}; and
- $delete(A, i)$ deletes a_i from the sequence.

The *Dynamic Bit Vector with Indels* problem is a restriction of the above to alphabet $\{0, 1\}$. Then we use short-hand notation $rank(A, i) = rank_1(A, i)$ and $select(A, i) = select_1(A, i)$. Notice that $rank_0(A, i) = i - rank_1(A, i)$, but same does not apply for $select_0(A, j)$; we consider this case separately.

3 Previous Results

3.1 Entropy-Bound Structures for Bit Vectors

Raman et al. [14] proposed a data structure to solve *rank* and *select* queries in constant time over a static bit vector $A = a_1 \dots a_n$ with binary zero-order entropy H_0. The structure requires $nH_0 + o(n)$ bits.

The idea is to split A into *superblocks* $S_1 \dots S_{n/s}$ of $s = \log^2 n$ bits. Each superblock S_i is in turn divided into $2 \log n$ blocks $B_i(j)$, of $t = (\log n)/2$ bits each, thus $1 \le j \le s/t$. Each such block B_i is said to belong to *class* c if it has exactly c bits set, for $0 \le c \le t$. For each class c, a universal table G_c of $\binom{t}{c}$ entries is precomputed. Each entry corresponds to a possible block belonging to class c, and it stores all the local *rank* answers for that block. Overall all the G_c tables add up $2^t = \sqrt{n}$ entries, and $O(\sqrt{n}\ \mathrm{polylog}(n))$ bits.

Each block $B_i(j)$ of the sequence is represented by a pair $D_i(j) = (c, o)$, where c is its class and o is the index of its corresponding entry in table G_c. A block of class c thus requires $\log(c+1) + \log\binom{t}{c}$ bits. The first term is $O(\log\log n)$, whereas all the second terms add up $nH_0 + O(n/\log n)$ bits. To see this, note that $\log\binom{t}{c_1} + \log\binom{t}{c_2} \le \log\binom{2t}{c_1+c_2}$, and that $nH_0 \ge \log\binom{t(n/t)}{c_1+\ldots+c_{n/t}}$. The pairs $D_i(j)$ are of variable length and are all concatenated into a single sequence.

Each superblock S_i stores a pointer P_i to its first block description in the sequence (that is, the first bit of $D_i(1)$) and the *rank* value at the beginning of the superblock, $R_i = rank(A,(i-1)s)$. P and R add up $O(n/\log n)$ bits. In addition, S_i contains s/t numbers $L_i(j)$, giving the initial position of each of its blocks in the sequence, relative to the beginning of the superblock. That is, $L_i(j)$ is the position of $D_i(j)$ minus P_i. Similarly, S_i stores s/t numbers $Q_i(j)$ giving the *rank* value at the beginning of each of its blocks, relative to the beginning of the superblock. That is, $Q_i(j) = rank(A,(i-1)s+(j-1)t) - R_i$. As those relative values are $O(\log n)$, sequences L and Q require $O(n\log\log n/\log n)$ bits.

To solve $rank(A,p)$, we compute the corresponding superblock $i = 1 + \lfloor p/s \rfloor$ and block $j = 1 + \lfloor (p-(i-1)s)/t \rfloor$. Then we add the *rank* value of the corresponding superblock, R_i, the relative *rank* value of the corresponding block, $Q_i(j)$, and complete the computation by fetching the description (c,o) of the block where p belongs (from bit position $P_i + L_i(j)$) and performing a (precomputed) local *rank* query in the universal table, $rank(G_c(o), p-(i-1)s-(j-1)t)$.

The overall space requirement is $nH_0 + O(n\log\log n/\log n)$ bits, and *rank* is solved in constant time. We do not cover *select* because it is not necessary to follow this paper.

3.2 Dynamic Structures for Bit Vectors

Hon et al. [9] show how to handle a bit vector $A = a_1 \ldots a_n$ in $n + o(n)$ bits of space, so that *rank* and *select* can be solved in $O(\log_b n)$ time, while insertions and deletions to the sequence can be handled in $O(b)$ amortized time, for any parameter $b = \Omega(\text{polylog}(n))$. Hence, they provide a solution to the *Dynamic Bit Vector with Indels* problem. Their main structure is a weight-balanced B-tree (WBB) [5, 13].

Our goal is to obtain $nH_0 + o(n)$ bits of space and $O(\log n)$ worst-case time for all the operations above. We build over a simplified version of their structure, which uses standard balanced trees and achieves $O(\log n)$ time and $O(n)$ bits of space [4]. We assume red-black trees in the following, as we later use the property of constant number of rotations to rebalance the tree.

Consider a balanced binary tree on A whose left-most leaf contains bits $a_1a_2\cdots a_{\log n}$, second left-most leaf contains bits $a_{\log n+1}a_{\log n+2}\cdots a_{2\log n}$, and so on. Each node v contains counters $p(v)$ and $r(v)$ telling the number of positions stored and the number of bits set in the subtree rooted at v, respectively. Note that this tree, with all its $\log n$-size pointers and counters, requires $O(n)$ bits.

To perform $rank(A,i)$, we enter the tree to find the leaf containing position i. We start with $rank \leftarrow 0$. If $p(left(v)) \ge i$ we enter the left subtree, otherwise we enter the right subtree with $i \leftarrow i - p(left(v))$ and $rank \leftarrow rank + r(left(v))$. In $O(\log n)$ time we reach the desired leaf and complete the rank query in $O(\log n)$

time by scanning the bit sequence corresponding to that node. For *select* we proceed similarly, except that the roles of $p()$ and $r()$ are reversed. For $select_0$ the computation is analogous.

Insertions and deletions are handled by entering to the correct leaf like in *rank*, and replacing its bit-sequence with the new content. Then the $p(v)$ and $r(v)$ counters in the path from the leaf to the root are changed accordingly. To keep the tree balanced, the leaves can be split and merged on updates: When a leaf is updated to contain $2\log n$ bits, it is split into two leaves each containing $\log n$ bits. When a leaf is updated to contain $(\log n)/2$ bits, it is merged with its sibling. If this merging produces a leaf with more than $2\log n - 1$ bits, this leaf is again split into two equal-size halves. After splitting and merging, the tree needs to be rebalanced and the counters updated in the nodes on the way to the root.

To obtain $n + o(n)$ bits of space instead of $O(n)$, we can use the superblock-block hierarchy from the previous section: The tree is built on the superblocks, i.e., each leaf corresponds to a $\log^2 n$-length superblock of A. A precomputed table G is used to answer *rank* queries for each $(\log n)/2$-length bit-sequence. Then one can scan through the $\log^2 n$-length superblock summing up *rank* answers to each $(\log n)/2$-length block in constant time until reaching the block containing the query position. The remaining bits can be read one-by-one to complete the *rank* query inside a superblock in $O(\log n)$ time. Answering *select* is similar. The problem, however, is that we cannot allocate $2\log^2 n$ space for a superblock that will hold only $\log^2 n$ bits, as otherwise we could spend as much as $2n$ bits for the blocks. To obtain $n + o(n)$ space one must force very tight usage of the leaf space: spending $(1+\epsilon)\log^2 n$ bits, for any constant $\epsilon > 0$, is forbidden. This is problematic because bit insertions on a leaf would cause an overflow propagation to the next leaves that cannot be fixed with a constant number of block splits. The complete solution is quite involved, and we present it in the next sections, already coupled with the technique to achieve $nH_0 + o(n)$ bits (the reader can easily simplify it to obtain the $n + o(n)$ bits solution that already improves [9] in some aspects). We must also pay attention to the case where $\log n$ changes.

4 Dynamic Entropy-Bound Structures for Bit Vectors

We design a data structure to represent a bit sequence $A = a_1 \ldots a_n$ of binary zero-order entropy H_0, using $nH_0 + o(n)$ bits of space and performing operations *rank*, *select*, *insert* and *delete* all in $O(\log n)$ time. Hence, we show that the *Dynamic Bit Vector with Indels* problem can be solved using less than $\Theta(n)$ space on compressible sequences, without sacrificing the logarithmic time bound on the operations.

4.1 High-Level Hierarchy

We maintain the universal tables G_c as in Section 3.1, but this time they store only the explicit content of the blocks. This still requires $O(\sqrt{n}\ \mathrm{polylog}(n))$ bits of space.

We also divide A into blocks and superblocks, except that this time superblocks do not span a constant amount of bits of A, but of its (compressed) representation. That is, each superblock S will maintain $s = f(n) \log n$ bits (for some $f(n) = O(\text{polylog}(n))$ to be determined later), and this will correspond to as many (complete) blocks as can be represented with s bits considering their D, L, and Q entries. Blocks are still of $t = (\log n)/2$ bits. Since each L and Q value requires $O(\log \log n)$ bits, and a D entry may require up to $t + O(\log \log n)$ bits, a superblock may handle from $O(f(n))$ to $O(f(n) \log n / \log \log n)$ blocks. Similarly, a block can have up to $O(\log n)$ unused space, because the next block does not fit in it. This unused space adds up $O(n/f(n))$ bits overall. Otherwise the space usage is the same as in the static case.

4.2 Operations Inside a Superblock

A $\mathit{rank}(S_i, p)$ query inside a superblock is handled in $O(\log n)$ time by adding the corresponding $Q_i(j)$ entry to $\mathit{rank}(G_c(o), p')$, where $j = 1 + \lfloor p/t \rfloor$, $p' = p - (j-1)t$, and (c, o) is found at position $L_i(j)$ in the memory area of the superblock. Here, $\mathit{rank}(G_c(o), p')$ is computed in $O(\log n)$ time by a bitwise scan over $G_c(o)$. A $\mathit{select}(S_i, p)$ query is solved in time $O(\log n)$ by binary searching Q_i for the largest *rank* value not exceeding p, and then a bitwise scan for query $\mathit{select}(G_c(o), p')$. Computation for select_0 is analogous.

To insert a bit q at position p of S_i, we essentially recompute the superblock by brute force. However, we must be careful so as to work only $O(f(n))$ time per superblock. For example, we cannot decompress, modify, and then recompress the superblock because that way we could work $O(f(n) \log n / \log \log n)$ time (as the uncompressed superblock can be up to $O(f(n) \log^2 n / \log \log n)$ bits long).

We first determine the block j where the insertion is to take place, that is, $j = 1 + \lfloor p/t \rfloor$. All the $D_i(1 \ldots j-1)$, $L_i(1 \ldots j)$, and $Q_i(1 \ldots j)$ entries are direcly copied into a new memory area where the updated representation of S_i is to be built. On a RAM machine this copying can be done in $O(f(n))$ time.

Modifying each block in constant time. The block $D_i(j) = (c, o)$ to modify starts at position $L_i(j)$ within the superblock. We use $G_c(o)$ to obtain the uncompressed content of this block. Let $B = b_1 \ldots b_t$ be the bits of this block, and let $p' = p - (j-1)t$ be the position to insert the bit q within B. Thus we compute $B' = b_1 \ldots b_{p'-1} q b_{p'+1} \ldots b_{t-1}$ and save b_t for later. To compress B' in constant time we use another universal table H, which is indexed by numbers of t bits and stores, at each entry, the c and o value of the corresponding binary vector. H requires $O(\sqrt{n}\ \text{polylog}(n))$ bits, and gives $H(B') = (c', o')$ in constant time. This description $D_i(j)' = (c', o')$ is appended at the updated copy of S_i we are constructing.

We must now take care of the remaining blocks to the right. We have a bit b_t that fell off B. In addition we must shift the values L_i to the right by $|o'| - |o|$ and Q_i by $q - b_t$. To perform all this propagation in $O(f(n))$ time, we use yet another universal table $J(b, l, q, x)$, where b is a bit to insert at the beginning of the next block, $l = O(\text{polylog}(n))$ is the next L_i value, $q = O(\text{polylog}(n))$ is the next Q_i value, and x is the sequence of the first t bits of $D_i(j+1\ldots)$. If $J(b,l,q,x)=(D',L',Q',b',l',q')$, this means that, if we decode from x as many integral blocks as we can, append bit b at the beginning, and recode them, we obtain sequence D'. Their corresponding positions, starting in l, are encoded in L', and their corresponding ranks, starting at q, are encoded in Q'. Furthermore, bit b' falls off at the end of D', the next L_i value should be l', and the next Q_i value should be q'. Another table $V(x)=r'$ tells us how many bits we could use from x, so we can advance in the processing of sequence D_i by r' bits.

Therefore, after having modified the j-th block, we start by assigning $r=L_i(j)$ and obtain $J(b_t, L_i(j)+|o'|-|o|, Q_i(j)+q-b_t, D_i[r\ldots])$ and $V(D_i[r\ldots])$. Then we copy D', L', and Q' to the updated version of S_i we are building, and continue with $J(b',l',q',D_i[r+r'\ldots])$ and $V(D_i[r+r'\ldots])$, until processing the whole superblock. At the end, we rewrite S with its updated version. Note that we still have one overflown bit.

Tables J and V require $O(\sqrt{n}\ \mathrm{polylog}(n))$ bits, and they process $\Theta(\log n)$ bits of the superblock in constant time (each two applications it must be possible to process at least t bits of D_i), plus the time necessary to write the modified superblock. As there are $O(f(n)\log n)$ bits in the superblock, we can process the whole superblock in $O(f(n))$ time using J and V, plus the size of the new superblock measured in $\Theta(\log n)$-size chunks.

Let us consider how much can the superblock grow by the insertion of a single bit. If a new block is started, we need $O(\log\log n)$ more bits. In addition, the D entry of a block may grow because its (c,o) descriptor changes. The maximum value of $\log\binom{t}{c+1}-\log\binom{t}{c}$ is $\log t$, achieved when $c=0$. Propagated over $O(f(n)\log n/\log\log n)$ blocks, the sequence of D values might be increased by $O(f(n)\log n)$ bits. This is as large as a whole superblock, and means that a single bit insertion might double the size of the superblock in some extreme cases. For example, if the sequence is $(0^t1^t)^r$, all the c values will be 0 or t, and the o indexes will be empty, thus we will store $f(n)\log n/\log\log n$ blocks in the superblock. If we now insert a 1 at the beginning of the sequence, each o descriptor becomes $\log t = O(\log\log n)$ bits wide, which adds up $f(n)\log n$ extra bits. Still, the new superblock is also $O(f(n)\log n)$ size and can be output using J and V in $O(f(n))$ time.

Overflow to the next superblock. At the end of the operation, it might be that the new sequence does not fit within the s bits allocated to the superblock. If so, we take out as many blocks as necessary from the end of the superblock, so as to move them to the beginning of the next superblock. We have seen that we might have to move up to $O(f(n)\log n)$ bits. In addition we must insert the excess bit at the next superblock (after the blocks we are moving, if any).

The process completely rewrites the next superblock S'. We move the overflowing D, L and Q entries to the beginning of S', but the L and Q values moved must be shifted. This has to be done by chunks of $\Theta(\log n)$ bits using a universal table to ensure $O(f(n))$ overall time. Then we must insert the carry bit at the beginning of the original entries of S', which in addition must be shifted to account for the blocks moved from the overflowing superblock. This can be carried out in $O(f(n))$ time using tables J and V. Yet, this bit insertion may produce another $O(f(n) \log n)$-bits overflow, in addition to the original $O(f(n) \log n)$ bits. The exponential growth is avoided because we can create a new superblock as soon as we have enough overflown bits. The propagation can thus be carried out in $O(f(n))$ time per superblock rewritten/created. Yet, we still need a mechanism to prevent that the propagation continues too far.

Limiting the propagation of overflows. Every $f(n)$ superblocks we permit the formation of a *partial* superblock, which reserves $f(n) \log n$ bits but might be partially full, and in addition permits having at the end an underfilled block (with less than t bits). This partial block needs some care to be correctly handled, such as padding it with dummy bits to obtain a representation in G, taking care of its real length, and so on. Partial superblocks waste $O(n/f(n))$ bits overall, and ensure that we never traverse more than $f(n)$ superblocks in the overflow process. Thus the overall insertion work is $O(f(n)^2)$.

To ensure the desired density of partial superblocks, we first check whether there is a partial superblock among the next $2f(n)$ superblocks. If there is one, we carry out the propagation up to it. Otherwise, we propagate $f(n)$ superblocks and create a new partial superblock. In both cases we work over $O(f(n))$ superblocks, and guarantee that every partial superblock is $f(n)$ superblocks away from any other. We note that partial superblocks may end up overflowing, at which point they are not considered partial anymore. We can create a new partial superblock immediately following it, as it is already ensured that the new partial superblock is far away from others.

Note that, when a partial superblock overflows, its last block can still be partial. This is not a real problem, because we are creating next a new partial superblock containing that partial block at the end, plus sufficient complete blocks from the end of the overflowing superblock.

Controlling the underflow. For deletions we proceed similarly, using a table J' very similar to J: J' deletes the first bit of the blocks represented by x and adds bit b at their end. The bit b we give to J' is obtained in constant time using G, as the first bit of $D_i[r + V(x)...]$. Also, we ensure that superblocks are as full as possible. If some space is left at the end of the superblock, we check that the first blocks from the next superblock can be moved back, and propagate the underflow similarly as the overflows. If we reach a partial superblock, no further propagation of underflows is necessary. If after $2f(n)$ attempts we do not reach a partial superblock, we permit the underflow at the $f(n)$-th superblock and declare it partial. On the other hand, a partial superblock that gets empty must be deleted.

Note that, because of the changes in $|o|$ widths, an insertion can actually produce an underflow and a deletion can produce an overflow. This is not problematic. Overall (still not considering how to manage superblocks), we have $O(n/f(n))$ extra space and $O(f(n)^2)$ insertion/deletion time. We can choose, for example, $f(n) = \sqrt{\log n}$ to obtain $O(\log n)$ time and $O(n/\sqrt{\log n})$ space.

4.3 Global *Rank* and *Select*

We have seen how to perform *rank* and *select* inside a superblock in $O(\log n)$ time. To perform the global *rank* and *select* we can use the balanced tree on the superblocks as explained in Section 3.2. Finding the correct superblock takes $O(\log n)$ time, hence the whole query takes $O(\log n)$ time.

Inserting and deleting bits from this tree requires rewriting the $p()$ and $r()$ values from the affected superblock(s) through the root. Creation and deletion of superblocks and internal tree nodes is easily handled together with the maintenance of $r()$ and $p()$. We note, however, that we permit that a single update affects $O(f(n))$ superblocks. Once the leaf to be inserted or deleted is located, the red-black tree needs constant time to rebalance, so this adds up $O(f(n))$ time per insertion. As for propagating the red-black coloring and updating the $r()$ and $p()$ values through the root, note that those $O(f(n))$ superblocks are contiguous in the tree and therefore their total number of ancestors do not exceed $f(n) + O(\log n) = O(\log n)$. It is not hard to organize the updates to work $O(\log n)$ time overall.

4.4 Changing $\log n$

Our result so far assumes that $\log n$ stays constant during the operations. This value fixes the superblock/block hierarchy and the global preprocessed tables. This assumption can be removed in two ways: (1) performing a global rebuild whenever $\log n$ changes; (2) maintaining partial structures ready for values $(\log n) -1$, $\log n$, and $(\log n) + 1$ (which we call the *previous*, *current*, and *next*).

Approach (1) is easy to implement. We can rebuild all structures in $O(n)$ time when necessary to accommodate the new value of $\log n$. Amortized over all insertions and deletions, this costs only $O(1)$ time per operation.

Approach (2) is more complex but is inspired on a standard mechanism to convert amortized complexity into worst-case complexity. The idea is to split the current elements among the *previous*, *current*, and *next* structures, so that the first elements are in *previous*, the last are in *next*, and *current* holds the middle elements. It is trivial to run *rank* and *select* queries on this split structure. Initially, all the elements are in *current*, and the other two are empty. Upon an insertion, the size of *next* must grow by 2 and *previous* must shrink by 1 unless it is already empty; a deletion must cause the opposite effect; and *current* acts as a variable-size buffer.

To achieve this, let us denote $x \rightarrow y$ or $x \leftarrow y$ the movement of one element among structures, for $x, y \in \{p, c, n\}$, e.g. $p \leftarrow c$ means moving the first element of *current* to *previous*. If the source structure is empty, the movement is just ignored. Then, we insert (delete) in the proper structure and then, depending on where the insertion (deletion) point lies, we move elements as follows:

– *previous*: $p \rightarrow c$, $p \rightarrow c$, $c \rightarrow n$, $c \rightarrow n$ ($c \leftarrow n$, $c \leftarrow n$, $p \leftarrow c$, $p \leftarrow c$).
– *current*: $p \rightarrow c$, $c \rightarrow n$, $c \rightarrow n$ ($c \leftarrow n$, $c \leftarrow n$, $p \leftarrow c$).
– *next*: $p \rightarrow c$, $c \rightarrow n$ ($c \leftarrow n$, $p \leftarrow c$).

It is easy to see that, after n net insertions, *next* will hold all the $2n$ elements, and that after $n/2$ net deletions, *prev* will hold all the $n/2$ remaining elements. This is true even if the insertions and deletions are intermixed. When *next* holds all the elements, it becomes *current* and the new *previous* and *next* structures are empty; similarly when *previous* holds all the elements. At those points, precisely, $\log n$ has changed its value. The space requirement is still $nH_0 + o(n)$.

The only remaining problem is that we do not have time to build the new G, J, etc. tables, as we would need them immediately available to handle the new *next* or *previous* structure. For this sake, we maintain all the time 5 versions of those tables, for $(\log n) - 2 \ldots (\log n) + 2$. As we move to $(\log n) + 1$, we have immediately available the required tables for $(\log n)$, $(\log n) + 1$ and $(\log n) + 2$. The construction of the tables for $(\log n) + 3$ is easily spread during the next $O(\sqrt{n} \text{ polylog } n)$ insertions, building just a new cell at the time. These insertions are much less than the necessary to make $\log n$ grow again. If, instead, $\log n$ shrinks back, we just abandon the partial table construction. Thus we achieve the following result:

Theorem 1. *The Dynamic Bit Vector with Indels problem can be solved using* $nH_0 + O(n/\sqrt{\log n})$ *bits of space supporting the operations rank, select, insert, and delete in* $O(\log n)$ *worst-case time.*

5 Extensions and Applications

5.1 General Alphabets

Theorem 1 can be extended to the *Dynamic Sequence with Indels* problem using wavelet trees [8]. The wavelet tree is a balanced binary tree built on the alphabet symbols, containing bit vectors in its internal nodes. When these node bit vectors are preprocessed for the *Dynamic Bit Vector with Indels* problem (taking some care on the sub-linear terms [7]), we obtain the following result.

Theorem 2. *The Dynamic Sequence with Indels problem can be solved using* $nH_0 + o(n \log \sigma)$ *bits of space supporting the operations rank, select, insert, and delete, in* $O(\log n \log \sigma)$ *worst-case time. Here* H_0 *is the zero-order entropy of the sequence and* σ *its alphabet size.*

5.2 Dynamic Full-Text Indexes

Chan, Hon, and Lam [4] show how to use a solution to *Dynamic Sequence with Indels* problem to obtain a dynamic full-text index. The idea is to simulate the *backward search* algorithm of Ferragina and Manzini [6]: After preprocessing a text T, the backward search algorithm finds the number of occurrences of a given pattern P in T in $O(|P|)$ steps. One step essentially makes two *rank* queries to the *Burrows-Wheeler* transform [3] of T, $A = bwt(T)$. We note $H_0(A) = H_0(T)$ as the transform is a permutation.

They [4] show that one can dynamically maintain a collection of texts, by keeping a data structure supporting *rank*, *insert* and *delete* on the Burrows-Wheeler transform of the concatenation of the texts in the collection (symbol 0 is reserved for separating two texts). We can as a black box replace their `COUNT` structure (that takes $O(n\sigma)$ bits, supporting the operations in $O(\log n)$ time) with the structure in Theorem 2 to obtain the following result.

Theorem 3. *A dynamic collection of texts $\mathcal{C} = \{T_1, T_2, \ldots, T_m\}$, where each $T_i \in \{1, 2, \ldots \sigma - 1\}^*$, can be maintained in $nH_0(\mathcal{C}) + o(n \log \sigma)$ bits supporting counting of occurrences of a pattern P in $O(|P| \log n \log \sigma)$ time, inserting a text T in $O(|T| \log n \log \sigma)$ time, and deleting a text T in $O(|T| \log^2 n \log \sigma)$ time. Here n is the length of concatenation $C = 0T_1 0T_2 \cdots 0T_m$ of $\mathcal{C}$, and $H_0(\mathcal{C}) = H_0(C)$. We assume that $\mathcal{C}$ starts initially empty.*

The index can be extended to support reporting the occurrences using the `MARK` structure of [4]. This structure takes $O(n)$ bits, and with our *rank* structure can be used to report each occurrence in $O(\log^2 n \log \sigma)$ time.

As a consequence, we obtain an $O(n \log n \log \sigma)$ time construction algorithm for a compressed self-index requiring $nH_0 + o(n \log \sigma)$ bits *working space* during construction: This is obtained by just inserting text T to the empty collection. This index can be converted to a more efficient static self-index, like a *succinct suffix array* [10], within the same time bound. The static structure requires the same $nH_0 + o(n \log \sigma)$ bits, but the counting of pattern occurrences can then be done in $O(|P|)$ time if $\sigma = O(\mathrm{polylog}(n))$, and $O(|P| \log \sigma / \log \log n)$ in general.

6 Conclusions

We have introduced the *first* entropy-bound dynamic data structure answering *rank* and *select* queries on bit arrays. We can represent a vector of n bits with zero-order entropy H_0 using $nH_0 + o(n)$ bits of space, so that we can answer *rank* and *select* queries, as well as inserting and deleting bits, in $O(\log n)$ worst-case time. This improves in several aspects the best existing solution to the *Searchable Partial Sums with Indels Problem* [9] for the case of bit sequences: we achieve logarithmic worst-case bounds for insertions and deletions (previous solution achieved $\Theta(\mathrm{polylog}(n))$ amortized time) and require less than n bits on compressible sequences. We apply these results to compressed full-text self-indexing, achieving the first FM-index-like structure that can be built within zero-order entropy space. This index permits insertion and deletion of texts with better bounds than previous solutions [4].

Our result works under weaker assumptions on the RAM model than the previous results on dynamic settings. We assumed $\log n = \Theta(w)$ to simplify matters; in the full version, this assumption will be loosened to $\log n = O(w)$. This complicates the memory allocation, as we can not e.g. represent tree pointers in $O(\log n)$ bits, when $\log n = o(w)$. However, our results remain unchanged under the weaker model.

The succinct suffix array (SSA) constructed using the dynamic index can currently only count the pattern occurrences, unless paying $O(n)$ bits extra space for the MARK structure of [4]. We plan to study whether this could be improved to $o(n)$ bits. We plan also to study general searchable partial sums, and larger alphabets with multiary wavelet trees [7] to improve the time bounds in Theorem 2 by a $\log \log n$ factor.

References

1. A. Apostolico. The myriad virtues of subword trees. In *Combinatorial Algorithms on Words*, NATO ISI Series, pages 85–96. Springer-Verlag, 1985.
2. D. Arroyuelo and G. Navarro. Space-efficient construction of LZ-index. In *Proc. ISAAC'05*, LNCS 3827, pages 1143–1152, 2005.
3. M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. Technical Report Technical Report 124, Digital Equipment Corporation, 1994.
4. W.-L. Chan, W.-K. Hon, and T.-W. Lam. Compressed index for a dynamic collection of texts. In *Proc. CPM'04*, LNCS 3109, pages 445–456, 2004.
5. P. Dietz. Optimal algorithms for list indexing and subset rank. In *Proc. WADS'89*, pages 39–46, 1989.
6. P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proc. FOCS'00*, pages 390–398, 2000.
7. P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representation of sequences and full-text indexes. *ACM Transactions on Algorithms*, 2006. To appear. Preliminary versions in *Proc. SPIRE 2004* and Tech. Rep. TR/DCC-2004-5, Dept. of Computer Science Univ. of Chile, `ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/sequences.ps.gz`.
8. R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proc. SODA'03*, pages 841–850, 2003.
9. W.-K. Hon, K. Sadakane, and W.-K. Sung. Succinct data structures for searchable partial sums. In *Proc. ISAAC'03*, LNCS 2906, pages 505–516, 2003.
10. V. Mäkinen and G. Navarro. Succinct suffix arrays based on run-length encoding. *Nordic Journal of Computing*, 12(1):40–66, 2005.
11. U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, pages 935–948, 1993.
12. G. Navarro. Indexing text using the Ziv-Lempel trie. *Journal of Discrete Algorithms (JDA)*, 2(1):87–114, 2004.
13. R. Raman, V. Raman, and S. Srinivasa Rao. Succinct dynamic data structures. In *Proc. WADS'01*, pages 426–437, 2001.
14. R. Raman, V. Raman, and S. Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. SODA'02*, pages 233–242, 2002.

Reducing the Space Requirement of LZ-Index*

Diego Arroyuelo[1], Gonzalo Navarro[1], and Kunihiko Sadakane[2]

[1] Dept. of Computer Science, Universidad de Chile
{darroyue, gnavarro}@dcc.uchile.cl
[2] Dept. of Computer Science and Communication Engineering,
Kyushu University, Japan
sada@csce.kyushu-u.ac.jp

Abstract. The LZ-index is a *compressed full-text self-index* able to represent a text $T_{1...u}$, over an alphabet of size $\sigma = O(\text{polylog}(u))$ and with k-th order empirical entropy $H_k(T)$, using $4uH_k(T) + o(u \log \sigma)$ bits for any $k = o(\log_\sigma u)$. It can report all the *occ* occurrences of a pattern $P_{1...m}$ in T in $O(m^3 \log \sigma + (m + occ) \log u)$ worst case time. Its main drawback is the factor 4 in its space complexity, which makes it larger than other state-of-the-art alternatives. In this paper we present two different approaches to reduce the space requirement of LZ-index. In both cases we achieve $(2 + \epsilon)uH_k(T) + o(u \log \sigma)$ bits of space, for any constant $\epsilon > 0$, and we simultaneously improve the search time to $O(m^2 \log m + (m + occ) \log u)$. Both indexes support displaying any subtext of length ℓ in optimal $O(\ell / \log_\sigma u)$ time. In addition, we show how the space can be squeezed to $(1 + \epsilon)uH_k(T) + o(u \log \sigma)$ to obtain a structure with $O(m^2)$ average search time for $m \geqslant 2 \log_\sigma u$.

1 Introduction and Previous Work

Given a sequence of symbols $T_{1...u}$ (the text) over an alphabet Σ of size σ, and given another (short) sequence $P_{1...m}$ (the *search pattern*) over Σ, the *full-text search problem* consists in finding all the *occ* occurrences of P in T.

Applications of full-text searching include text databases in general, which typically contain natural language texts, DNA or protein sequences, MIDI pitch sequences, program code, etc. A central goal of modern text databases is *to provide fast access to the text using as little space as possible*. Yet, these goals are opposed: to provide fast access we must build an index on the text, increasing the space requirement. The main motivation of using little space is to store the indexes of very large texts entirely in main memory. This can compensate for significant CPU time to access them. In recent years there has been much research on *compressed text databases*, focusing on techniques to represent the text and the index using little space, yet permitting efficient text searching.

A concept related to text compression is the k-th order empirical entropy of a sequence T, denoted $H_k(T)$ [9]. The value $uH_k(T)$ is a lower bound to the

* Supported in part by CONICYT PhD Fellowship Program (first author) and Fondecyt Grant 1-050493 (second author) and the Grant-in-Aid of the Ministry of Education, Science, Sports and Culture of Japan (third author).

M. Lewenstein and G. Valiente (Eds.): CPM 2006, LNCS 4009, pp. 318–329, 2006.

number of bits needed to compress T using any compressor that encodes each symbol considering only the context of k symbols that precede it in T. It holds $0 \leqslant H_k(T) \leqslant H_{k-1}(T) \leqslant \cdots \leqslant H_0(T) \leqslant \log \sigma$ (log means $\log_2$ in this paper).

The current trend on compressed text databases is *compressed full-text self-indexing*. A *self-index* allows searching and retrieving any part of the text without storing the text itself. A *compressed index* requires space is proportional to the compressed text size. Then a compressed full-text self-index *replaces* the text with a more space-efficient representation of it, which at the same time provides indexed access to the text. This is an unprecedented breakthrough in text indexing and compression. Some compressed self-indexes are [16, 4, 7, 5].

The LZ-index [14] is another compressed full-text self-index, based on the Ziv-Lempel [18] parsing of the text. If the text is parsed into n phrases by the LZ78 algorithm, then the LZ-index takes $4n \log n(1 + o(1))$ bits of space, which is 4 times the size of the compressed text, i.e. $4uH_k(T) + o(u \log \sigma)$ bits, for any $k = o(\log_\sigma u)$ [8, 4]. The LZ-index answers queries in $O(m^3 \log \sigma + (m+occ) \log n)$ worst case time. The index can also reproduce a context of length ℓ around an occurrence found (and in fact any sequence of phrases) in $O(\ell \log \sigma)$ time, or obtain the whole text in time $O(u \log \sigma)$.

However, in practice the space requirement of LZ-index is relatively large compared with competing schemes: 1.2–1.6 times the text size versus 0.6–0.7 and 0.3–0.8 times the text size of *CS-Array* [16] and *FM-index* [4], respectively. Yet, the LZ-index is faster to report and to display the context of an occurrence. Fast displaying of text substrings is very important in self-indexes, as the text is not available otherwise.

In this paper we study how to reduce the space requirement of LZ-index, using two different approaches. The first one, a *navigational scheme* approach, consists in reducing the redundancy among the different data structures that conform the LZ-index. These data structures allow us moving among data representations. In this part we define new data structures allowing the same navigation, yet reducing the original redundancy. In the second approach we combine the balanced parentheses representation of Munro and Raman [13] of the LZ78 trie with the *xbw transform* of Ferragina et al. [3], whose powerful operations are useful for the LZ-index search algorithm.

Despite these approaches are very different, in both cases we achieve $(2 + \epsilon)uH_k(T) + o(u \log \sigma)$ bits of space, for any constant $\epsilon > 0$, and we simultaneously *improve* the search time to $O(m^2 \log m + (m + occ) \log n)$ (worst case). In both cases we also present a version requiring $(1 + \epsilon)uH_k(T) + o(u \log \sigma)$ bits, with average search time $O(m^2)$ if $m \geqslant 2 \log_\sigma n$. In all cases, the worst case time to display a context of length ℓ around any occurrence found is optimal $O(\ell / \log_\sigma u)$.

Note that, just as LZ-index, our data structures are the only compressed full-text self-indexes of size $O(uH_k(T))$ able of spending $O(\log n)$ time per occurrence reported, if $\sigma = \Theta(\text{polylog}(u))$. Other data structures achieving the same or better complexity for reporting occurrences either are of size $O(uH_0(T))$ bits [16], or they achieve it for constant-size alphabets [4], or for quite large alphabets ($\log \sigma = \Theta(\log n)$) [7, Theorem 4.1]. The case $\sigma = O(\text{polylog}(u))$, which represents moderate-size alphabets, is very common in practice and does not fit in the above cases. Our data structures are not competitive against schemes requiring about the same space [7, 5] for counting the number of occurrences of P in T. Yet, in many practical situations, it is necessary to report the occurrence positions as well as displaying their contexts. In this aspect, LZ-index is superior.

2 The LZ-Index Data Structure

Assume that the text $T_{1\dots u}$ has been compressed using the LZ78 [18] algorithm into $n+1$ phrases[1] $T = B_0 \dots B_n$, such that $B_0 = \varepsilon$ (the empty string); $\forall k \neq \ell$, $B_k \neq B_\ell$; and $\forall k \geqslant 1$, $\exists \ell < k$, $c \in \Sigma$, $B_k = B_\ell \cdot c$. To ensure that B_n is not a prefix of another B_i, we append to T a special symbol "\$" $\notin \Sigma$, assumed to be smaller than any other symbol. We say that i is the *phrase identifier* corresponding to B_i, $0 \leqslant i \leqslant n$. The following data structures conform the LZ-index [14]:

LZTrie: The trie of all the phrases $B_0 \dots B_n$. Given the properties of LZ78 compression, this trie has exactly $n+1$ nodes, each corresponding to a string.
RevTrie: The trie of all the reverse strings $B_0^r \dots B_n^r$. In this trie there could be internal nodes not representing any phrase. We call these nodes "*empty*".
Node: A mapping from phrase identifiers to their node in *LZTrie*.
Range: A data structure for two-dimensional searching in the space $[0 \dots n] \times [0 \dots n]$. We store the points $\{(revpos(B_k^r), pos(B_{k+1})), k \in 0 \dots n-1\}$, where *revpos* is the lexicographic position in $\{B_0^r \dots B_n^r\}$ and *pos* is the lexicographical position in $\{B_0 \dots B_n\}$. For each such point, the corresponding k value is stored.

Each of these four structures requires $n \log n(1+o(1))$ bits of space if they are represented succinctly, for example, using the balanced parentheses representation [13] for the tries. For *Range*, a data structure of Chazelle [2] permits two-dimensional range searching in a grid of n pairs of integers in the range $[0 \dots n] \times [0 \dots n]$, answering queries in $O((occ+1)\log n)$ time, where occ is the number of occurrences reported, and requiring $n \log n(1+o(1))$ bits. As $n \log u = uH_k(T) + O(kn \log \sigma) \leqslant u \log \sigma$ for any k [8], the final size of the LZ-index is $4uH_k(T) + o(u \log \sigma)$ bits for $k = o(\log_\sigma u)$. The succinct representation given in the original work [14] implements (among others) the operations $parent(x)$ (which gets the parent of node x) and $child(x, \alpha)$ (which gets the child of node x with label $\alpha \in \Sigma$) both in $O(\log \sigma)$ time for *LZTrie*, and $O(\log \sigma)$ and $O(h \log \sigma)$ time respectively for *RevTrie*, where h is the depth of node x in *RevTrie*. The operation $ancestor(x, y)$, which is used to ask if node x is an ancestor of node y, is implemented in $O(1)$ time both in *LZTrie* and *RevTrie*. These operations are basically based on rank/select operations on bit vectors. Given a bit vector $\mathcal{B}_{1\dots n}$, we define the function $rank_0(\mathcal{B}, i)$ (similarly $rank_1$) as the number of 0s (1s) occurring up to the i-th position of $\mathcal{B}$. The function $select_0(\mathcal{B}, i)$ (similarly $select_1$) is defined as the position of the i-th 0 (1) in $\mathcal{B}$.

[1] According to [18], $\sqrt{u} \leqslant n \leqslant \frac{u}{\log_\sigma u}$; thus, $n \log u \leqslant u \log \sigma$ always holds.

These operations can be supported in constant time and requiring $n + o(n)$ bits [11], or $H_0(\mathcal{B}) + o(n)$ bits [15].

Let us consider now the search algorithm for a pattern $P_{1...m}$ [14]. We distinguish three types of occurrences of P in T, depending on the phrase layout:

1. The occurrence lies inside a single phrase (there are occ_1 occurrences of this type)**.** Given the properties of LZ78, every phrase B_k containing P is formed by a shorter phrase B_ℓ concatenated to a symbol c. If P does not occur at the end of B_k, then B_ℓ contains P as well. We want to find the shortest possible phrase B in the LZ78 referencing chain for B_k that contains the occurrence of P. This phrase B finishes with the string P, hence it can be easily found by searching for P^r in *RevTrie* in $O(m^2 \log \sigma)$ time. Say we arrive at node v. Any node v' descending from v in *RevTrie* (including v itself) corresponds to a phrase terminated with P. Thus we traverse and report all the subtree of the *LZTrie* node corresponding to each v'. Occurrences of type 1 are located in $O(m^2 \log \sigma + occ_1)$ time;
2. The occurrence spans two consecutive phrases, B_k and B_{k+1}, such that a prefix $P_{1...i}$ matches a suffix of B_k and the suffix $P_{i+1...m}$ matches a prefix of B_{k+1} (there are occ_2 occurrences of this type): P can be split at any position, so we have to try them all. The idea is that, for every possible split, we search for the reverse pattern prefix in *RevTrie* and for the pattern suffix in *LZTrie*. Now we have two ranges, one in the space of reversed strings (phrases finishing with the first part of P) and one in that of the normal strings (phrases starting with the second part of P), and need to find the phrase pairs $(k, k+1)$ such that k is in the first range and $k+1$ is in the second range. This is what the range searching data structure is for. Occurrences of type 2 are located in $O(m^3 \log \sigma + (m + occ_2) \log n)$ time; and
3. The occurrence spans three or more phrases, $B_k \dots B_\ell$, such that $P_{i...j} = B_{k+1} \dots B_{\ell-1}$, $P_{1...i-1}$ matches a suffix of B_k and $P_{j+1...m}$ matches a prefix of B_ℓ (there are occ_3 occurrences of this type): For this part, the LZ78 algorithm guarantees that every phrase represents a different string. Hence, there is at most one phrase matching $P_{i...j}$ for each choice of i and j. This fact severely limits the number of occurrences of this class that may exist, $occ_3 = O(m^2)$. The idea is to identify maximal concatenations of phrases $P_{i...j} = B_k \dots B_\ell$ contained in the pattern, and thus determine whether B_{k-1} finishes with $P_{1...i-1}$ and $B_{\ell+1}$ starts with $P_{j+1...m}$. If this is the case we can report an occurrence. We first search for every pattern substring in *LZTrie*, in $O(m^2 \log \sigma)$ time. Then, the $O(m^2)$ maximal concatenations of phrases are obtained in $O(m^2 \log m)$ worst case time and $O(m^2)$ time on average. Finally, each of those maximal concatenations is verified in $O(m \log \sigma)$ time using operation *parent* for B_k. Overall, occurrences of type 3 are located in $O(m^3 \log \sigma)$ time.

Note that each of the $occ = occ_1 + occ_2 + occ_3$ possible occurrences of P lies exactly in one of the three cases above. Overall, the total search time to report the occ occurrences of P in T is $O(m^3 \log \sigma + (m + occ) \log n)$. Finally, we can uncompress and display the text of length ℓ surrounding any occurrence reported in $O(\ell \log \sigma)$ (as long as this context spans an integral number of phrases) time, and uncompress the whole text T in $O(u \log \sigma)$ time.

3 LZ-Index as a Navigation Scheme

In the practical implementation of LZ-index [14, see Tech.Report], the *Range* data structure is replaced by *RNode*, which is a mapping from phrase identifiers to their node in *RevTrie*. Now occurrences of type 2 are found as follows: For every possible split $P_{1...i}$ and $P_{i+1...m}$ of P, assume the search for $P^r_{1...i}$ in *RevTrie* yields node v_{rev}, and the search for $P_{i+1...m}$ in *LZTrie* yields node v_{lz}. Then, we check each phrase k in the subtree of v_{rev} and report it if $Node[k+1]$ descends from v_{lz}. Each such check takes constant time. Yet, if the subtree of v_{lz} has less elements, we do the opposite: Check phrases from v_{lz} in v_{rev}, using *RNode*. Unlike when using *Range*, now the time to solve occurrences of type 2 is proportional to the smallest subtree size among v_{rev} and v_{lz}, which can be arbitrarily larger than the number of occurrences reported. That is, by using *RNode* we have no worst-case guarantees at search time. However, the average search time for occurrences of type 2 is $O(n/\sigma^{m/2})$. This is $O(1)$ for long patterns, $m \geqslant 2\log_\sigma n$. The *RNode* data structure requires $uH_k(T) + o(u \log \sigma)$ bits, and so this version of LZ-index also requires $4uH_k(T) + o(u \log \sigma)$ bits, for any $k = o(\log_\sigma u)$.

Both *LZTrie* and *RevTrie* use originally the *balanced parentheses* representation [13], in which every node, represented by a pair of opening and closing parentheses, encloses its subtree. When we replace *Range* by *RNode* structure, the result is actually a "navigation" scheme that permits us moving back and forth from trie nodes to the corresponding *preorder positions*[2], both in *LZTrie* and *RevTrie*. The phrase identifiers are common to both tries and permit moving from one trie to the other.

Figure 1 (left) shows the navigation scheme. Dashed arrows are asymptotically "for free" in terms of memory, since they are followed by applying *rank* on the corresponding parentheses structure. The other four arrows are in fact the four main components in the space usage of the index: Array of phrase identifiers in *LZTrie* (*ids*) and in *RevTrie* (*rids*), and array of *LZTrie* nodes for each phrase (*Node*) and *RevTrie* nodes for each phrase (*RNode*). The structure is symmetric and we can move from any point to any other.

The structure, however, is redundant, in the sense that the number of arrows is not minimal. We start by defining the following reduced scheme for LZ-index:

LZTrie: The Lempel-Ziv trie, implemented with the following data structures:

- $par_{0...2n}$ and *lets*: The tree shape of *LZTrie* according to the DFUDS representation [1], which requires $2n + n\lceil\log\sigma\rceil + o(n) + O(\log\log\sigma)$ bits to support the operations *parent*(x), *child*(x, α), *subtree size* (including the root of the subtree), and *node degree*, all of them in $O(1)$ time. It also supports the operation

[2] In the representation [13], the preorder position of a node is the number of opening parentheses before the one representing the node. This is $rank_0$ at the node position in the bit sequence representing the parentheses, if bit 1 represents ')'.

child(x, i) in constant time, which gets the i-th child of node x. To get this representation, we perform a preorder traversal on the trie, and for every node reached we write its degree in unary using parentheses (for example, 3 reads '`((()`' and it is writen '`0001`'), What we get is almost a balanced parentheses representation (we only need to add a fictitious '`(`' at the beginning of the sequence). A node of degree d is represented by the position of the first of the $(d + 1)$ parentheses corresponding to the node. Given a node in this representation, say at position i, its preorder position can be computed by $rank_1(par, i)$. Given a preorder position p, the corresponding node is computed by $select_1(par, p) + 1$. With this representation we can compute all the operations required by *LZTrie* [14] in $O(1)$ time, including $ancestor(x, y)$ [3]. The symbols labeling the arcs of the trie are represented implicitly. We denote by $lets(i)$ the symbol corresponding to the node at position $select_0(par, i) + 1$ (i.e., the symbol with preorder position i), which is computed in constant time.

- $ids_{0...n}$: The array of LZ78 phrase identifiers in preorder. We use the representation of Munro et al. [12] for *ids* such that the inverse permutation ids^{-1} can be computed in $O(1/\epsilon)$ time, requiring $(1 + \epsilon)n \log n$ bits[4].

RevTrie: The *PATRICIA* tree [10] of the reversed LZ78 phrases, which is implemented with the following data structures

- $rpar_{0...2n'}$ and *rlets*: The *RevTrie* structure represented using DFUDS [1], compressing empty unary paths and thus ensuring $n' \leqslant 2n$ nodes, because empty non-unary nodes still exist. The space requirement is $2n' + n'\lceil\log\sigma\rceil + o(n') + O(\log\log\sigma)$ bits to support the same functionalities as *LZTrie*. - $B_{0...n'}$: A bit vector supporting *rank* and *select* queries, and requiring $n'(1 + o(1))$ bits [11]. The j-th bit of B is 1 iff the node with preorder position j in *rpar* is not an empty node, otherwise the bit is 0. Given a position p in *rpar* corresponding to a *RevTrie* node, the corresponding bit in B is $B[rank_1(rpar, p)]$.

- $R_{0...n}$: A mapping from *RevTrie* preorder positions to *LZTrie* preorder positions defined as $R[i] = ids^{-1}(rids[i])$. R is implemented using the succinct data structure for permutations of Munro et al. [12], requiring $(1 + \epsilon)n \log n$ bits to represent R and compute R^{-1} in $O(1/\epsilon)$ worst-case time. Given a position i in *rpar* corresponding to a *RevTrie* node, the corresponding R value is $R[rank_1(B, rank_1(rpar, i))]$.

- $skips_{0...n'}$: The PATRICIA tree skips of the nodes in preorder, using $\log\log u$ bits per node and inserting empty unary nodes when the skip exceeds $\log u$. In this way, one out of $\log u$ empty unary nodes could be explicitly represented. In the worst case there are $O(u)$ empty unary nodes, of which $O(u/\log u)$ are explicitly represented. This adds $O(u/\log u)$ to n', which translates into $O(\frac{u(\log\sigma + \log\log u)}{\log u}) = o(u \log\sigma)$ bits overall.

Fig. 1 (right) shows the resulting navigation scheme. The search algorithm remains the same since we can map preorder positions to nodes in the new rep-

[3] As $ancestor(x, y) \equiv rank_1(par, x) \leqslant rank_1(par, y) \leqslant rank_1(par, x) + subtreesize(par, x) - 1$.

[4] This data structure ensures that one finds the inverse after following the permutation $O(1/\epsilon)$ times.

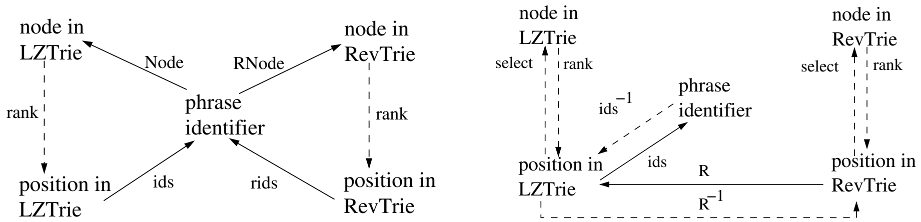


Fig. 1. The original (left) and the reduced (right) navigation structures over index components

resentation of the tries (and vice versa), and we can simulate $rids[i] = ids[R[i]]$, $RNode[i] = select_1(rpar, R^{-1}(ids^{-1}(i))) + 1$, and $Node[i] = select_1(par, ids^{-1}$ $(i))$ $+1$, all of which take constant time.

The space requirement is $(2+\epsilon)n\log n+3n\log\sigma+2n\log\log u+8n+o(u\log\sigma) = (2+\epsilon)n\log n+o(u\log\sigma)$ bits if $\log\sigma = o(\log u)$. As $n\log u = uH_k(T)+O(kn\log\sigma)$ for any k [8], the space requirement is $(2+\epsilon)uH_k(T) + o(u\log\sigma)$ bits, for any $k = o(\log_\sigma u)$. The *child* operation on *RevTrie* can now be computed in $O(1)$ time, versus the $O(h\log\sigma)$ time of the original LZ-index [14]. Hence, the *occ* occurrences of P can be reported in $O(\frac{m^2}{\epsilon} + \frac{n}{\epsilon\sigma^{m/2}})$ average time, for $0 < \epsilon < 1$.

Reducing Further. To simplify notation, given a *LZTrie* node with preorder position $R[i]$, suppose that operation $parent(R[i])$ gives the preorder position of its parent.

Definition 1. *We define function* φ *as* $\varphi(i) = R^{-1}(parent(R[i]))$.

That is, let ax $(a \in \Sigma)$ be the i-th string in *RevTrie*. Then, $\varphi(i) = j$, where the j-th string in *RevTrie* is x. Thus φ is a *suffix link* function in *RevTrie*. As x^Ra must be a *LZTrie* phrase, by the LZ78 parsing it follows that x^R is also a *LZTrie* phrase and thus x is a *RevTrie* phrase. Hence, every non-empty node in *RevTrie* has a suffix link.

Let us show how to compute R using only φ. We define array $L_{1\ldots n}$ such that $L[i] = lets(R[i])$. As $L[i]$ is the first character of the i-th string in *RevTrie*, we have that $L[i] \leqslant L[j]$ whenever $i \leqslant j$, and L can be divided into σ runs of equal symbols. Thus L can be represented by an array L' of $\sigma\log\sigma$ bits and a bit vector L_B of $n + o(n)$ bits, such that $L_B[i] = 1$ iff $L[i] \neq L[i-1]$, for $i = 2\ldots n$, and $L_B[1] = 0$ (this position belongs to the text terminator "\$"). For every i such that $L_B[i] = 1$, we store $L'[rank_1(L_B, i)] = L[i]$. Hence, $L[i]$ can be computed as $L'[rank_1(L_B, i)]$ in $O(1)$ time. To simplify the notation assume that, given a *LZTrie* position $R[i]$, operation $child(R[i], \alpha)$ yields the *LZTrie* preorder position belonging to the child (by symbol α) of the node corresponding to $R[i]$.

Lemma 1. *Given* $0 \leqslant i \leqslant n$*, the value* $R[i]$ *can be computed by the following recurrence:*

$$R[i] = \begin{cases} child(R[\varphi(i)], L[i]) & \textit{if } i \neq 0 \\ 0 & \textit{if } i = 0 \end{cases}$$

Proof. $R[0] = 0$ holds from the fact that the preorder position corresponding to the empty string, both in *LZTrie* and *RevTrie*, is 0. To prove the other part we note that if x is the parent in *LZTrie* of node y with preorder position $R[i]$, then the symbol labeling the arc connecting x to y is $L[i]$. That is, $child(parent(R[i]), L[i]) = R[i]$. The lemma follows from this fact and replacing $\varphi(i)$ by its definition (Def. 1) in the recurrence. □

As in the case of function Ψ of *Compressed Suffix Arrays* [16], we can prove the following lemma for function φ, which is the key to compress R.

Lemma 2. *For every* $i < j$, *if* $lets(R[i]) = lets(R[j])$, *then* $\varphi(i) < \varphi(j)$.

Proof. Let $str_r(i)$ denote the i-th string in the lexicographically sorted set of reversed strings. Note that $str_r(i) < str_r(j)$ iff $i < j$. If $i < j$ and $lets(R[i]) = lets(R[j])$ (i.e., $str_r(i)$ and $str_r(j)$ start with the same symbol, as their reverses end with the same symbol), then $str_r(\varphi(i)) < str_r(\varphi(j))$ (as $str_r(\varphi(i))$ is $str_r(i)$ without its first symbol), and thus $\varphi(i) < \varphi(j)$. □

Corollary 1. φ *can be partitioned into at most* σ *strictly increasing sequences.*

As a result, we replace R by φ, L_B and L', and use them to compute a given value $R[i]$. According to Lemma 1, we can represent φ using the idea of Sadakane [16], requiring $nH_0(lets) + O(n \log\log\sigma)$ bits and allowing to access $\varphi(i)$ in constant time, and hence we replace the $n \log n$-bit representation of R by the $nH_0(lets) + O(n \log\log\sigma) + n + O(\sigma\log\sigma) + o(n)$ bits representation of φ, L_B and L'.

The time to compute $R[i]$ is now $O(|str_r(i)|)$, which actually corresponds to traversing *LZTrie* from the root with the symbols of $str_r(i)$ in reverse order. But we can store ϵn values of R in an array R', plus a bit vector R_B of $n + o(n)$ bits indicating which values of R have been stored, ensuring that $R[i]$ can be computed in $O(1/\epsilon)$ time and requiring $\epsilon n \log n$ extra bits. To determine the R values to be explicitly stored, for each *LZTrie* leaf we traverse the upward path to the root, marking one out of $O(1/\epsilon)$ nodes, and stopping the procedure for the current leaf when we reach the root or when we reach an already marked node. If the node to mark is at preorder position j, then we set $R_B[R^{-1}(j)] = 1$. After we mark the positions of R to be stored, we scan R_B sequentially from left to right, and for every i such that $R_B[i] = 1$, we set $R'[rank_1(R_B, i)] = R[i]$. Then, we free R since $R[i]$ can be computed by:

$$R[i] = \begin{cases} child(R[\varphi(i)], L'[rank_1(L_B, i)]) & \text{if } R_B[i] = 0 \\ R'[rank_1(R_B, i)] & \text{if } R_B[i] = 1 \end{cases}$$

Note that the same structure used to compute R^{-1} before freeing R can be used under this scheme, with cost $O(1/\epsilon^2)$ (recall footnote 6).

Theorem 1. *There exists a compressed full-text self-index requiring* $(1 + \epsilon)$ $uH_k(T) + o(u \log\sigma)$ *bits of space, for* $\sigma = O(\text{polylog}(u))$, *any* $k = o(\log_\sigma u)$ *and any constant* $0 < \epsilon < 1$, *and able to report the occ occurrences of a pattern* $P_{1...m}$ *in a text* $T_{1...u}$ *in* $O(\frac{m^2}{\epsilon^2} + \frac{n}{\epsilon^2\sigma^{m/2}})$ *average time, which is* $O(m^2)$ *if* $m \geqslant 2\log_\sigma n$. *It can also display a text substring of length* ℓ *in* $O(\ell(1 + \frac{1}{\epsilon\log_\sigma \ell}))$ *worst-case time.*

The bound $O(\ell(1+\frac{1}{\epsilon\log_\sigma \ell})$ in the displaying time holds from the fact that we perform ℓ *parent* operations, and we must pay $O(1/\epsilon)$ to use ids^{-1} each time we pass to display the next (previous) phrase, which in the (very) worst case is done $O(\ell/\log_\sigma \ell)$ times. We still assume that these ℓ symbols form whole phrases.

We can get worst case guarantees in the search process by adding *Range*, the two-dimensional range search data structure defined in Section 2. Occurrences of type 2 can now be solved in $O(m^2 + (m + occ)\log n)$ time.

Theorem 2. *There exists a compressed full-text self-index requiring* $(2+\epsilon)$ $uH_k(T) + o(u\log\sigma)$ *bits of space, for* $\sigma = O(\text{polylog}(u))$*, any* $k = o(\log_\sigma u)$ *and any constant* $0 < \epsilon < 1$*, and able to report the occ occurrences of a pattern* $P_{1\ldots m}$ *in a text* $T_{1\ldots u}$ *in* $O(m^2(\log m + \frac{1}{\epsilon^2}) + (m + occ)\log n + \frac{occ}{\epsilon}) = O(m^2\log m + (m + occ)\log n)$ *worst-case time. It can also display a text substring of length* ℓ *in* $O(\ell(1+\frac{1}{\epsilon\log_\sigma \ell}))$ *worst-case time.*

4 Using the *xbw* Transform to Represent LZTrie

A different idea to reduce the space requirement of LZ-index is to use the *xbw transform* of Ferragina et al. [3] to represent the *LZTrie*. This succinct representation supports the operations $parent(x)$, $child(x,i)$, and $child(x,\alpha)$, all of them in $O(1)$ time and using $2n\log\sigma + O(n)$ bits of space. The representation also allows *subpath queries*, a very powerful operation which, given a string s, returns all the nodes x such that s is a suffix of the string represented by x. We represent LZ-index with the following data structures:

Balanced parentheses *LZTrie*: The trie of the Lempel-Ziv phrases, storing

- *par* : The balanced parentheses representation [13] of *LZTrie*. In order to index the *LZTrie* leaves with *xbw*, we have to add a dummy child to each. In this way, the trie has $n' \leqslant 2n$ nodes. The space requirement is $4n + o(n)$ bits in the worst case if we use the Munro and Raman representation [13]. We use the bit 0 to represent '`(`' and 1 to represent '`)`'. In this way, the preorder position of a node is computed by a $rank_0$ query, and the node corresponding to a preorder position by a $select_0$ query, both in $O(1)$ time.

- *ids* : The array of LZ78 phrase identifiers in preorder, represented by the data structure of Munro et al. [12], such that we can compute the inverse permutation ids^{-1} in $O(1/\epsilon)$ time, requiring $(1+\epsilon)n\log n$ bits.

xbw LZTrie: The *xbw* representation [3] of the *LZTrie*, where the nodes are lexicographically sorted according to their upward paths in the trie. We store

- S_α : The array of symbols labeling the arcs of the trie. In the worst case *LZTrie* has $2n$ nodes (because of the dummy leaves we add), and then this array requires $2n\log\sigma$ bits.

- S_{last} : A bit array such that $S_{last}[i] = 1$ iff the corresponding node in *LZTrie* is the last child of its parent. The space requirement is $2n(1 + o(1))$ bits.

Pos: A mapping from *xbw* positions to the corresponding preorder positions. In the worst case there are $2n$ such positions, and so the space requirement is

$2n \log (2n)$ bits. We can reduce this space to $\epsilon n \log (2n)$ bits by storing in an array Pos' one out of $O(1/\epsilon)$ values of Pos, such that $Pos[i]$ can be computed in $O(1/\epsilon)$ time. We need a bit vector Pos_B of $2n(1+o(1))$ bits indicating which values of Pos have been stored. Assume we need compute $Pos[i]$, for a given *xbw* position i. If $Pos_B[i] = 1$, then such value is stored at $Pos'[rank_1(Pos_B, i)]$. Otherwise, we simulate a preorder traversal in *xbw* from node at *xbw* position i, until $Pos_B[j] = 1$, for a *xbw* position j. Once this j is found, we map to the preorder position $j' = Pos'[rank_1(Pos_B, j)]$. If d is the number of steps in preorder traversal from *xbw* position i to *xbw* position j, then $j' - d$ is the preorder position corresponding to the node at *xbw* position i. We also need to compute Pos^{-1}, which can be done in $O(1/\epsilon^2)$ time under this scheme, requiring $\epsilon n \log (2n)$ extra bits if we use the representation of [12].

Range: A *range search* data structure in which we store the point k (belonging to phrase identifier k) at coordinate (x, y), where x is the *xbw position of phrase* k and y is the *preorder position of phrase* $k + 1$. We use the data structure of Chazelle [2] requiring $n \log n(1 + o(1))$ bits, as for the original LZ-index.

The total space requirement is $(2+\epsilon)n \log n(1+o(1))+2n \log \sigma+(8+\epsilon)n+o(n)$ bits, which is $(2+\epsilon)uH_k(T)+o(u \log \sigma)$ bits if $\log \sigma = o(\log u)$ and $k = o(\log_\sigma u)$.

We depict now the search algorithm for pattern P. For occurrences of type 1, we perform a subpath query for P to obtain the interval $[x_1, x_2]$ in the *xbw* of *LZTrie* corresponding to all the nodes whose phrase ends with P. For each position $i \in [x_1, x_2]$, we can get the corresponding node in the parentheses representation using $select_0(par, Pos[i])$, and then we traverse the subtrees of these nodes and report all the identifiers found, as done with the usual LZ-index.

To solve occurrences of type 2, for every possible partition $P_{1 \ldots i}$ and $P_{i+1 \ldots m}$ of P, we traverse the *xbw* from the root, using operation $child(x, \alpha)$ with the symbols of $P_{i+1 \ldots m}$. Once this is found, say at *xbw* position i, we switch to the preorder tree (parentheses) using $select_0(par, Pos[i])$, to get the node v_{lz} whose subtree has the preorder interval $[y_1, y_2]$ of all the nodes that start with $P_{i+1 \ldots m}$. Next we perform a subpath query for $P_{1 \ldots i}$ in *xbw*, and get the *xbw* interval $[x_1, x_2]$ of all the nodes that finish with $P_{1 \ldots i}$ (we have to replace $x_r \leftarrow rank_1(S_{last}, x_r)$ to avoid counting the same node multiple times, see [3]). Then, we search structure *Range* for $[x_1, x_2] \times [y_1, y_2]$ to get all phrase identifiers k such that phrase k finishes with $P_{1 \ldots i}$ and phrase $k + 1$ starts with $P_{i+1 \ldots m}$.

For occurrences of type 3, one could do mostly as with the original *LZTrie* (navigating the *xbw* instead), so as to find all the nodes equal to substrings of P in $O(m^2)$ time. Then, for each maximal concatenation of phrases $P_{i \ldots j} = B_{k+1} \ldots B_{\ell-1}$ we must check that phrase B_ℓ starts with $P_{j+1 \ldots m}$ and that phrase B_k finishes with $P_{1 \ldots i-1}$. The first check can be done in constant time using ids^{-1}. As we have searched for all substrings of P in the trie, we know the preorder interval of descendants of $P_{j+1 \ldots m}$, thus we check whether the node at preorder position $ids^{-1}(\ell)$ belongs to that interval. The second check can also be done in constant time, by determining whether k is in the *xbw* interval of $P_{1 \ldots i-1}$ (that is, B_k finishes with $P_{1 \ldots i-1}$). The *xbw* position is $Pos^{-1}(ids^{-1}(k))$.

To display the text around an occurrence, we use ids^{-1} to find the preorder position of the corresponding phrase, and then we use *parent* on the parentheses to find the symbols in the upward path. To know the symbol, we have to use Pos^{-1} to go to the *xbw* position and read S_α.

For the search time, occurrences of type 1 cost $O(m + occ/\epsilon)$, type 2 cost $O(m^2 + m/\epsilon + m(occ + \log n))$, and type 3 cost $O(m^2(\log m + \frac{1}{\epsilon^2}))$. Thus, we have achieved Theorem 2 again with radically different means. The displaying time is $O(\ell/\epsilon^2)$, but it can also become $O(\ell(1 + \frac{1}{\epsilon \log_\sigma \ell}))$ if we store the array of symbols in the balanced parentheses *LZTrie*, which adds $o(u \log \sigma)$ bits of space. We can get a version requiring $(1+\epsilon)uH_k(T)+o(u \log \sigma)$ bits and $O(m^2)$ average reporting time if $m \geqslant 2 \log_\sigma n$ (as in Theorem 1) if we solve occurrences of type 2 similarly as we handled occurrences of type 3, and dropping *Range*.

5 Displaying Text Substrings

LZ-index is able to report occurrences in the format (k, offset), where k is the phrase in which the occurrence starts and *offset* is the distance between the beginning of the occurrence and the end of the phrase. However, we can report occurrences as *text positions* by adding a bit vector $V_{1\dots u}$ that marks the n phrase beginnings. Then $rank_1(V, i)$ is the phrase number i belongs to, and $select_1(V, j)$ is the text position of the j-th phrase. Such V can be represented with $H_0(V) + o(u) \leqslant n \log(u/n) + o(u) \leqslant n \log \log u + o(u) = o(u \log \sigma)$ bits [15]. We can also add, to both proposed indexes, an operation for displaying a subtext $T_{i\dots i+\ell-1}$ for any given position i, in optimal $O(\ell/\log_\sigma u)$ time.

A compressed data structure [17] to display any text substring of length $\Theta(\log_\sigma u)$ in constant time, turns out to have similarities with LZ-index. We take advantage of this similarity to plug it within our index, with some modifications, and obtain improved time to display text substrings. They proposed auxiliary data structures of $o(u \log \sigma)$ bits to *LZTrie* to support this operation efficiently. Given a position i of the text, we first find the phrase including the position i by using $rank_1(V, i)$, then find the node of *LZTrie* that corresponds to the phrase using ids^{-1}. Then displaying a phrase is equivalent to outputting the path going from the node to the root of *LZTrie*. The auxiliary data structure, of size $O(n \log \sigma) = o(u \log \sigma)$ bits, permits outputting the path by chunks of $\Theta(\log_\sigma u)$ symbols in $O(1)$ time per chunk. In addition, we can now display not only whole phrases, but any text substring within this complexity. The reason is that any prefix of a phrase is also a phrase, and it can be found in constant time by using a level-ancestor query [6] on the *LZTrie*.

We modify this method to plug into our indexes. In their original method [17], if more than one consecutive phrases have length less than $(\log_\sigma u)/2$ each, their phrase identifiers are not stored. Instead the substring of the text including those phrases are stored without compression. This guarantees efficient displaying operation without increasing the space requirement. However this will cause the problem that we cannot find patterns including those phrases. Therefore in our modification we store both the phrases themselves and their phrase identifiers. The search algorithm remains as before. To decode short phrases we can just output the explicitly stored substring including the phrases. For each phrase with length at most $(\log_\sigma u)/2$, we store a substring of length $\log u$ containing the phrase. Because there are at most $O(\sqrt{u})$ such phrases, we can store the substrings in $O(\sqrt{u}\log u) = o(u)$ bits. These auxiliary structures work as long as we can convert a phrase identifier into a preorder position in *LZtrie* in constant time. Hence they can be applied to all the data structures in Sections 3 and 4.

Theorem 3. *The indexes of Theorem 1 and Theorem 2 (and those of Section 4) can be adapted to display a text substring of length ℓ surrounding any text position in optimal $O(\frac{\ell}{\log_\sigma u})$ worst case-time.*

References

1. D. Benoit, E. Demaine, I. Munro, R. Raman, V. Raman, and S.S. Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005.
2. B. Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM J. on Computing*, 17(3):427–462, 1988.
3. P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Structuring labeled trees for optimal succinctness, and beyond. *Proc. FOCS*, pp. 184–196, 2005.
4. P. Ferragina and G. Manzini. Indexing compressed texts. *J. of the ACM* 54(4):552–581, 2005.
5. P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. An alphabet-friendly FM-index. *Proc. SPIRE*, LNCS 3246, pp. 150–160, 2004. Extended version to appear in *ACM TALG*.
6. R. Geary, R. Raman, and V. Raman. Succinct ordinal trees with level-ancestor queries. *Proc. SODA*, pp. 1–10, 2004.
7. R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. *Proc. SODA*, pp. 841–850, 2003.
8. R. Kosaraju and G. Manzini. Compression of low entropy strings with Lempel-Ziv algorithms. *SIAM J. on Computing*, 29(3):893–911, 1999.
9. G. Manzini. An analysis of the Burrows-Wheeler transform. *J. of the ACM* 48(3):407–430, 2001.
10. D. R. Morrison. Patricia – practical algorithm to retrieve information coded in alphanumeric. *J. of the ACM* 15(4):514–534, 1968.
11. I. Munro. Tables. *Proc. FSTTCS*, LNCS 1180, pp. 37–42, 1996.
12. I. Munro, R. Raman, V. Raman, and S.S. Rao. Succinct representations of permutations. *Proc. ICALP*, LNCS 2719, pp. 345–356, 2003.
13. J. I. Munro and V. Raman. Succinct Representation of Balanced Parentheses and Static Trees. *SIAM J. on Computing*, 31(3):762–776, 2001.
14. G. Navarro. Indexing text using the Ziv-Lempel trie. *Journal of Discrete Algorithms (JDA)*, 2(1):87–114, 2004. See also TR/DCC-2003-0, Dept. of CS, U. Chile. `ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/jlzindex.ps.gz`.
15. R. Raman, V. Raman, and S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. *Proc. SODA*, pp. 233–242, 2002.
16. K. Sadakane. New Text Indexing Functionalities of the Compressed Suffix Arrays. *J. of Algorithms*, 48(2):294–313, 2003.
17. K. Sadakane and R. Grossi. Squeezing Succinct Data Structures into Entropy Bounds. *Proc. SODA*, pp. 1230–1239, 2006.
18. J. Ziv and A. Lempel. Compression of individual sequences via variable–rate coding. *IEEE Trans. Information Theory*, 24(5):530–536, 1978.

Faster Algorithms for Computing Longest Common Increasing Subsequences

Gerth Stølting Brodal[1,*], Kanela Kaligosi[2],
Irit Katriel[1,**], and Martin Kutz[2]

[1] BRICS[***], University of Aarhus, Århus, Denmark
{gerth, irit}@daimi.au.dk
[2] Max-Plank-Institut für Informatik, Saarbrücken, Germany
{kaligosi, mkutz}@mpi-inf.mpg.de

Abstract. We present algorithms for finding a longest common increasing subsequence of two or more input sequences. For two sequences of lengths m and n, where $m \geq n$, we present an algorithm with an output-dependent expected running time of $O((m + n\ell) \log\log \sigma + \mathit{Sort})$ and $O(m)$ space, where ℓ is the length of an LCIS, σ is the size of the alphabet, and *Sort* is the time to sort each input sequence. For $k \geq 3$ length-n sequences we present an algorithm which improves the previous best bound by more than a factor k for many inputs. In both cases, our algorithms are conceptually quite simple but rely on existing sophisticated data structures. Finally, we introduce the problem of longest common weakly-increasing (or non-decreasing) subsequences (LCWIS), for which we present an $O(m + n \log n)$-time algorithm for the 3-letter alphabet case. For the extensively studied longest common subsequence problem, comparable speedups have not been achieved for small alphabets.

1 Introduction

Algorithms that search for the longest common subsequence (LCS) of two input sequences or the longest increasing subsequence (LIS) of one input sequence date back several decades.

Formally, given two sequences $A = (a_1, \ldots, a_n)$ and $B = (b_1, \ldots, b_m)$ with elements from an alphabet Σ and with $m \geq n$, a *common subsequence* of A and B is a subsequence $(a_{j_1} = b_{\kappa_1}, a_{j_2} = b_{\kappa_2}, \ldots a_{j_\ell} = b_{\kappa_\ell})$, where $j_1 < j_2 < \cdots < j_\ell$ and $\kappa_1 < \kappa_2 < \cdots < \kappa_\ell$. Given one sequence $A = (a_1, \ldots, a_n)$ where the a_i's are drawn from a totally ordered set, an *increasing subsequence* of A is a subsequence $(a_{j_1}, a_{j_2}, \ldots, a_{j_\ell})$ such that $j_1 < j_2 < \cdots < j_\ell$ and $a_{j_1} < a_{j_2} < \cdots < a_{j_\ell}$.

[*] Supported by the Danish Natural Science Research Council(grant # 21-04-0389).
[**] Supported by the Danish Research Agency(grant # 272-05-0081).
[***] Basic Research in Computer Science, funded by the Danish National Research Foundation.

M. Lewenstein and G. Valiente (Eds.): CPM 2006, LNCS 4009, pp. 330–341, 2006.

A classic algorithm by Wagner and Fischer [12] solves the LCS problem using dynamic programming in $O(mn)$ time and space. Hirschberg [7] reduced the space complexity to $O(n)$, using a divide-and-conquer approach. The fastest known algorithm by Masek and Paterson [9] runs in $O(n^2/\log n)$ time. Faster algorithms are known for special cases, such as when the input consists of permutations or when the output is known to be very long or very short. Hunt and Szymanski [8] studied the complexity of the LCS problem in terms of matching index pairs, i.e., they defined r to be the number of index-pairs (i, j) with $a_i = b_j$ (such a pair is called a *match*) and designed an algorithm that finds the LCS of two sequences in $O(r \log n)$ time. For a survey on the LCS problem see [2].

Fredman [5] showed how to compute an LIS of a length-n sequence in optimal $O(n \log n)$ time. When the input sequence is a permutation of $\{1, \ldots, n\}$, Hunt and Szymanski [8] designed an $O(n \log \log n)$-time solution, which was later simplified by Bespamyatnikh and Segal [3]. The expected length of a longest increasing subsequence of a random permutation has been shown (after successive improvements) to be $2\sqrt{n} - o(\sqrt{n})$; for a survey see [1].

Note that after sorting both input sequences we can in linear time remove symbols that do not appear in both sequences and rename the remaining symbols to the alphabet $\{1, 2, \ldots, \sigma\}$. We can therefore assume that this preprocessing stage was performed and hence the size of the alphabet, σ, is at most n. In the following we let $Sort_\Sigma(m)$ denote the time required to sort a length-m input sequence drawn from the alphabet Σ.

Recently, Yang et al. [14] combined the two concepts and defined a *common increasing subsequence* (CIS) of two sequences A and B, i.e., an increasing sequence that is a subsequence of both A and B. They designed a dynamic programming algorithm that finds a *longest CIS* (an LCIS, for short) of A and B using $\Theta(mn)$ time and space.

Subsequently, Chan et al. [4] obtained an upper bound of $O(\min\{r \log \sigma, m\sigma + r\} \log \log m + Sort_\Sigma(m))$. The number of matches r is in the worst case $\Omega(mn)$, but in some important cases it is much smaller. For instance, when A and B are permutations of $\{1, \ldots, n\}$ then $r = O(n)$. They then proceeded to generalize their algorithm to find an LCIS of $k \geq 3$ length-n sequences. They show that this can be done in $O(\min\{kr^2, kr \log \sigma \log^{k-1} r\} + kSort_\Sigma(n))$ time, where r is again the number of matches, i.e., k-coordinate vectors that contain an index from each input sequence, all with the same symbol.

1.1 Our Results

In this paper we present three new upper bounds for the LCIS problem. The first is an output-dependent algorithm which runs in $O((m+n\ell) \log \log \sigma + Sort_\Sigma(m))$ expected time and $O(m)$ worst-case space, where ℓ is the length of an LCIS. Whenever $n = \Omega(\log \log \sigma + Sort_\Sigma(m)/m)$ and either $m = \Omega(n \log \log \sigma)$ or $\ell = o(n/\log \log n)$, it is faster than Yang et al.'s $\Theta(mn)$-time algorithm.

Table 1. Parameters of the LCIS/LCWIS problems

Symbol	Meaning
m, n	Lengths of input sequences (we assume $m \geq n$).
ℓ	Length of the LCIS/LCWIS.
k	Number of input sequences.
σ	Size of the alphabet (number of different symbols).
r	Number of matches in the input sequences.

Table 2. Previous and new results. The new upper bounds apply to both LCIS and LCWIS.

	Previous Results	New
$k = 2$	$O(mn)$ [14] $O(\min\{r \log \sigma, m\sigma + r\} \log \log m$ $+ Sort_{\Sigma}(m))$ [4]	$O((m + n\ell) \log \log \sigma + Sort_{\Sigma}(m))$ $O(m)$ when $\sigma = 2$ $O(m + n \log n)$ when $\sigma = 3$
$k \geq 3$	$O(\min\{kr^2, kr \log \sigma \log^{k\ \ 1} r\}$ $+ kSort_{\Sigma}(n))$ [4]	$O(\min\{kr^2, r \log^{k\ \ 1} r \log \log r\}$ $+ kSort_{\Sigma}(n))$

For a strictly-increasing subsequence we have $\ell \leq \sigma$. However, in the weakly-increasing (i.e. non-decreasing) variant, the length of the output can be arbitrarily larger than the size of the alphabet. We show that a *longest common weakly increasing subsequence* (LCWIS) can be found in linear time for an alphabet of size two and in $O(m + n \log n)$ time for an alphabet of size three. These results are interesting because they pinpoint what seems to be a fundamental difference between the LCS and LWCIS problems. The approach we use cannot be applied to LCS, and to date, comparable speedups have not been achieved for LCS with small alphabets.

Finally, we consider the case of $k \geq 3$ length-n sequences. The upper bound of Chan et al. is achieved by two algorithms; the first is a simple $O(kr^2 + kSort_{\Sigma}(n))$ time algorithm and the second is a more complex implementation of the same approach, which runs in $O(kr \log \sigma \log^{k-1} r + kSort_{\Sigma}(n))$ time. We describe an algorithm which is significantly simpler than the latter and obtain a running time of $O(\min\{kr^2, r \log^{k-1} r \log \log r\} + kSort_{\Sigma}(n))$.

Table 1 provides a list of the symbols used in the paper and Table 2 summarizes the previous and new results.

The rest of the paper is organized as follows. In Section 2 we describe a dynamic programming algorithm that uses a data structure based on van Emde Boas trees and runs in expected $O((m + n\ell) \log \log \sigma + Sort_{\Sigma}(m))$ time and $O(m)$ space. In Section 3 we present our results on LCWIS with small alphabets, which use different techniques. Finally, in Section 4 we describe how to use a data structure by Gabow et al. [6] to obtain an algorithm for finding an LCIS or LCWIS of $k \geq 3$ sequences, which is simpler and faster than Chan et al.'s algorithm.

2 An Output-Dependent Upper Bound

2.1 Bounded Heaps

In our output-dependent algorithm we need to access items that carry two integer parameters: *priorities* and *keys*. The basic query will be for the highest-priority element amongst all those whose keys are below a given threshold. We use a data structure, subsequently called a *bounded heap* (BH), that supports the following operations:

- $\mathit{Insert}(\mathcal{H}, k, p, d)$: Insert into the *BH* $\mathcal{H}$ the key k with priority p and associated data d.
- $\mathit{DecreasePriority}(\mathcal{H}, k, p, d)$: If the *BH* $\mathcal{H}$ does not already contain the key k, perform $\mathit{Insert}(\mathcal{H}, k, p, d)$. Otherwise, set this key's priority to $\min\{p, p'\}$, where p' is its previous priority.
- $\mathit{BoundedMin}(\mathcal{H}, k)$: Return the item that has minimum priority among all items in $\mathcal{H}$ with key smaller than k. If $\mathcal{H}$ does not contain any items with key smaller than k, return "invalid".

The priority search tree (PST) of McCreight [10] supports each of these operations in $O(\log n)$ time. However, the PST also allows deletions, which the BH is not required to support. Using van Emde Boas trees, we obtain a faster BH for integer keys:

Lemma 1. *There exists an implementation of bounded heaps that requires $O(n)$ space and supports each of the above operations in $O(\log \log n)$ amortized time, where keys are drawn for the set $\{1, \ldots, n\}$.*

Proof (sketch). The data structure applies standard techniques, such as those described in Section 3 of [6]. We rely on the fact that a snapshot of the heap, at any point in time, can be represented as a decreasing step function. More precisely, let $BM(s)$ be the value that would be returned by a $\mathit{BoundedMin}(\mathcal{H}, s)$ query. Then $BM(s) \leq BM(s')$ whenever $s > s'$, i.e., the function BM is non-increasing in s (see Figure 1).

key k	1	2	3	4	5	6	7	8	9	10
priority	7	10	6	8	5	3	2	4	1	9
$BM(k)$	∞	7	7	6	6	5	3	2	2	1

Fig. 1. Example of *BM* values

Assume that the keys are $s_1, s_2, \ldots$ with $s_i \leq s_{i+1}$ for all i. To answer *BoundedMin* queries, it suffices to maintain a search structure that contains the $BM(s_i)$ value for every s_i at which the function BM changes, i.e., $BM(s_i) < BM(s_{i-1})$. Then, we answer a $BoundedMin(s)$ by searching the data structure for the largest key which is at most s and returning its BM value. With a van Emde Boas tree [11] as search structure, this takes $O(\log\log n)$ time. The implementation of Insert and DecreasePriority are described in the full version of this paper. □

2.2 An $O((m + n\ell)\log\log\sigma + Sort_\Sigma(m))$ Time Algorithm

Our output-dependent algorithm for the LCIS problem begins with a preprocessing step, where it removes from each sequence all elements that do not appear in the other sequence; this is easy after the sequences are sorted. For every remaining element s, it generates a sorted list Occ_s that contains ∞ and the indices of all occurrences of s in B.

Then, in n iterations the algorithm identifies common increasing subsequences (CISs) of increasing lengths: In iteration i it identifies length-i CISs (using the results of iteration $i-1$). More precisely, for every element a_j in A, it identifies the minimum index κ in B such that there is a length-i CIS which ends at a_j in A and at b_κ in B. The index κ is stored in $L_i[j]$.

To compute the array $L_1[1 \ldots n]$, the algorithm traverses A and for each a_j, sets $L_1[j]$ to be the minimum index in the list Occ_{a_j}, i.e., the earliest occurrence of a_j in B. Note that due to the preprocessing, there exists such an index in B.

For $i > 1$, the ith iteration proceeds as follows. The algorithm traverses A again, and for every a_j, it checks whether a_j (together with some b_κ) can extend a length-$(i-1)$ CIS to a length-i CIS, and if so, identifies the minimum such κ. For this purpose, the algorithm maintains a bounded heap $\mathcal{H}$. When it begins processing a_j, $\mathcal{H}$ contains all elements $a_t \in \{a_1, \ldots, a_{j-1}\}$ for which $L_{i-1}[t] \neq \infty$. The key of a_t in $\mathcal{H}$ is a_t itself and its priority is $L_{i-1}[t]$, i.e., the minimum index of the endpoint in B of a length-$(i-1)$ CIS which ends, in A, at index t. The algorithm queries $\mathcal{H}$ to find the leftmost endpoint (in B) of a length-$(i-1)$ CIS, which contains only elements smaller than a_j. Let κ' be this endpoint. Then, $L_i[j]$ is set to the first occurrence of a_j in B which lies behind κ'; we prove that this is the leftmost endpoint in B of a length-i CIS which ends, in A, at a_j. A formal description of the algorithm is given in the full version of this paper.

We emphasize that $\mathcal{H}$ is built anew for every single pass. The only information saved between different scans of A and B is maintained in the arrays L_i.

The arrays $Link_1, Link_2, \ldots$ are used to save the information we need in order to construct the LCIS: Whenever we detect that the index pair (j, κ) can extend a length-$(i-1)$ CIS which ends at the index pair (j', κ'), we set $Link_i[j] = j'$. Finally, if there is a length-$(i-1)$ CIS which ends at a_j, then a_j is inserted into $\mathcal{H}$ with priority $L_{i-1}[a_j]$; it may later be extended into a length-i CIS by some $a_{j'}$ with $j' > j$.

Correctness. The correctness of the algorithm relies on the following lemma, which states that if there is a solution then the algorithm finds it. It is straightforward to show that the algorithm will not produce an invalid sequence.

Lemma 2. *Let A and B be two sequences that have a length-ℓ CIS which ends in A at index j and in B at index κ. Then at the end of the iteration in which $i = \ell$, $L_\ell[j] \leq \kappa$.*

Proof. By induction on ℓ. For $\ell = 1$, the claim is obvious. Assume that it holds for any length-$(\ell - 1)$ CIS and that we are given A and B which have a length-ℓ CIS $c_1, \ldots, c_\ell$ that is located in A as $a_{j_1}, \ldots, a_{j_\ell}$ and in B as $b_{\kappa_1}, \ldots, b_{\kappa_\ell}$.

By the induction hypothesis, at the end of the $i = \ell - 1$ iteration, L_{i-1} contains entries that are not equal to ∞. Hence, the algorithm will proceed to perform iteration $i = \ell$. Again by the induction hypothesis, $L_{\ell-1}[j_{\ell-1}] \leq \kappa_{\ell-1}$.

Since $a_{j_{\ell-1}} < a_{j_\ell}$, it is guaranteed that when $j = j_\ell$, $\mathcal{H}$ contains an item with key $a_{j_{\ell-1}}$, priority $\kappa' \leq \kappa_{\ell-1}$, and $d = (j_{\ell-1}, \kappa')$. So the *BoundedMin* operation will return a valid value. If the value returned is $(j_{\ell-1}, \kappa_{\ell-1})$, then the smallest occurrence of a_ℓ in B after $\kappa_{\ell-1}$ is not beyond κ_ℓ. So the algorithm will set $L_\ell[j_\ell] \leq \kappa_\ell$. On the other hand, if the value returned is not $(j_{\ell-1}, \kappa_{\ell-1})$, then it is $(j_{\ell-1}, \kappa')$ for some $\kappa' \leq \kappa_{\ell-1}$. Since $a_{j'} < a_\ell$, again we get that the smallest occurrence of a_ℓ in B after $\kappa_{\ell-1}$ is not beyond κ_ℓ. So the algorithm will set $L_\ell[j_\ell] \leq \kappa_\ell$. □

Time complexity. The preprocessing phase takes $O(Sort_\Sigma(m))$ time, to sort each of the sequences A and B. The construction of the Occ_s's takes $O(m)$ time.

The array A is traversed $O(\ell)$ times. During each traversal, $O(n)$ operations are performed on the bounded heap, each of which takes $O(\log\log\sigma)$ amortized time, and the Occ_s lists are queried at most n times. We now sketch a possible implementation of the Occ_s lists.

We partition the range $\{1, \ldots, m\}$ into m/σ blocks of σ consecutive locations and for every $1 \leq i \leq m/\sigma$ we denote by b_i the block containing locations $(i-1)\sigma + 1, \ldots, i\sigma$. For each i and each $s \in \Sigma$ we create a data structure that represents occurrences of s in the block b_i and is based on Willard's y-fast tries [13]. In addition, for each block we store the first occurrence of s succeeding the block. To answer a query in Occ_s, we first identify the block containing the query point in constant time. We then search for the smallest index larger than the query point in the y-fast trie for this block in time $O(\log\log\sigma)$. If we found one, we are done. Otherwise, we return the first s succeeding the block, using the stored information. Initializing the m y-fast tries with a total of m elements takes $O(m \log\log\sigma)$ expected time. Note that this initialization step needs to be carried out only once.

In total, the main loop takes $O(m + n\ell \log\log\sigma)$ time. Finally, Constructing the LCIS takes $O(\ell)$ time. We get that the total expected running time of the algorithm is $O((m + n\ell)\log\log\sigma + Sort_\Sigma(m))$.

Space complexity. As for space complexity, note that in the main loop we only use L_{i-1} and L_i. Therefore, we do not need to save the previous L's. In order to construct the LCIS, the algorithm as described requires $O(n\ell)$ space for the *Link* arrays.

However, we can reduce the space complexity to $O(m)$ with the technique developed by Hirschberg [7] for LCS. First, we run the algorithm once to compute ℓ (without constructing the *Link* arrays). Then we run a recursive version of the algorithm that construct the LCIS. The top recursive level invokes the usual algorithm, except that this time we remember only some of the *Link* information: Each match in the second half of a CIS knows the location in A and B of the $\lfloor \ell/2 \rfloor$-th match of the CIS that it was appended to. This information is found in the $\lfloor \ell/2 \rfloor$-th iteration of the main loop and propagated by the later iterations while the L arrays are constructed. Then, we know for every LCIS the location (i, j) in A and B of the middle match. We select one LCIS and recursively run the same algorithm to find the length-$\lfloor \ell/2 \rfloor - 1$ LCIS of $(a_1, \ldots, a_{i-1})$ $(b_1, \ldots, b_{j-1})$ and the length-$\lceil \ell/2 \rceil$ LCIS of $(a_{i+1}, \ldots, a_n)$ and $(b_{j+1}, \ldots, b_m)$. The base case is when we look for a constant-size LCIS. Then we run the original algorithm in linear space. To achieve that the time complexity remains unchanged we need to limit the work done processing B during the recursion. For the preprocessing for the outermost recursion we need time $Sort_\Sigma(m)$. For the remaining recursive calls we do not need to sort the arrays again and the preprocessing time is $O(m)$. The computation of a middle match considers at most matches involving $n\ell$ entries from B. These entries in B can be marked during the computation of the middle match, and only this subsequence of B is provided to the recursive calls. The thinning of B is done before each recursive call. Let $T(m, n, \ell)$ be the running time of the recursion on two sequences of lengths n and m with a length-ℓ LCIS and $m \leq n\ell$. Assume that the middle match is (n_1, m_1). Then $T(m, n, \ell) \leq n\ell \log\log\sigma + n\ell + T(m_1, n_1, \ell/2) + T(m_2, n_2, \ell/2)$, where $n_1 + n_2 + 1 = n$ and $m_1 + m_2 + 1 \leq m$. This recurrence solves to $O(n\ell \log\log\sigma)$. The total running time becomes $O((m + n\ell)\log\log\sigma + Sort_\Sigma(m))$. It is easy to see that the amount of space we need is $O(m)$.

In conclusion, we have shown:

Theorem 1. *An LCIS of two sequences of lengths m and n with $m \geq n$ can be found in $O((m + n\ell)\log\log\sigma + Sort_\Sigma(m))$ expected time and $O(m)$ worst-case space where ℓ is the length of the output and $Sort_\Sigma(m)$ is the time required to sort a length-m input sequence.*

3 Weakly Increasing Subsequences

We now turn to longest common non-decreasing or *weakly increasing* subsequences (LCWIS) for small alphabets. By simply replacing $<$ by $\leq$ in the *BoundedMin* operation in our algorithm for the LCIS problem, it is straightforward to verify that the algorithm solves the LCWIS problem in $O((m + n\ell)\log\log\sigma + Sort(m))$ time. But while the LCIS problem can be solved in linear time for alphabets of bounded size t, simply because the length of the solution is then also bounded by t, it is not clear how this fact should carry over to LCWIS, where the output size need not relate to t at all.

We show how to solve LCWIS for the 2– and the 3-letter alphabet in linear respectively $O(m + n \log n)$ time. This is in contrast to the classic LCS problem, where already the 2-letter case seems to be essentially as hard as the general problem. In fact, it seems that LCWIS behaves very different from both LCIS and LCS.

3.1 Preprocessing

Let us use as our alphabet the Greek letters $\Sigma = \{\alpha, \beta, \gamma\}$ in their standard order: $\alpha < \beta < \gamma$. For both tasks, the 2-letter and 3-letter cases, we prepare arrays $\mathtt{Num}_{A,\alpha}$,$\mathtt{Num}_{B,\alpha}$,$\mathtt{Num}_{A,\beta}$, ...,$\mathtt{Num}_{B,\gamma}$ that count the number of αs, βs and γs, respectively, in prefixes of A and B. For example, $\mathtt{Num}_{A,\gamma}[9]$ contains the number of γs in A up to position 9 (inclusively). We also create arrays $\mathtt{Pos}_{A,\alpha}$ through $\mathtt{Pos}_{B,\gamma}$, which provide us with the position of the ith occurrence of α, β, or γ in A or B. These arrays can clearly be prepared in $O(m)$ time.

3.2 The 2-Letter Case Is Simple

After the preprocessing, the 2-letter case becomes trivial. For each i, where $0 \leq i \leq \min\{\mathtt{Num}_{A,\alpha}[n], \mathtt{Num}_{B,\alpha}[m]\}$, we determine the position of the ith α in A and B and then the number of βs that come after those positions in the two sequences. This gives us, for every i, the length of an LCWIS of type $\alpha^i\beta^*$. The longest of them over all i are the LCWISs of the two sequences. The total time is $O(m)$.

3.3 The Three-Letter Case—Split Diagrams

The naïve extension of the above approach to three letters would have to deal with a quadratic number of tentative exponent pairs (i, j) for subsequences of type $\alpha^i\beta^j\gamma^*$. We somehow need to avoid the testing of all such pairs. The basis of our near-linear-time algorithm for a 3-letter alphabet are what we like to call "split-diagrams," a data structure that stores information about parts of the given sequences in a compact way.

Assume we were only interested in subsequences of A that have all their αs up to some fixed position s and all their γs strictly after s. Likewise, we only consider subsequences in B with all αs up to some position t and all γs after that. We shall see that under these conditions, with a fixed *split* between αs and γs, it is possible to find an LCWIS in linear time.

Say, we try and see how long a sequence we can build if we started with exactly i many αs. We determine the ith pair of αs from the left and then count the number of βs in A and B up to the split (s, t). There are $p = \mathtt{Num}_{A,\beta}[s] - \mathtt{Num}_{A,\beta}[\mathtt{Pos}_{A,\alpha}[i]]$ such βs in A and $q = \mathtt{Num}_{B,\beta}[t] - \mathtt{Num}_{B,\beta}[\mathtt{Pos}_{B,\alpha}[i]]$ in B.

Assume $p \leq q$ for the moment. For the three values i, p, q, we define a piecewise-linear function $f_i^{s,t}$ consisting of a slope-1 segment from $(0, i + p)$ to $(q - p, i + q)$ and a horizontal extension from that point to infinity as shown in the left diagram of Figure 2.

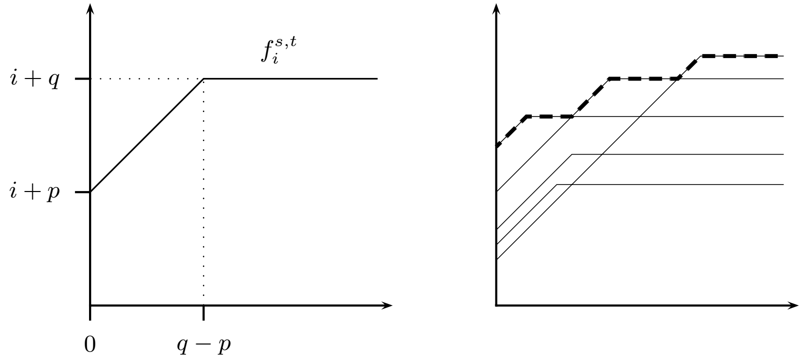


Fig. 2. Split diagrams

What is the purpose of this function? Assume we tried to find a long common subsequence by matching exactly j many γs in the two sequences. We would align these j pairs as far to the right as possible in order to gain as many βs as possible. So count the number of βs between position s and the leftmost matched γ in A and likewise in B. Say, there are x such βs in A and y in the respective part of B. We can now use our function $f_i^{s,t}$ to obtain the length of an LCWIS of type $\alpha^i\beta^*\gamma^j$: Compute the surplus $z = x - y$ of unmatchable βs in A on the right (assuming $x \geq y$ for the moment) and read off the function value of $f_i^{s,t}$ for that argument. The value $f_i^{s,t}(z)$ tells us exactly how long a subsequence we can build to the left of the split if we throw in a surplus of z βs into A.

For example, with no extra βs from the right, we only get $\min(p, q) = p$ many pairs of βs, which together with the i αs yield a sequence of length $f_i^{s,t}(0) = i + p$. If we have $q - p$ free βs on the right, we could get a sequence of length $f(q - p) = i + q$. More βs would not bring an advantage, which is expressed in the stagnation of the function f beyond $q - p$. The case $q > p$, which we had originally excluded for cleaner presentation, is simply covered by a function $\bar{f}_i^{s,t}$, defined in the obvious way to handle free βs on the right of the split in sequence B.

Of course, we have not gained anything yet from the function $f_i^{s,t}$. The trick is now to draw the functions $f_i^{s,t}$ for all values of i into *one* diagram. Their pointwise maximum $f^{s,t}$, the *upper envelope* of their plots, indicated in the right of Figure 2, gives us the best possible length to the left of the split for any surplus of βs from the right.

Lemma 3. *Amongst all subsequences that have all their αs to the left and all γs to the right of a fixed split (s, t), we can find an LCWIS in linear time.*

In order to turn the split technique into a fast algorithm for the general case, where we do not have any pre-knowledge about good splits, we will have to refine it a little further. If we know that there is an LCWIS with many βs, we can apply Lemma 3 immediately.

Theorem 2. *For two length-n sequences over three letters $\alpha < \beta < \gamma$, we can find an LCWIS that contains at least rn many βs ($r \in (0, 1)$) in $O(n/r^2)$ time.*

Proof. Put a marker every rn positions in A and also in B. Test all $\lceil 1/r \rceil^2$ candidate splits at marker pairs. Any $\alpha^* \beta^{\lceil rn \rceil} \beta^* \gamma^*$ subsequence must cover at least one of those pairs with its β-section. Hence we will find it. □

3.4 A Hierarchy of Splits

In the general case, when we need to make sure that we identify subsequences with only a few βs, we need a few tricks to further reduce the number of splits. To this end, first note that we may restrict attention to splits (s, t) that are given by left-aligned α-matches: The collection $\mathcal{S}$ of all splits of the form $(\mathtt{Pos}_{A,\alpha}[i], \mathtt{Pos}_{B,\alpha}[i])$ suffices to find an LCWIS.

Note that $\mathcal{S}$ comes with a natural linear order since no two of its splits cross and hence, $|\mathcal{S}| = O(n)$. Yet, if we drew a complete split diagram for every split in $\mathcal{S}$, we would still face a quadratic running-time. To reduce the work, we avoid drawing complete diagrams for all splits but spread information over splits. Therefore, assign levels to the splits in $\mathcal{S}$: let the level of the ith split (counting from left) be the index of the least significant bit equal to one in the binary representation of i. This scheme has the nice property that between any two splits on the same level there lies another split on a higher level.

Conceptually, our algorithm proceeds in two sweeps over the sequences. In the first sweep it constructs a split diagram for each of the splits in $\mathcal{S}$. However, not all left-side configurations are entered into all diagrams. For each integer i, match the first i αs from A and B and enter the corresponding functions into the split diagram of the closest split (s, t) to the right on each level. This means that the effect of starting with exactly i αs is entered into $O(\log |\mathcal{S}|) = O(\log n)$ diagrams. After all diagrams are prepared, the algorithm makes a second sweep of the sequences forming all right-aligned matches of γs. For each such partial subsequence we then query the split diagrams for the closest split to the left on each level to obtain the maximum length of an LCWIS with these many γs. A formal description of the algorithm is given in the full version of this paper.

The two sweeps can be implemented to run in $O(m + n \log n)$ time as follows. During the first sweep we simply create a list of $O(n \log n)$ quadruples (i, p, q, s) that represent the contents of the $O(n)$ splitters: s is the identity of a splitter and (i, p, q) are the parameters that define one of the functions illustrated in the left of Figure 2. Similarly, during the second sweep we construct a list of $O(n \log n)$ quadruples (i, p, q, s) where (i, p, q) is a query and s is the splitter on which it is to be performed. After bucket-sorting each list, all queries can be answered by a simultaneous linear scan of the lists.

Theorem 3. *We can find an LCWIS of two three-letter sequences of lengths m and n, with $m \geq n$, in $O(m + n \log n)$ time.*

4 Multiple Sequences

In this section we consider the problem of finding an LCIS of k length-n sequences, for $k \geq 3$. We will denote the sequences by $A^1 = (a^1_1, \ldots, a^1_n)$, $A^2 = (a^2_1, \ldots, a^2_n)$, ..., $A^k = (a^k_1, \ldots, a^k_n)$. A *match* is a vector $(i_1, i_2, \ldots, i_k)$ of indices such that $a^1_{i_1} = a^2_{i_2} = \cdots = a^k_{i_k}$. Let r be the number of matches. Chan et al. [4] showed that an LCIS can be found in $O(\min(kr^2, kr \log\sigma \log^{k-1} r) + kSort_\Sigma(n))$ time (they present two algorithms, each corresponding to one of the terms in the min). We present a simpler solution which replaces the second term by $O(r \log^{k-1} r \log\log r)$.

We denote the ith coordinate of a vector v by $v[i]$, and the alphabet symbol corresponding to the match described by a vector v will be denoted $s(v)$. A vector v *dominates* a vector v' if $v[i] > v'[i]$ for all $1 \leq i \leq k$, and we write $v' < v$. Clearly, an LCIS corresponds to a sequence $v_1, \ldots, v_\ell$ of matches such that $v_1 < v_2 < \cdots < v_\ell$ and $s(v_1) < s(v_2) < \cdots < s(v_\ell)$.

To find an LCIS, we use a data structure by Gabow et al. [6, Theorem 3.3], which stores a fixed set of n vectors from $\{1, \ldots, n\}^k$. Initially all vectors are *inactive*. The data structure supports the following two operations:

1. *Activate* a vector with an associated priority.
2. A query of the form "what is the maximum priority of an active vector that is dominated by a vector p?"

A query takes $O(\log^{k-1} n \log\log n)$ time and the total time for at most n activations is $O(n \log^{k-1} n \log\log n)$. The data structure requires $O(n \log^{k-1} n)$ preprocessing time and space.

Each of the r matches $v = (v_1, \ldots, v_k)$ corresponds to a vector. The priority of v will be the length of the longest LCIS that ends at the match v. We will consider the matches by non-decreasing order of their symbols. For each symbol s of the alphabet, we first compute the priority of every match v with $s(v) = s$. This is equal to 1 plus the maximum priority of a vector dominated by v. Then, we activate these vectors in the data structure with the priorities we have computed; they should be there when we compute the priorities for matches v with $s(v) > s$.

The algorithm applies to the case of a common weakly-increasing subsequence by the following modification: The matches will be considered by non-decreasing order of $s(v)$ as before, but within each symbol also in non-decreasing lexicographic order of v. For each match, we compute its priority and immediately activate it in the data structure (so that it is active when considering other matches with the same symbol). The lexicographic order ensures that if $v > v'$ then v' is in the data structure when v is considered.

Theorem 4. *An LCIS or LCWIS of k length-n sequences can be computed in $O(r \log^{k-1} r \log\log r)$ time, where r counts the number of match vectors.*

5 Outlook

The central question about the LCS problems is, whether it can be solved in $O(n^{2-\epsilon})$ time in general. It seems that with LCIS we face the same frontier. Our new algorithms are fast in many situations, but in general, we do not obtain subquadratic running-time, either.

On the other hand, LCWIS seems to behave very different from the other two problems. Our result shows that it behaves somewhat like a mixture of LCS and LCIS. While already the 2-letter problem is unsolved for LCS, finite alphabets are trivial for LCIS. With LCWIS now, we present near-linear-time solutions for alphabets with up to three letters, while it is unclear whether similar results can be obtained for all finite alphabets.

References

1. D. Aldous and P. Diaconis. Longest increasing subsequences: From patience sorting to the Baik-Deift-Johansson theorem. *Bull. AMS*, 36(4):413–432, 1999.
2. L. Bergroth, H. Hakonen, and T. Raita. A survey of longest common subsequence algorithms. In *SPIRE '00*, pages 39–48. IEEE Computer Society, 2000.
3. S. Bespamyatnikh and M. Segal. Enumerating longest increasing subsequences and patience sorting. *Inf. Process. Lett.*, 76(1-2):7–11, 2000.
4. W.-T. Chan, Y. Zhang, S. P.Y. Fung, D. Ye, and H. Zhu. Efficient Algorithms for Finding A Longest Common Increasing Subsequence. In *ISAAC '05*, 2005.
5. M.L. Fredman. On computing the length of longest increasing subsequences. *Discrete Mathematics*, 11(1):29–35, 1975.
6. H. N. Gabow, J. L. Bentley, and R. E. Tarjan. Scaling and related techniques for geometry problems. In *STOC '84*, pages 135–143. ACM Press, 1984.
7. D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Commun. ACM*, 18(6):341–343, 1975.
8. J. W. Hunt and T. G. Szymanski. A fast algorithm for computing longest common subsequences. *Commun. ACM*, 20(5):350–353, 1977.
9. W.J. Masek and M.S. Paterson. A faster algorithm computing string edit distances. *J. Comput. System Sci.*, 20:18–31, 1980.
10. E. M. McCreight. Priority search trees. *SIAM Journal on Computing*, 14(2):257–276, 1985.
11. P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977.
12. R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *J. ACM*, 21(1):168–173, 1974.
13. D. E. Willard. Log-logarithmic worst-case range queries are possible in space Theta(N). *Inf. Process. Lett.*, 17(2):81–84, 1983.
14. I.-H. Yang, C.-P. Huang, and K.-M. Chao. A fast algorithm for computing a longest common increasing subsequence. *Inf. Process. Lett.*, 93/5:249–253, 2005.

New Algorithms for Text Fingerprinting
(Extended Abstract)

Roman Kolpakov[1] and Mathieu Raffinot[2]

[1] Liapunov French-Russian Institute, Lomonosov Moscow State University,
Moscow, Russia
foroman@mail.ru

[2] CNRS, Poncelet Laboratory, Independent University of Moscow,
11 street Bolchoï Vlassievski, 119 002 Moscow, Russia
mathieu@raffinot.net

Abstract. Let $s = s_1..s_n$ be a text (or sequence) on a finite alphabet Σ. A fingerprint in s is the set of distinct characters contained in one of its substrings. Fingerprinting a text consists of computing the set $\mathcal{F}$ of all fingerprints of all its substrings and being able to efficiently answer several questions on this set. A given fingerprint $f \in \mathcal{F}$ is represented by a binary array, F, of size $|\Sigma|$ named a fingerprint table. A fingerprint, $f \in \mathcal{F}$, admits a number of maximal locations (i, j) in S, that is the alphabet of $s_i..s_j$ is f and $s_{i\ \ 1}, s_{j+1}$, if defined, are not in f. The total number of maximal locations is $\mathcal{L} \le n|\Sigma|+1$. We present new algorithms and a new data structure for the three problems: (1) compute $\mathcal{F}$; (2) given F, answer if F represents a fingerprint in $\mathcal{F}$; (3) given F, find all maximal locations of F in s. These problems are respectively solved in $O((\mathcal{L}+n)\log|\Sigma|)$, $\Theta(|\Sigma|)$, and $\Theta(|\Sigma|+K)$ time - where K is the number of maximal locations of F.

1 Introduction

We consider a finite ordered alphabet, Σ, and $s = s_1..s_n$ a sequence of n letters, $s_i \in \Sigma$. The set of all sequences over Σ is denoted Σ^*. The rank of each letter α in Σ is given by $f_\Sigma(\alpha)$ that ranges between 0 and $|\Sigma| - 1$. A sequence $v \in \Sigma^*$ is a factor or substring of s if $s = uvw$. The fingerprint, $C(s)$, of a sequence s is the set of distinct letters in s. By extension, $C_s(i, j)$ is the set of distinct letters in $s_i..s_j$. A fingerprint is represented below by a binary table of F of size $|\Sigma|$. If s contains the character α, $F[\alpha] \leftarrow 1$, otherwise $F[\alpha] \leftarrow 0$.

Definition 1. *Let $\mathcal{C}$ be a set of letters of Σ. A maximal location of $\mathcal{C}$ in $s = s_1..s_n$ is an interval $[i, j]$, $1 \le i \le j \le n$, such that*

$$(1)\quad C_s(i,j) = \mathcal{C}; \quad (2)\quad \text{if } i > 1, s_{i-1} \notin C_s(i,j); \quad (3)\quad \text{if } j < n, s_{j+1} \notin C_s(i,j)$$

We denote by $\mathcal{F}$ the number of distinct fingerprints and by $\mathcal{L}$ the number of maximal locations of all fingerprints of $\mathcal{F}$. In this paper, given a sequence s, we are interested in three main algorithmic problems: 1. Compute the set $\mathcal{F}$ of all fingerprints in s; 2. Given a fingerprint table F, find if F represents a fingerprint in $\mathcal{F}$ or not; 3. Given a fingerprint table F, find all the maximal locations of F in s.

M. Lewenstein and G. Valiente (Eds.): CPM 2006, LNCS 4009, pp. 342–353, 2006.

Efficient answers to these questions have many applications in information retrieval, computational biology and natural language processing [1]. The input alphabet Σ is considered to be the alphabet of the input sequence, thus $|\Sigma| \leq n$. The best actual algorithms solve Problem 1 in $\Theta(\min\{n|\Sigma|\log|\Sigma|, n^2\})$ time. The bound $\Theta(n|\Sigma|\log|\Sigma|)$ is that of the algorithm of Tsur in [4] that we present in detail. The $\Theta(n^2)$ bound is obtained using the algorithm of Didier also presented in [4], although this algorithm was first presented with $O(n^2 \log n)$ and $\Omega(n^2)$ time complexities in [3]. The $\log n$ gain between these two versions has been obtained using a lowest common ancestor algorithm (LCA). Problem 2 is solved in $O(|\Sigma|\log n)$ time and Problem 3 in $O(|\Sigma|\log n + n)$ time in [1, 4]. Surprisingly enough, and this a strong motivational factor for this paper, these complexities are independent of the sizes of $\mathcal{F}$ and $\mathcal{L}$, although many sequence families have few fingerprints or few maximal locations.

In this paper we present new algorithms and a new tree structure for solving these three problems. Problem 1 is solved in $O((|\mathcal{L}| + n)\log|\Sigma|)$ time. As $|\mathcal{L}| \leq n|\Sigma| + 1$, our algorithm is, at worst, as efficient as that of Tsur, but much more efficient on many sequence families. It can however be slower than that of Didier when $|\Sigma| = \Omega(n/\log n)$. Although, even in this case, the real complexity of our algorithm depends of the number of maximal locations that can be much less than $n|\Sigma|+1$. Our algorithm also has the advantage of being simple to implement in its real worst case complexity. Problem 2 is optimally solved in $\Theta(|\Sigma|)$ using a new tree structure, improving the fastest algorithm by $\log n$. Finally, Problem 3 can be solved either in $\Theta(|\Sigma| + K)$ time - where K is the number of maximal locations of F - using $\Theta(|\mathcal{F}|\log|\Sigma| + |\mathcal{L}|)$ space. To maintain continuity with the previous approaches, our algorithms improve a naming technique introduced in [1] and [4]. The paper is organized as follows. The original naming technique is presented first in Section 2. In Section 3 we present our new naming algorithm. In the next Section 4 we detail our tree data structure and the algorithmic improvements it permits. We do not provide any proof in this extended abstract. The interested reader should refer to [5] for details.

2 Fingerprints and Naming Technique

In this section we recall the naming technique introduced in [1] and then improved by Tsur in [4]. The naming technique is used to give a unique name to each fingerprint of a substring of s. We first describe the naming technique and then we explain how to use it to name all fingerprints of s.

Naming Technique. We assume for simplicity, but without loss of generality, that $|\Sigma|$ is a power of two. We consider a stack of $\log|\Sigma| + 1$ arrays on top of each other. Each level is numbered from 0. The lowest, called the fingerprint table, contains $|\Sigma|$ names that might be only [0] or [1]. Each other array contains half the number of names that the array it is placed on. The highest array only contains a single name that will be the name of the whole array. Such a name is called a fingerprint name. Fig 1 shows a simple example with $|\Sigma| = 8$.

<table>
<tr><td colspan="8">[7]</td></tr>
<tr><td colspan="4">[5]</td><td colspan="4">[6]</td></tr>
<tr><td colspan="2">[2]</td><td colspan="2">[2]</td><td colspan="2">[3]</td><td colspan="2">[4]</td></tr>
<tr><td>[1]</td><td>[0]</td><td>[1]</td><td>[0]</td><td>[1]</td><td>[1]</td><td>[0]</td><td>[0]</td></tr>
</table>

Fig. 1. Naming example

The names in the fingerprint table are only [0] or [1] and are given. Each cell, c, of an upper array represents two cells of the array it is placed on, and thus a pair of two names. The naming is done in the following way: for each level going from the lowest to the highest, if the cell represents a new pair of names, give this pair a new name and assign it to the cell. If the pair has already been named, place this name into the cell. In the example in Fig. 1, the name [2] is associated to $([1],[0])$ the first time this pair is encountered. The second time, this name is directly retrieved.

Naming All Fingerprints. A change in the lowest level of this array, that is changing a [1] to a [0] or a [0] to a [1] causes, at most, $\log|\Sigma|+1$ changes in the names on the path from the modified cell to the root. This property is used in the original algorithm of [1]. Their idea is to enumerate all fingerprints containing a fixed number k of different characters by shifting two indices $1 \leq i \leq j \leq |s|$ on the sequence. The algorithm first identifies a pair $(i_0 = 1, j_0)$ such that $s_{i_0}..s_{j_0}$ contains exactly k distinct characters. This fingerprint is named using the previous technique. Then the two indices are shifted to (i_1, j_1) that points the beginning and the end of the next substring containing exactly k different characters and with a different fingerprint than the previous one. The key point of the algorithm of [1] is that this new fingerprint only differs from the previous one in two positions in the lowest array. Updating the array of names thus requires, at most, $2\log|\Sigma|+2$ changes. *Complexity.* Each change in the name array requires checking whether a pair of names has already received a name or not. At each level there are, at most, n new names, and thus searching for the pair can be done in $\log n$ time using a balanced tree. For each value of k, initializing the array of names requires $O(|\Sigma|\log|\Sigma|)$ time. Then each new pair (at most n when reading the sequence) requires, at most, 2 global changes in the whole array, each requiring $O(\log|\Sigma|\log n)$. Thus, for each value of k, building the names requires $O(n\log|\Sigma|\log n)$ time and as $k = 1..|\Sigma|$, the whole algorithm takes $O(n|\Sigma|\log|\Sigma|\log n)$ time.

In [4] Tsur presented a faster algorithm to build all names. The algorithm still performs $|\Sigma|$ iterations in a similar way to the previous one, but fills the names level by level. The list of changes in each level over the whole sequence is recorded in an ordered list. This list is sorted using an $O(n)$ sort algorithm (for instance radix sort) and new names are given according to this sort. These new names are placed in the original list (the order of this list is important) that is used to build the initial list of the next level. A pseudo-code of the main part of the algorithm (slightly modified) can be found in [5].

We number the level from 1, the lowest, to $\log|\Sigma|+1$. For each k, $1 \leq k \leq |\Sigma|$, the initialization of the ordered list L_1 at level 1 is performed by reading the sequence. This list records the changes at level 1. In order to build it, we move two pointers on the sequence in exactly the same way as in the previous algorithm. When the first pair $(i_0 = 1, j_0)$ is encountered, the values in the fingerprint table A (the array of level 1) is registered in L_1 under the form of pairs $\{A[i], i\}$ for $i = 0..|\Sigma|-1$. The two pointers are then moved and for each new pair of pointers there are only two modifications in the array A. For each such modification, if A changed in position j, $0 \leq j \leq |\Sigma| - 1$, this change is recorded by adding $\{A[j], j\}$ to the end of L_1. At the end of this process, the ordered list L_1 records all changes to be performed at level 1.

This initial list is then used to compute all names of the cells in the second level. A table, FT, of $|\Sigma|$ names temporary records the pair of names to be coded. A list L'_1 of pairs of names is built in the following way. The first $|\Sigma|$ elements of L_1 are read to initialize FT. The list L'_1 is initialized with $|\Sigma|/2$ pairs built by reading FT. Then, the remaining of the list L_1 is read and for each new element $\{[a], j\}$ *(1)* the table FT is changed in position j by $FT \leftarrow [a]$ and *(2)* the pair $\{(FT[2\lfloor j/2 \rfloor], FT[2\lfloor j/2 \rfloor + 1]), j/2\}$ if added to the end of L'_1. This means that in cell $j/2$ of the second level a name has to be given to the name pair $(FT[2\lfloor j/2 \rfloor], FT[2\lfloor j/2 \rfloor + 1])$.

At this point L'_1 records the list of changes to be made in the cells at level 2 and the pairs of names that must receive a name. The pairs in this list are then sorted in lexicological order (through a radix sort) and a new name is assigned to each distinct pair of names (n_1, n_2). A new list L_2 is built from L'_1 (keeping the initial order of L'_1 and thus of L_1) by replacing each pair with its new name. For instance, if $\{([1], [0]), 1\}$ was in the list L'_1 and if the pair $([1], [0])$ received the new name $[2]$, then L_2 now contains $\{[2], 1\}$. The list L_2 is the input at level 2 and the same process is repeated to obtain the names in the third level, and so on. The last list $L_{\log|\Sigma|+1}$ contains the names of all fingerprints containing exactly k distinct characters in the original sequence. *Complexity.* The initialization of L_1 is $\Theta(n)$ time. Then a linear sort of at most $\Theta(n)$ elements is performed for every level. As there are $\log|\Sigma| + 1$ levels, the process is $\Theta(n \log|\Sigma|)$ time. As $k = 1..|\Sigma|$, the whole complexity is $\Theta(n|\Sigma|\log|\Sigma|)$ time. This saves $\log n$ over the previous algorithm.

3 New Algorithm to Compute All Fingerprints

The faster algorithm of Section 2 is $\Theta(n|\Sigma|\log|\Sigma|)$ time. This complexity is independent of the number of maximal locations, $\mathcal{L}$, although this is one of the main parameters of the fingerprinting problem. The naming algorithm we present depends on $\mathcal{L}$ and its complexity is $O((\mathcal{L} + n)\log|\Sigma|)$ time. As $\mathcal{L}$ is, at most, $n|\Sigma| + 1$, but is much less on many sequence families, our algorithm is faster than or as efficient as previous ones.

Moreover, the fingerprint tree we present in the next section permits efficient searching for a given fingerprint to appear in the sequence. However, it requires

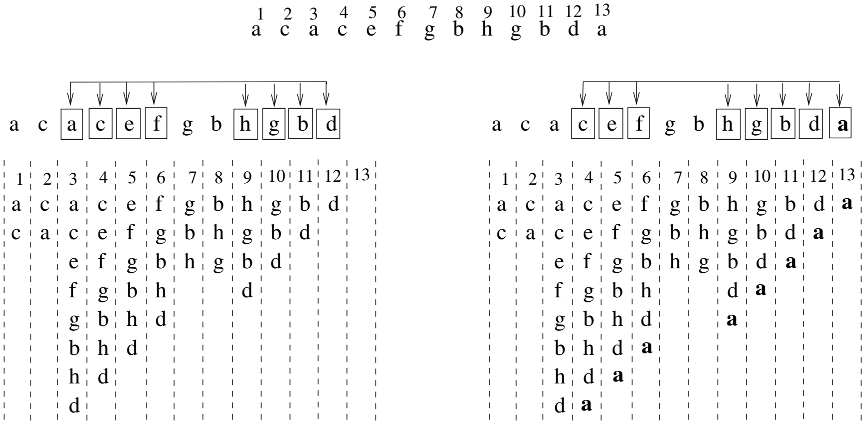


Fig. 2. A (schematic) step of algorithm FINGERPRINT_CHANGES that is the first phase for computing all fingerprints. We add the character **a** in table TN_1.

all fingerprint names to be globally built on the same name subsets, which is not the case of the names generated by the two algorithms of Section 2. It also requires these names to be sorted in the lexicographical order of their fingerprint tables. We present in this section a new naming algorithm that fulfills these requirements.

The main idea of the second algorithm of Section 2 is to record the changes in the fingerprint tables before computing names. We reuse this approach but we (a) process the whole sequence once before computing all names and (b) record only changes corresponding to maximal locations. Point (a) is achieved by keeping for each position i in the sequence a virtual list of all fingerprint tables of substrings $s_i..s_j$, $i \leq j \leq n$, beginning in i. This list is virtual in the sense that instead of keeping a list of fingerprint tables we only record all changes in the fingerprint table in i. At the beginning all fingerprint tables are considered *empty*, that is full of $|\Sigma|$ zeros. Point (b) consists in considering only positions that correspond to maximal locations.

Our algorithm runs in two phases. The first phase identifies on the sequence the fingerprints that must be encoded. The second phase builds names for all these fingerprints. The sequence is thus read once.

We assume that without loss of generality below the input sequence does not contain two consecutive repeating characters. Such a sequence is named *simple.* The segments of repeating characters (say α) of any input sequence can be reduced to a unique occurrence of α. The two sequences have the same sets, $\mathcal{F}$, and the same set, $\mathcal{L}$, up to small changes in the bounds. These changes can, however, be simply retrieved in $\Theta(1)$ per maximal location. The reducing algorithm is $\Theta(n)$. This technical trick really simplifies the algorithms we present by removing many straightforward technical cases.

First Phase. Let $s = s_1..s_n$ be a sequence of characters over Σ. For each character $\alpha \in \Sigma$ we define R_s^α as the indice in s of the rightmost occurrence

```
FINGERPRINT_CHANGES($s = s_1..s_n$)
Let $TN_1[1..n]$ a table of $n$ lists
    $TN_1[i]$ are all initialized to an empty list
$L \leftarrow (0)$
For i=1..n Do
    $\alpha \leftarrow s_i$
    add $\{[1], f_\Sigma(\alpha)\}$ on top of $TN_1[i]$
    $j \leftarrow$ top element of $L$
    While $j > 0$ AND $s_j \neq \alpha$ Do
        add $\{[1], f_\Sigma(\alpha)\}$ on top of $TN_1[j]$
        $j \leftarrow$ previous element in $L$
    End of while
    If $j > 0$ Then /* there is a indice of an $\alpha$ in $L$ */
        remove $j$ from $L$
    End of if
    add $i$ on top of $L$
End of for
```

Fig. 3. Computing all fingerprint changes as a first phase of the new naming procedure

of α in s, and we fix $R_s^\alpha = 0$ if there is no such occurrence. We define the last occurrence list L_s as being the sorted list (in increasing order) of all indices in s of character last occurrences. We add an arbitrary 0 (if not already there) before all indices; Notice that the last indice of L_s must be n. Thus, formally, $L_s = (0, R_s^{\alpha_1}, R_s^{\alpha_2}, \ldots, R_s^{\alpha_k} = n)$.

Suppose now that $C_s(i, j)$ is known for all pairs $1 \leq i \leq j \leq |s|$. When concatenating a letter α to s, we aim to compute all $C_{s\alpha}(i, j)$ for $1 \leq i \leq j \leq |s\alpha|$.

Lemma 1. *Let $s = s_1..s_n$, $s_i, \alpha \in \Sigma$ and $L_s = (0, R_s^{\alpha_1}, R_s^{\alpha_2}, \ldots, R_s^{\alpha_k} = n)$. The following properties hold:*

1. *For all pairs $1 \leq i \leq j \leq n$, $C_{s\alpha}(i, j) = C_s(i, j)$.*
2. *Let z such that $L_s[l-1] < z \leq L_s[l]$, $0 < l \leq k$. Then $C_{s\alpha}(z, |s\alpha|) = C_s(L_s[l], |s|) \cup \{\alpha\}$.*
3. *If $R_s^\alpha > 0$, let $0 < z \leq R_s^\alpha$. Then $C_{s\alpha}(z, |s\alpha|) = C_s(z, |s|)$.*

The first phase of the algorithm reads the sequence s one character after the other. Assume that we have already read and processed the characters up to position j. For each position $k = 1..j$ a list encodes the series of fingerprint changes to code for $C(k, j)$. We read the character $\alpha = s_{j+1}$. According to lemma 1 the algorithm goes down the indice list of last character occurrences until either (a) the same character α is encountered (points 2 and 3 of lemma 1), or (b) the beginning of the list is reached (point 2 of lemma 1). For each indice i touched in this list, we add the character α to the list representing $C(i, j)$ in order to obtain $C(i, j+1)$. The list of last occurrences is then updated by removing the indice of the previous occurrence of α and adding $j + 1$ as the new indice of α. The first phase of the algorithm is called FINGERPRINT_CHANGES and its pseudo-code is given in Figure 3. A step of the algorithm is shown in Figure 2.

Second Phase. The second phase is based on the second algorithm of section 2. It remains to name all fingerprints appearing as a prefix of each list in TN_1. This is done with the algorithm NAME_ALL_LISTS for which the pseudo-code is given in Figure 4. In a similar way to the second algorithm of Section 2, $\log|\Sigma|$ iterations are performed for each fingerprint array level.

In each iteration, each list in TN_k is read to build a corresponding list of cell changes in level $k+1$ (lines 5-19). A new list table TN'_k is thus built and records all these new lists. The pair of names in TN'_k are sorted altogether in lexicographic order through a radix sort (line 23). A new name is then given to each different pair (line 24). A new list table TN_{k+1} is then built by copying TN'_k, but replacing each name pair with its new name (line 25). This list is the input list of the next iteration of the general loop (lines 2-27).

```
NAME_ALL_LISTS($TN_1[1..n]$ initial table of fingerprint changes)
 1. $ninit_1 \leftarrow [0]$
 2. For $k = 1..\log|\Sigma|$ Do
 3.      $FT_k \leftarrow$ name table of size $|\Sigma|/2^{k\ \ 1}$ all initialized to $ninit_k$
 4.      Let $TN_k[1..n]$ be a table of $n$ lists.
 5.      For $i = 1..n$ Do
 6.           initialize $TN_k[i]$ to the empty list
 7.           $L \leftarrow$ first element of $TN_k[i]$
 8.           While $L$ exists Do
 9.                $\{[a], j\} \leftarrow L$
10.                $FT_k[j] \leftarrow [a]$
11.                add $\{(FT_k[2\lfloor j/2\rfloor], FT_k[2\lfloor j/2\rfloor + 1]), j/2\}$ to end of $TN_k[i]$
12.                $L \leftarrow$ next element in $TN_k[i]$
13.           End of while
14.           $L \leftarrow$ first element of $TN_k[i]$
15.           While $L$ exists Do
16.                $\{[a], j\} \leftarrow L$
17.                $FT_k[j] \leftarrow ninit_k$
18.                $L \leftarrow$ next element in $TN_k[i]$
19.           End of while
20.      End of for
21.      $Sl \leftarrow$ list of all cell pairs in $TN_k$
22.      add the pair $(ninit_k, ninit_k)$ to $Sl$
23.      sort $Sl$ in lexicographical order
24.      give new names for each different pair in $Sl$
25.      build $TN_{k+1}$ by copying $TN_k$ but replacing each pair by its new name
26.      $ninit_{k+1} \leftarrow$ name of the pair $(ninit_k, ninit_k)$
27. End of for
```

Fig. 4. Naming all fingerprints in all lists of TN_1

Special care is required for the initialization process of the temporary table FT of size $|\Sigma|/2^{k-1}$. The table is initialized once (line 3) and reinitialized after coding each list of cell changes in TN'_k. However, for complexity issues, this reinitialization is performed in an amount of time proportional to the size of this list by simply erasing the changes that have been made (lines 14-19).

The result of the first iteration of the NAME_ALL_LISTS algorithm on the TN_1 table is given in Figure 5.

1	2	3	4	5	6	7	8
{([1],[0]),0}	{([1],[0]),1}	{([1],[0]),0}	{([1],[0]),1}	{([1],[0]),2}	{([0],[1]),2}	{([1],[0]),3}	{([0],[1]),0}
{([1],[0]),1}	{([1],[0]),0}	{([1],[0]),1}	{([1],[0]),2}	{([1],[1]),2}	{([1],[0]),3}	{([0],[1]),0}	{([0],[1]),3}
		{([1],[0]),2}	{([1],[1]),2}	{([1],[0]),3}	{([0],[1]),0}	{([1],[1]),3}	{([1],[1]),3}
		{([1],[1]),2}	{([1],[0]),3}	{([0],[1]),0}	{([1],[1]),3}		
		{([1],[0]),3}	{([0],[1]),0}	{([1],[1]),3}	{([0],[1]),1}		
		{([1],[1]),0}	{([1],[1]),3}	{([0],[1]),1}	{([1],[1]),0}		
		{([1],[1]),3}	{([1],[1]),1}	{([1],[1]),0}			
		{([1],[1]),1}	{([1],[1]),0}				

Fig. 5. First columns of the table TN_1 of lists of cell changes

The last sort of the table $TN_{\log|\Sigma|}$ records the fingerprint names. The NAME_ALL_LISTS algorithm obviously insures that these names are sorted in the lexical order of their fingerprint tables.

Lemma 2. *Let $s = s_1..s_n$, $s_i, \alpha \in \Sigma$ and $L_s = (0, R_s^{\alpha_1}, R_s^{\alpha_2}, \ldots, R_s^{\alpha_k} = n)$. Let z be the indice of R_s^{α} in L_s. Then for all indices l in L_s such that $l > z$, interval $[l+1..|s|]$ is a maximal location. If $z = 0$, $[1..|s|]$ is also a maximal location.*

Theorem 1. *Our algorithm names all distinct fingerprints of s in $O((\mathcal{L} + n)\log|\Sigma|)$ time.*

It remains to prove that $\mathcal{L}$ is bounded by $n(|\Sigma| + 1)$.

Proposition 1. *The number $\mathcal{L}$ of maximal locations is bounded by $n(|\Sigma| + 1)$.*

Corollary 1. *Our naming algorithm is $\Theta(n|\Sigma|\log|\Sigma|)$ worst case time.*

Computing all Maximal Locations. In order to efficiently solve problem 3, we associate each fingerprint name with its maximal locations. Our approach is to compute the maximal locations during the first phase of the previous algorithm and maintain them through the second phase.

Proposition 2. *Let $s = s_1..s_n$ and $[i, j]$, $0 < i \leq j \leq n$, be a maximal location in s of a fingerprint f; let $w = s_1..s_j$ and $L_w = (0, R_w^{\alpha_1}, R_w^{\alpha_2}, \ldots, R_w^{\alpha_k} = j)$. There exits a unique p in L_w such that $f = C(p+1, j) = \cup_{R_w^{\alpha_h} > p} \{\alpha_h\}$.*

We modify the algorithm FINGERPRINT_CHANGES to associate each new maximal location with its alphabet that is to be coded in the second phase of the naming. Each time a new j (line 7) except the first one is encountered at iteration i (lines 4-16), the maximal location (insured by lemma 2) $[j+1, i-1]$

is associated with its alphabet, which is, according to proposition 2, the last but one element in the column corresponding to the indice after j in $L_{s_1 \dots s_{i-1}}$. Notice that according to proposition 2 a final iteration of the loop i (lines 4-16) is required to compute the maximal locations appearing at the end of the sequence. In this last iteration, the maximal location $[j+1, n]$ is associated with the *last* element in the column corresponding to the indice after j in $L_{s_1 \dots s_n}$. This technical add-on could also be fixed by adding a last virtual character to the input sequence. After the first phase, names are built using the NAME_ALL_LISTS algorithm, slightly modified for keeping track in all cell change lists of the associated maximal locations. A last phase is necessary to group the set of maximal locations of each fingerprint name.

4 Fingerprint Tree

Once the fingerprints have been named by one of the two algorithms of Section 2, searching for a given fingerprint to appear in the sequence can be carried out in $O(|\Sigma| \log n)$ time by a similar process to that in the first algorithm, that is, filling the level from bottom to top and checking for each cell if a name has already been given to a pair of cells [1].

This searching could be made $O(|\Sigma|)$ expected time using perfect hashing on name pairs [2] of each level. The hash tables would require $O(\mathcal{F} \log |\Sigma|))$ expected memory space and could be built in $O(\mathcal{F} \log |\Sigma|))$ expected time. However, in the worst case, these hash tables could require $O(\mathcal{F}^2 \log |\Sigma|))$ memory space and be built in $O(\mathcal{F}^2 \log |\Sigma|))$ time.

We propose another approach for solving this problem in $O(|\Sigma|)$ worst case time, requiring in the worst case $O(\mathcal{F})$ additional memory space and $O(\mathcal{F} \log |\Sigma|)$ additional preprocessing time. For these purposes we present a new tree structure of size $O(\mathcal{F})$ that permits searching for a given fingerprint in $O(|\Sigma|)$ time. This tree can be built in $O(\mathcal{F} \log |\Sigma|)$ time. The searching phase requires the names to be given from the same set of names for all the $k = 1..|\Sigma|$ iterations. Therefore, the names in this tree cannot be generated using one of the first algorithms of Section 2. The construction itself requires the fingerprint names to be sorted in the lexicographical order of their corresponding fingerprint tables, which is a property of the naming algorithm we presented. The fingerprint tree is a binary tree in which each fingerprint name is a leaf. Edges are labeled with a triplet $\{DT, l, r\}$ where DT is either *(i)* a single name [1] or [0] if one of l or r is equal to 1 and the other to 0; or *(ii)* a pair of names (n_l, n_r) and $1 \le l \le |\Sigma|$ and $1 \le r \le |\Sigma|$ two lengths. The pair of names (n_l, n_r) is related to l and r by being the lowest pair of consecutive names at the same level that cover the segment from the beginning of l to the end of r in the name array of one of the leaves in the subtree. Figure 6 shows an edge label.

Building the Tree. We denote by *name tree* the tree formed by recursively developing all name pairs deriving from a given name.

Definition 2. *Let F_1 and F_2 be two fingerprint tables. The Longest Common Prefix* (lcp) *of F_1 and F_2 is the longest equal sub-table beginning F_1 and F_2.*

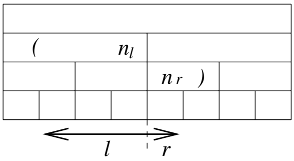


Fig. 6. Label of a transition. The pair (n_l, n_r) is the lowest consecutive pair of names covering the segments l and r.

By extension, we denote the lcp of two fingerprint names the lcp of the fingerprint tables they encode. Let $n_1 \dots n_{\mathcal{F}}$ be the fingerprint names whose fingerprint tables are sorted in lexicographic order (requirement). The construction of the tree is done in $O(\mathcal{F} \log |\Sigma|)$ time in three phases. *(i)* Compute for every two consecutive names n_i, n_{i+1} their lcp in $LCP[i, i+1]$. *(ii)* Build a skeleton of the tree containing all necessary nodes but in which each edge is labeled by an interval $[k..l]$. This interval denotes that the label of the edge must code for the interval $[k..l]$ in any fingerprint table of the leaves in the subtree starting at this edge. *(iii)* Build the label of each edge. We now detail these three steps.

Computing the Lcp. We begin with the two given names and a current lcp fixed to $|\Sigma|$. The lcp of two fingerprint names can be computed by simultaneously going down each name tree. At each step of the algorithm we compare two names. If these names are equal, the resulting lcp is the current lcp. Otherwise, each name corresponds to two other names (unless they are [0] or [1]). If their first name (the left part) is equal, then the lcp is the size of this left part (current lcp /2) plus the lcp of the second (right part). Otherwise the lcp is the current lcp plus the lcp of the first part. A pseudo-code of a recursive version of the algorithm can be found in [5].

Lemma 3. *Let n_1 and n_2 be two fingerprint names. The lcp of n_1 and n_2 can be computed in $O(\log |\Sigma|)$ time.*

The table LCP is built by $\mathcal{F} - 1$ iterations of algorithm lcprec. The whole complexity of this first phase is thus $O(\mathcal{F} \log |\Sigma|)$ time.

Building a Skeleton Tree. We build a skeleton of the fingerprint tree by adding a branch and a leaf for each name n_i. This construction is illustrated in Fig. 7.

The initial tree is a single root. This branch is plugged at depth $LCP[i-1, i]$ to the branch previously built for the name n_{i-1}. This is done by going up in the tree from the last leaf created for n_{i-1} (denoted $L(n_{i_1})$) to the root. On this path, we isolate the first node q with depth d less than or equal to $LCP[i-1, i]$. If d is exactly $LCP[i-1, i]$, then we just plug a new branch to q. Otherwise we create a new node p with depth $LCP[i-1, i]$ that becomes a new child of q. This node p now has two children, one corresponding to the previous subtree of q. The other is the new branch that is terminated with the new leaf $L(n_i)$ from where the next step begins. A pseudo-code of this construction is given in Fig. 8. In this code, the depth of any $L(n_i)$ is fixed to $|\Sigma|$.

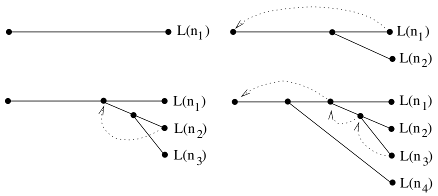


Fig. 7. Building the skeleton of the fingerprint tree. Backward dashed edges illustrate searching for the position of the new branch.

```
BUILD_SKELETON_TREE($n_1 \ldots n$ , $LCP$)
  $depth(root) \leftarrow 0$
  branch $L(n_1)$ on $root$
  $current \leftarrow root$
  $prev \leftarrow L(n_1)$
  For $i = 2..|\mathcal{F}|$ Do
      While $depth(current) > LCP[i-1, i]$ Do
          $prev \leftarrow current$
          $current \leftarrow father(current)$
      End of while
      If $depth(current) = LCP[i-1, i]$ Then
          branch $L(n_i)$ on $current$
      Else
          cut the edge ($current$,$prev$) with $newnode$
          $depth(newnode) \leftarrow LCP[i-1, i]$
          branch $L(n_i)$ on $newnode$
          $current \leftarrow newnode$
      End of if
      $prev \leftarrow L(n_i)$
  End of for
```

Fig. 8. Computing the skeleton tree

The complexity of building the skeleton tree is $O(\mathcal{F})$ time since each node is at most visited twice, once when created and at most once when going up the nodes to search for the position of the new branch.

Building Edge Labels. Once the skeleton tree has been built, each node has a depth associated. Each edge from node q to node p corresponds to the segment $[depth(q) + 1..depth(p)]$ in each fingerprint array of each leaf in the subtree. This segment permits efficient coding of each edge in $O(\log |\Sigma|)$ time. Coding all edges of the fingerprint tree thus requires $O(\mathcal{F} \log |\Sigma|)$ time.

Searching for a Fingerprint in the Tree. The coding of each edge of the fingerprint tree has an important property for the time complexity of the search.

Lemma 4. *Let $\{(n_l, n_r), l, r\}$ be the label of an edge in a fingerprint tree. The level of the pair of consecutive names (n_l, n_r) is $O(\log(l + r))$.*

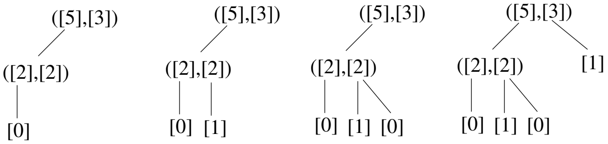


Fig. 9. Decoding the edge label $\{(5,3),3,1\}$

This property permits the decoding of an edge of the fingerprint tree in a time proportional to the length of this edge. The idea is to traverse the tree formed by the pair of names in a prefix order. Figure 9 illustrates this traversal. As the height of this tree is logarithmic in the length of the segment, the total complexity of decoding an edge $\{(n_l, n_r), l, r\}$ is $O(l + r)$ time. Therefore, searching in the fingerprint tree for a fingerprint given in the form of a fingerprint table of length $|\Sigma|$ is $O(|\Sigma|)$ time. The algorithmic results of this section can be stated in the following theorem.

Theorem 2. *Building a fingerprint tree takes $\Theta(|\mathcal{F}| \log |\Sigma|)$ time and space. Searching for a given fingerprint, F, in this tree takes $\Theta(|\Sigma|)$ time.*

Considering the maximal locations of a given fingerprint F, two main options exist. The first is to search for the maximal locations once F is known to appear in the sequence. This can be performed by reading the sequence in $\Theta(n)$ time [1]. No extra memory requirement is necessary. The second is to attach to each leaf in the tree the set of its maximal locations computed using the modified naming algorithm of 3. This allows searching for all maximal locations of F in $\Theta(|\Sigma| + K)$ time, where K is the number of maximal locations of F. An extra $\Theta(\mathcal{L})$ memory is, however, necessary to record all the maximal locations.

Acknowledgment. We thank Guillaume Blin for a complete proofreading of a draft version of this article. We also thank two anonymous referees for pointing us the perfect hashing approach.

References

1. A. Amir, A. Apostolico, G. M. Landau, and G. Satta. Efficient text fingerprinting via parikh mapping. *J. Discrete Algorithms*, 1(5-6):409–421, 2003.
2. T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms, 2nd Edition.* MIT Press, Cambridge, MA, USA, 2001.
3. G. Didier. Common intervals of two sequences. In *WABI*, number 2812 in Lecture Notes in Computer Science, pages 17–24. Springer-Verlag, Berlin, 2003.
4. G. Didier, T. Schmidt, J. Stoye, and D. Tsur. Character sets of strings. 2004. Submitted.
5. R. Kolpakov and M. Raffinot. New Algorithms for Text Fingerprinting. *unpublished*, 2006. Submitted. `http://www-igm.univ-mlv.fr/ raffinot/ftp/ fingerprint.pdf`.

Sublinear Algorithms for Parameterized Matching

Leena Salmela and Jorma Tarhio*

Helsinki University of Technology
{lsalmela, tarhio}@cs.hut.fi

Abstract. Two strings parameterize match if there is a bijection that transforms the first string character by character into the second string. This problem has been studied in both one and two dimensions but the research has been centered on developing algorithms with good worst-case performance. We present algorithms that solve this problem in sublinear time on average for moderately repetitive patterns.

1 Introduction

In the parameterized matching problem a text and a pattern is given and the task is to find all substrings of the text that can be transformed into the pattern by using a bijection on the alphabet. This problem was first considered by Baker [5] with an application to software maintenance. Another application of parameterized matching is plagiarism detection [11].

Later the parameterized matching problem has been investigated in two dimensions by Amir et al. [1] and Hazay et al. [14]. This two-dimensional problem has a fairly obvious application in image searching. Parameterized matching can find an image even if its color map has been changed. Other related work includes parameterized matching of multiple patterns [16], parameterized matching with mismatches [13] and approximate parameterized search [7].

Previous research of parameterized matching has been centered in developing algorithms with good worst-case performance. Some effort to develop an algorithm fast on average was made by Baker [6] who developed an algorithm based on the famous Boyer-Moore algorithm [8] but the average case complexity was not analyzed. In fact the algorithm uses a linear preprocessing with respect to the length of the text and thus loses the good average case complexity of the Boyer-Moore algorithm.

In this paper we introduce new algorithms that are sublinear on average. We present practical solutions for both the one-dimensional and two-dimensional parameterized matching problems. We analyze the time complexities of the algorithms for random texts and moderately repetitive patterns, and present experimental results for certain interesting classes of patterns.

2 Definitions

Let S and S' be equal size strings of characters drawn from the alphabet Σ. S and S' parameterize match (or p-match for short) if there exists a bijection π such

* Work by Jorma Tarhio was supported by Academy of Finland.

M. Lewenstein and G. Valiente (Eds.): CPM 2006, LNCS 4009, pp. 354–364, 2006.

that for each i $S[i] = \pi(S'[i])$. So strings 'abac' and 'bcba' p-match because the bijection $\pi(a) = c, \pi(b) = a, \pi(c) = b$ transforms 'bcba' into 'abac'. On the other hand strings 'aabb' and 'acbb' do not p-match because a bijection cannot map both 'a' and 'c' to 'a' and thus there is no bijection that can transform 'aabb' to 'acbb'.

Let us now define the parameterized matching problem. In the one dimensional case, we are given a text $T[1..n]$ and a pattern $P[1..m]$ in alphabet Σ and the task is to find all substrings of the text that p-match with the pattern. In the two-dimensional case the input is a text T of size $n \times n$ and a pattern P of size $m \times m$. The task then is to find all those $m \times m$ substrings of the text that p-match to P.

Two disjoint alphabets were used in the original definition of the parameterized matching problem by Baker. One of the alphabets was a fixed alphabet like in the standard string matching problem and the other one was a parameterized alphabet like our Σ. Both the pattern and the text could contain characters from both alphabets but characters from the fixed alphabet were required to match exactly. We decided to use only the parameterized alphabet because that is natural for the two dimensional problem of image search and we wished to give a unified treatment to both the one dimensional and two dimensional cases.

Many of the algorithms make use of so called predecessor strings. A string S is transformed into a predecessor string as follows. If a character in position i has occurred earlier in the string in position j the position i in the predecessor string contains $i - j$. Otherwise the predecessor string contains 0. For example the string 'aabac' is transformed into 0-1-0-2-0. Now it can be fairly easily seen that if two strings p-match, their predecessor strings match exactly [5].

Another way to transform the two strings so that the transformed strings will match exactly if the original strings p-matched, is to transform them into restricted growth functions (RGF) [19]. A string is transformed into a RGF by replacing all occurrences of the first occurring character with 1, the second one with 2 and so on. We call the resulting string the RGF string. For example the string 'aabac' is transformed into 1-1-2-1-3. The properties of restricted growth functions have been studied previously. In [19] it is shown that there are b_n different RGFs of length n where b_n is the n:th Bell number. Kreher and Stinson [19] also give an algorithm for ranking RGFs.

To classify the repetitiveness of a pattern we introduce the concept of q-repetitiveness. A pattern is q-repetitive if for all substrings of length q there is a character that occurs at least twice in the substring. Thus the pattern “aaaa” is 2-repetitive while the pattern “aabb” is 3-repetitive but not 2-repetitive because the substring “ab” contains no repetition. Similarly a two-dimensional pattern is q-repetitive if for all substrings of size $q \times q$ there is a character that occurs at least twice in the substring.

3 Earlier Solutions

3.1 One-Dimensional Algorithms

In her original paper Baker [5] gave a suffix tree based algorithm for finding parameterized matches. The algorithm first preprocesses both the text and the pattern by transforming them into predecessor strings. After this preprocessing the problem can almost be solved by conventional exact string matching algorithms. The only remaining problem is that if we are considering a window on the text, the predecessor pointers might point to positions outside the window. Baker proposed modifications to the suffix tree construction algorithm that take care of this problem. The construction of the suffix tree was further improved by Kosaraju [18] and Cole and Hariharan [9].

In addition Baker [6] has proposed a Boyer-Moore based algorithm. Also this algorithm preprocesses both the text and the pattern into predecessor strings. Baker then uses a modification of the TurboBM algorithm [10] for finding the p-matches. The algorithm has a good worst case performance but because of the preprocessing the sublinearity of the Boyer-Moore type algorithms is lost unless several searches are made with the same text.

Amir et al. [2] have proposed an algorithm for the p-matching problem based on the Knuth-Morris-Pratt algorithm [17] for standard string matching. They also prove that their algorithm is optimal if the alphabet is unbounded.

3.2 Two-Dimensional Algorithms

The two-dimensional parameterized matching problem was first considered by Amir et al. [1] in the context of function matching. They give an algorithm that preprocesses the text into a predecessor representation suitable for two-dimensional strings and then apply a conventional two-dimensional algorithm. Hazay et al. [14] give another algorithm for two-dimensional parameterized matching that is based on the "duel-and-sweep" paradigm. Both of these algorithms are quite complicated and neither one of them has been implemented as far as we know.

4 Our Algorithms

In this section we develop Boyer-Moore type algorithms that do not preprocess the text and thus the preprocessing does not prevent average case sublinearity. Our algorithms use q-grams to achieve longer shifts. The use of q-grams is a well known technique to improve the efficiency of the exact Boyer-Moore-Horspool (BMH) algorithm [15] in case of small alphabets, see e.g. [3].

In this section we first describe the one-dimensional algorithm with several variations and then we discuss the two-dimensional algorithm.

4.1 Three One-Dimensional Algorithms

Our one-dimensional algorithms are derived from the Horspool variant of the Boyer-Moore algorithm. In the BMH algorithm the text is processed in windows of size m. The last character of the window is read first. If it does not match the last character of the pattern the window is shifted based on it. Otherwise the window is checked for a match after which a shift is made. In the parameterized matching problem the last character alone never tells that there cannot be a match and even the last two characters usually do not indicate that the window cannot match the pattern. Therefore we form a q-gram of the last q characters of the window and make the shift based on it.

The preprocessing phase of the BMH algorithm constructs the shift table which is consulted in the matching phase to find out the length of the shift based on the last character of the text window. The shift is calculated so that after the shift the last character of the previous window will be aligned with the last occurrence of that character in the pattern.

In the parameterized matching problem the shifts are made based on the last q-gram of the window and we wish to make a shift that aligns it with the last q-gram of the pattern that p-matches it. As described in Section 2 two strings p-match if their predecessor strings match or equally if their RGF strings match. Thus we wish to index the table with the predecessor or RGF strings. An obvious solution is using the rank of the RGF strings as indexes. We call this algorithm Parameterized Boyer-Moore-Horspool with RGF or PBMH-RGF for short.

The problem with this approach is that calculating the rank of an RGF takes quite a lot of time and this needs to be done for each inspected window. Another alternative for calculating the indexes is to transform the q-grams into predecessor strings and then reserving enough bits for each character of the predecessor strings in the index. The first character of the predecessor string is always 0 so we need not reserve any space for it. The second character is either 0 or 1 because the character in the original string is either the same as the first or not. The third character is 0, 1 or 2 with similar reasoning. This means that the last bit of the index is reserved for the second character of the predecessor strings, the next two bits are reserved for the third character, and so on. We call this algorithm Fast Parameterized Boyer-Moore-Horspool or FPBMH for short. This approach wastes some space but the indexes are much faster to calculate. The RGF approach needs a table of size b_q where b_q is the q:th Bell number while the FPBMH algorithms needs a table of size 2^s where $s = \sum_{i=2}^{q} \lceil \log_2 i \rceil$. Table 1 shows the number of entries in the shift table for both approaches for different values of q.

In a random text the distribution of the predecessor strings is very steep. The most common predecessor string of length q, 0^q, has a high probability if the alphabet is reasonably large while the least common predecessor string, 01^{q-1}, has a probability close to 0. This means that we might need to use quite large q-grams which is a problem for FPBMH. On the other hand hashing the q-grams cleverly might let us use even larger q-grams than the PBMH-RGF algorithm can

Table 1. The number of entries in the shift table for PBMH-RGF, FPBMH and PBMH-Hash for various values of q

Algorithm	$q = 2$	$q = 3$	$q = 4$	$q = 5$	$q = 6$	$q = 7$	$q = 8$	$q = 9$	$q = 10$
PBMH-RGF	2	5	15	52	203	877	4,140	21,147	115,975
FPBMH	2	8	32	256	2,048	16,384	131,072	2,097,152	33,554,432
PBMH-Hash	2	4	7	11	16	22	29	37	46

handle. We tried hashing the q-grams by transforming them first to predecessor strings and then adding up all the positions of the predecessor string. With this hashing scheme the most common q-gram is the only one hashed to 0 and thus the hashing might even out the distribution of the q-grams. This modification of the algorithm called PBMH-Hash needs a table of size $q(q-1)/2+1$. Table 1 includes the space requirement for this approach also.

4.2 A Two-Dimensional Algorithm

The two-dimensional algorithm is based on the two-dimensional string matching algorithm by Tarhio [20] which is an extension of the BMH algorithm. In the algorithm by Tarhio the text is divided into $\lceil (n-m)/m \rceil + 1$ strips each of which has m columns. Each strip is then searched for an occurrence with a BMH like algorithm and each potential match is verified with the trivial algorithm.

In each position the character at the lower righthand corner is investigated. If this character occurs in the lowest row of the pattern, there is a potential match which has to be verified. These are found with the help of two tables, M and N. $M[x]$ is the column where the character x occurs first in the lowest row of the pattern and N links the occurrences of x in the lowest row of the pattern. The pattern is shifted down the strip with another table D. $D[x]$ is the occurrence of x that is closest to the lowest row of the pattern but not in the last row. If x does not appear in the pattern, $D[x]$ is m.

The algorithm can be modified to read several characters and calculate the shifts based on all these characters. If we read $q \times q$ characters (a two-dimensional q-gram), the text will then be divided into $\lceil (n-m)/(m-q+1) \rceil + 1$ strips each containing $m-q+1$ columns.

This algorithm which uses q-grams can fairly easily be extended to parameterized matching in a similar fashion as the BMH algorithm was extended for one-dimensional parameterized matching. The resulting algorithm proceeds exactly like the algorithm by Tarhio but the read q-grams are transformed into predecessor strings and these are then used to index the tables. As with the one-dimensional case, there are several ways to transform the predecessor strings into indexes. We implemented the transformation the same way as in the FPBMH algorithm.

5 Analysis

We first analyze the worst and average case complexity of the one-dimensional algorithms and then turn to the two-dimensional case. When analyzing the average case complexity we assume the standard random string model where each character of the text is chosen independently and uniformly.

5.1 The One-Dimensional Algorithms

The preprocessing phase of the algorithms consists of initializing the shift table which takes time proportional to the number of entries in the table. In addition to preprocess the pattern we need to keep track of where the different symbols of the alphabet occurred previously and thus the preprocessing of the q-grams of the pattern takes $O(\sigma + mq)$ time where σ is the size of the alphabet. As stated earlier the number of entries in the shift table is b_q for PBMH-RGF, 2^s for FPBMH and $q(q-1)/2+1$ for PBMH-Hash where b_q is the q:th Bell number and $s = \sum_{i=2}^{q} \lceil \log_2 i \rceil$. Therefore the preprocessing phases of PBMH-RGF, FPBMH and PBMH-Hash have time complexities $O(b_q + \sigma + mq)$, $O(q^{q-1} + \sigma + mq)$, and $O(q^2 + \sigma + mq)$ respectively.

The matching phases of PBMH-RGF and FPBMH algorithms have the same time complexities. The only difference in the algorithms is in handling of the q-grams but both algorithms do this in $O(q)$ time and thus the resulting complexities will be the same. The hashing in the PBMH-Hash algorithm slightly changes the time complexity of the algorithm but the difference is negligible.

In the worst case the one-dimensional algorithms find a match in each position. This means that for each window the whole window is read and compared to the pattern so the worst case complexity of the algorithms is $O(nm)$.

Let us then analyze the average case complexity. In order to do that we need to consider the probability distribution of the different predecessor strings corresponding to random q-grams. Let σ denote the size of the alphabet and let z be the number of zeros in the given predecessor string. Each of the zeros presents a different character in the original string and each non-zero element of the predecessor string is defined by the zeros. Then the probability that the given predecessor string matches a random string is:

$$P(z, q) = \frac{\sigma!}{\sigma^q \cdot (\sigma - z)!}$$

The probability of a window to be checked is the probability that the last q-gram of the window p-matches the last (or $m-q$:th) q-gram of the pattern. Thus the expected number of checked windows is

$$C = (n - m + 1) \cdot P(z_{m-q}, q)$$

where z_{m-q} is the number of zeros in the last q-gram of the pattern. Now if we can choose q so that $z_{m-q} < q$ the probability $P(z_{m-q}, q)$ is low enough and there are only a few checked windows so the scanning time will dominate.

Let us now turn to analyzing the scanning time. We estimate the expected length of shift in the algorithm with

$$S \geq (m - q + 1) \cdot (1 - P(z_{\max}, q))^{m-q} + \sum_{i=1}^{m-q} i \cdot P(z_{m-q-i}, q) \cdot (1 - P(z_{\max}, q))^{i-1}$$

where $z_{\max}$ is the maximum number of zeros in the q-grams of the pattern. Note that this estimate for S is not quite accurate because the consecutive overlapping q-grams of the pattern are not independent. However the difference from the accurate value is insignificant.

If we now choose q to be the smallest q such that the pattern is q-repetitive, the probability $P(z_{\max}, q)$ will be low enough. Then the expected length of shift approaches the value $m-q+1$ so on average $O((qn)/(m-q+1))$ characters are read. Furthermore if $q<(m+1)/2$ the algorithm is sublinear on average.

Note that all patterns are not q-repetitive for any $q<(m+1)/2$ and in these cases we cannot guarantee the sublinearity of the algorithm. However parameterized matching is most often applied to searching for repetitive patterns so in most practical cases the sublinearity can be guaranteed.

The above analysis holds also if we have both a fixed and a parameterized alphabet. In fact the fixed alphabet makes the problem easier. In this case a sufficient condition for sublinearity is that each q-gram of the pattern contains repetition or at least one character from the fixed alphabet.

5.2 The Two-Dimensional Algorithm

Let us first consider the complexity of the preprocessing phase. The two-dimensional algorithm uses the strategy of the FPBMH algorithm when calculating the indexes of the shift table. Thus the number of entries in the shift table is 2^s where $s=\sum_{i=2}^{q^2}\lceil\log_2 i\rceil$. As with the one-dimensional algorithms we also need to keep track of the previous occurrences of the alphabet symbols and thus a table of size σ is needed for that. The time complexity of the preprocessing phase of the two-dimensional algorithms is thus $O((q^2)^{q^2-1}+\sigma+m^2q^2)$.

The worst case for the two-dimensional algorithm occurs when all positions of the text match. The worst case time complexity is then clearly $O(n^2m^2)$.

For the average case complexity we will need to estimate the number of checked windows. There are a total of $(n-m+1)^2$ windows so on average

$$C=(n-m+1)^2\cdot P(z_{m-q,m-q},q^2)$$

of them are checked where $z_{m-q,m-q}$ is the number of zeros in the q-gram of the pattern that starts at position $(m-q,m-q)$. If $z_{m-q,m-q}<q^2$, $P(z_{m-q,m-q},q^2)$ is low enough and there will only be a few checked windows. Therefore the scanning time will dominate.

Let us next consider the expected length of shift, S. The estimate is very similar to the one-dimensional case:

$$\begin{aligned}S\geq\ &(m-q+1)\cdot(1-P(z_{\max},q^2))^{(m-q)\cdot(m-q+1)}\\&+\sum_{i=1}^{m-q}i\cdot P(\min_{1\leq x\leq m-q+1}z_{m-q-i,x},q^2)\cdot(1-P(z_{\max},q^2))^{(i-1)\cdot(m-q+1)}\end{aligned}$$

where $z_{\max}$ is the maximum number of zeroes in the predecessor strings corresponding to any of the q-grams of the pattern. As with the one-dimensional case, if we now choose q to be the smallest value such that the pattern is q-repetitive, $z_{\max}<q^2$, $P(z_{\max},q^2)$ is low enough and S approaches $m-q+1$. So on average $O((n-m)/(m-q+1)\cdot q^2n/(m-q+1))=O(q^2n^2/(m-q)^2)$ characters are read. Therefore if the pattern is q-repetitive for a suitable q then the algorithm will be sublinear on average. Again some patterns are not q-repetitive for a suitable q and in these cases the sublinearity of the algorithm cannot be guaranteed.

6 Experimental Results

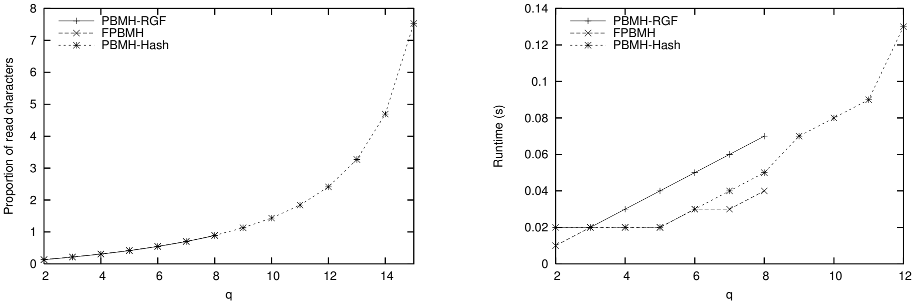
The analysis predicts that the value of q should be chosen to be the smallest q such that the pattern is q-repetitive. To validate this we ran our algorithms with several patterns and a randomly generated text with alphabet size 256. Figures 1, 2 and 3 show the proportion of read characters and the runtime for some patterns. The proportion of read characters is calculated as lookups divided by the length of the text so for a sublinear algorithm this value is less than one. All these tests were run on a computer with a 1.0 GHz AMD Athlon processor, 512 MB of memory and 256 kB on-chip cache. The computer was running Linux 2.4.22. The algorithms were written in C and compiled with gcc 3.2.2.

Figure 1 shows that choosing a larger q with a highly repetitive pattern does not make the algorithms perform faster. Using 2-grams already guarantees long enough shifts and thus assembling larger q-grams just wastes time. Figure 2 presents a completely different scenario. Here the pattern is not q-repetitive for any q and as can be seen we cannot choose large enough q to guarantee the sublinearity of the algorithms. In Figure 3 the situation is something in between. The pattern is 3-repetitive but not 2-repetitive. As can be seen the value $q = 3$ is optimal in this situation and using larger q-grams only makes the algorithms do more work.

Table 2 shows a runtime comparison of our one-dimensional algorithms and a Boyer-Moore-Horspool algorithm (PBMH) which we use as a reference method. The PBMH algorithm preprocesses the text into a predecessor string and then matches the pattern against the text. The preprocessing of the text is included in the figures but the preprocessing of the pattern is not. As can be seen our algorithms are faster when the pattern contains a substantial amount of repetition. However when the pattern contains no repetition the algorithm that preprocesses the text is faster.

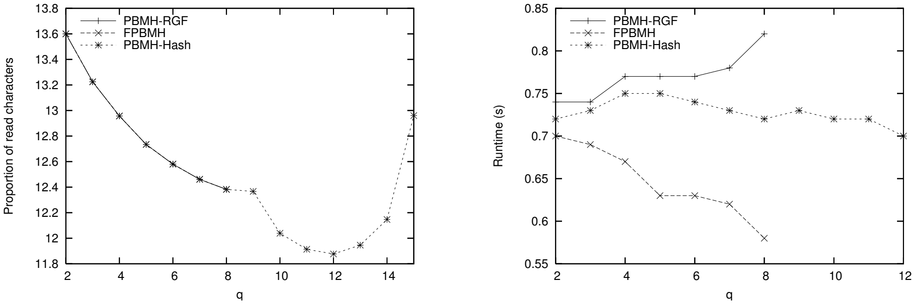
To demonstrate the performance of our algorithms in a more realistic scenario we ran some tests with DNA data. The text was a chromosome from the fruit fly genome (20 MB) and the patterns were chosen randomly from the text. Our algorithms were fastest when using 6-grams. Figure 4 shows the averages over 50 runs. As can be seen our algorithms have characteristics typical to Boyer-Moore based algorithms. With longer patterns the shifts get longer and thus the algorithms are faster.

We ran also some tests with the two-dimensional algorithm. We used two different texts. One was a randomly generated text where the characters were drawn from an alphabet of 256 characters. The other one was a picture of a map from the photo archive of Gimp-Savvy.com [12]. We examined the proportion of read characters for three different patterns of size 8×8. The first one contained repetitions of one character. The second pattern contained no repetitions and the third contained a map symbol which contains some repetition. Table 3 shows the results of the tests run with the two-dimensional algorithm using 3-grams. As can be seen the algorithm performs well when the text or the pattern contains repetitions.



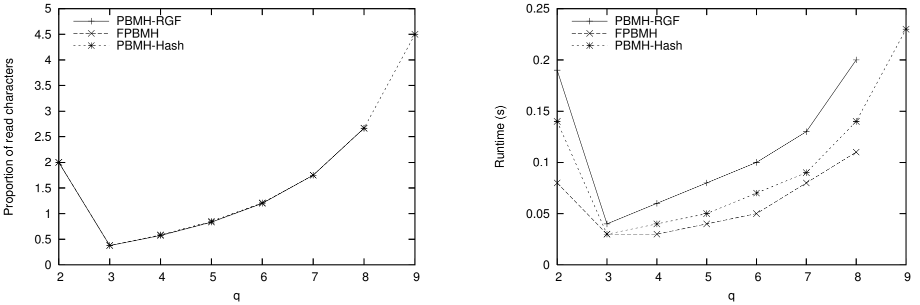
(a) (b)

Fig. 1. Proportion of read characters (a) and runtime (b) for the pattern “aaaaaaaaaaaaaaaa” in a random text



(a) (b)

Fig. 2. Proportion of read characters (a) and runtime (b) for the pattern “qwertyuiopsadfgh” in a random text

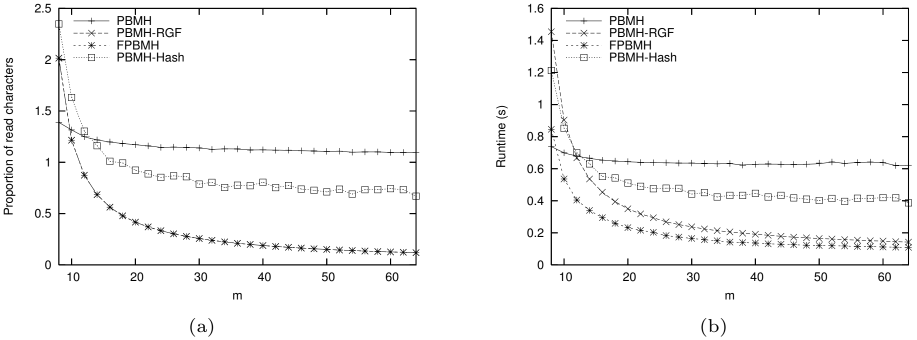


(a) (b)

Fig. 3. Proportion of read characters (a) and runtime (b) for the pattern “aassddssaa” in a random text

Table 2. Runtime comparison of the one-dimensional algorithms in a random text

Algorithm	P=aaaaaaaaaaaaaaaa	P=qwertyuiopasdfgh	P=aassddssaa
PBMH	0.08 s	0.29 s	0.08 s
PBMH-RGF	0.02 s	0.74 s	0.04 s
FPBMH	0.01 s	0.58 s	0.03 s
PBMH-Hash	0.02 s	0.70 s	0.03 s



(a) (b)

Fig. 4. Proportion of read characters (a) and runtime (b) for a text of DNA data and patterns of varying length. Our algorithms used 6-grams in these tests.

Table 3. Proportion of read characters for two different texts and several different patterns. All the patterns are of size 8 × 8.

Text	Single-character pattern	Pattern with no repetitions	Pattern with repetitions
Random	0.25	7.90	0.25
Map	1.14	0.25	0.33

7 Conclusions and Further Work

We have presented practical Boyer-Moore type algorithms for one and two-dimensional parameterized matching. We have showed that these algorithms are sublinear on average for q-repetitive patterns and confirmed this analysis with experiments.

Parallel to our work and independently of us Fredriksson and Mozgovoy [11] have also developed sublinear algorithms for one-dimensional parameterized matching. Their algorithms are based on the shift-or [4] and backward DAWG matching (BDM) [10] algorithms. As further work we need to compare our algorithms also with these algorithms.

The analysis assumes the random string model which might not be applicable especially with two-dimensional texts which are typically images. It is very characteristic of such data that the probability of two nearby characters being the same is very high. We need to further investigate these typical characteristics of texts and analyze our algorithms in this context. We also need to make further tests on real data to confirm the usefulness of our algorithms.

References

1. Amir, A., Aumann, Y., Cole, R., Lewenstein, M., Porat, E.: Function matching: algorithms, applications and a lower bound. In: Proceedings of ICALP. (2003) 929–942
2. Amir, A., Farach, M., Muthukrishnan, S.: Alphabet dependence in parameterized matching. Information Processing Letters **49**(3) (1994) 111–115
3. Baeza-Yates, R.: Improved string searching. Software – Practice and Experience **19**(3) (1989) 257–271
4. Baeza-Yates, R., Gonnet, G.: A new approach to text searching. Communications of ACM **35**(10) (1992) 74–82
5. Baker, B.S.: A theory of parameterized pattern matching: algorithms and applications. In: Proceedings of the 25th ACM Symposium on the Theory of Computation. (1993) 71–80
6. Baker, B.S.: Parameterized pattern matching by Boyer-Moore-type algorithms. In: Proceedings of the 6th Annual ACM Symposium on Theory of Computing. (1995) 541–550
7. Baker, B.S.: Parameterized diff. In: Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms. (1999) 854–855
8. Boyer, R., Moore, J.: A fast string searching algorithm. Communications of the ACM **20**(10) (1977) 762–772
9. Cole, R., Hariharan, R.: Faster suffix tree construction with missing suffix links. In: Proceedings of the 32nd ACM Symposium on the Theory of Computation (STOC). (2000) 407–415
10. Crochemore, M., Lecroq, T., Czumaj, A., Gąsieniec, L., Jarominek, S., Plandowski, W.: Speeding up two string-matching algorithms. In: 9th Annual Symposium on Theoretical Aspects of Computer Science (STACS'92). Volume 577 of LNCS. (1992) 589–600
11. Fredriksson, K., Mozgovoy, M.: Sublinear parameterized single and multiple string matching. Technical Report A-2006-2, Department of Computer Science, University of Joensuu (2006)
12. Gimp-Savvy.com: Copyright-free photo archive: Public domain photos and images. http://gimp-savvy.com/PHOTO-ARCHIVE/ (2000)
13. Hazay, C., Lewenstein, M., Sokol, D.: Approximate parameterized matching. In: Proceedings of the 12th European Symposium on Algorithms (ESA). (2004) 414–425
14. Hazay, C., Lewenstein, M., Tsur, D.: Two dimensional parameterized matching. In: Proceedings of the 16th Annual Symposium on Combinatorial Pattern Matching (CPM'05). Volume 3537 of LNCS. (2005) 266–279
15. Horspool, N.: Practical fast searching in strings. Software – Practise and Experience **10** (1980) 501–506
16. Idury, R.M., Schäffer, A.A.: Multiple matching of parameterized patterns. Theorethical Computer Science **154**(2) (1996) 203–224
17. Knuth, D.E., Morris, J.H., Pratt, V.R.: Fast pattern matching in strings. SIAM Journal of Computing **6** (1977) 323–350
18. Kosaraju, S.R.: Faster algorithms for the construction of parameterized suffix trees. In: Proceedings of the 36th Symposium on Foundation of Computer Science (FOCS). (1995) 631–637
19. Kreher, D.L., Stinson, D.R.: Combinatorial Algorithms: Generation, Enumeration and Search. CRC Press (1999)
20. Tarhio, J.: A sublinear algorithm for two dimensional string matching. Pattern Recognition Letters **17** (1996) 833–838

Approximate Matching in Weighted Sequences

Amihood Amir[1], Costas Iliopoulos[2], Oren Kapah[3], and Ely Porat[3]

[1] Department of Computer Science, Bar-Ilan University, Ramat-Gan 52900, Israel
and College of Computing, Georgia Tech, Atlanta, GA 30332-0280
+972 3 531-8770
amir@cs.biu.ac.il

[2] Department of Computer Science, King's College London,
Strand, London WC2R 2LS, United Kingdom
(+44) 20 7848 2809
csi@dcs.kcl.ac.uk

[3] Department of Computer Science, Bar-Ilan University, 52900 Ramat-Gan, Israel
(972-3)531-7620
{kapaho, porately}@cs.biu.ac.il

Abstract. Weighted sequences have been recently introduced as a tool to handle a set of sequences that are not identical but have many local similarities. The weighted sequence is a "statistical image" of this set, where the probability of every symbol's occurrence at every text location is given.

We address the problem of approximately matching a pattern in such a weighted sequence. The pattern is a given string and we seek all locations in the set where the pattern occurs with a high enough probability. We define the notion of Hamming distance and edit distance in weighted sequences and give efficient algorithms for computing them. We compute two versions of the Hamming distance in time $O(n\sqrt{m}\log m)$, where n is the length of the weighted text and m is the pattern length. The edit distance is computed in time $O(nm)$ and $O(nm^2)$, depending on the edit distance definition used. Unfortunately, due to space considerations, the edit distance details are left to the journal version.

We also define the notion of weighted matching in infinite alphabets and show that exact weighted matching can be computed in time $O(s\log^2 s)$, where s is the number of text symbols having non-zero probability. The weighted Hamming distance over infinite alphabets can be computed in time $\min(O(kn\sqrt{s}+s^{3/2}\log^2 s), O(s^{4/3}m^{1/3}\log s))$.

1 Introduction

Recently, a new pattern matching paradigm was introduced. *Weighted sequences* was initially motivated by trying to reconcile the differences between the genomic sequences of different individuals. The great effort of the *genome project*, that is now winding down, has been to construct a "consensus sequence" of the human genome. Individual human genomes are very similar therefore such a "generic" consensus sequence can be achieved. Nevertheless, clearly no two individuals have the same DNA sequence. Several methods have been proposed for dealing with this polymorphism. One proposed idea is that of the *Position Weight Matrix* (PWM for short) [14]. The PWM is a more precise encoding that takes into

M. Lewenstein and G. Valiente (Eds.): CPM 2006, LNCS 4009, pp. 365–376, 2006.

account the relative frequency of each nucleotide. The *weighted sequence* of length m (the PWM of a set of strings of length m) is a $|\Sigma| \times m$ matrix that reports the frequency of each symbol in finite alphabet Σ (nucleotide, in the genomic setting) for every possible location.

Originally, PWM sequences were used for relatively short sequences, e.g. binding sites or sequences resulting of multiple alignments. Iliopoulos et al. [11, 9, 3, 10] considered building very large Position Weight Matrices that correspond, for example, to complete chromosome sequences that have been obtained using a whole-genome shotgun strategy [15]. By keeping all the information the whole-genome shotgun produces, it should be possible to ferret out information that has been previously undetected after being faded during the consensus step. This concept is true for other applications where local similarities are thus encoded. It is therefore necessary to develop adequate algorithms on weighted sequences, that can be an aid to the application researchers for solving various problems they are liable to encounter.

In this paper we develop algorithms for approximate search on weighted sequence. We handle the case of *Hamming distance* (*edit distance* is deleted from this paper due to page limits). In approximate matching it is assumed that there are errors in the data and matches with a small number of errors are sought. In classical pattern matching it usually does not matter if the assumption is that the text is error-free and the errors are all in the pattern or vice-versa, since there is a symmetry in most edit operations. It turns out that in weighted matching (as in, e.g., hypertext matching [2]) there is a distinction between cases. Assuming mismatch errors in the weighted text, there is always an approximate match at every location, depending on the number of errors. However, assuming a "clean" text and mismatch errors in the pattern, there may be locations where a match can not ever be found, no matter what the cost. We show efficient algorithms for computing the Hamming distance at every location for both models.

The contributions of this paper are as follows.

1. We formalize two possible definitions for *Hamming distance* in weighted sequences. Since we are dealing with a new paradigm, this formalism is very important. Special care should be given for a definition that captures natural traits that will be useful in applications.
2. We provide the first efficient algorithms for computing the Hamming distance on weighted sequences. Our algorithms run in time $O(n\sqrt{n \log m})$, where the length of the weighted text is $|\Sigma|n$ and the length of the pattern is m. This algorithm is achieved via a non-trivial bounded divide-and-conquer algorithm, coupled with some insights we prove on weighted sequences.
3. We formalize two possible definitions for *edit distance* in weighted sequences, and provide dynamic programming algorithms for the edit distance. One definition leads to a $O(nm)$, algorithm and the other has a $O(nm^2)$ algorithm. This treatment is left for the journal version due to page limitations.
4. We define weighted matching over infinite alphabets and provide the first efficient algorithm to solve the problem. Our algorithm runs in time $O(s \log^2 s)$ and uses superimposed coding techniques.

5. We provide the first known efficient algorithms for computing Hamming distance of a pattern in a weighted text over infinite alphabet. Our algorithm runs in time $\min(O(kn\sqrt{s} + s^{3/2}\log^2 s), O(s^{4/3}m^{1/3}\log s))$.

2 Preliminaries

Definition 1. *A* weighted sequence $T = t_0, ..., t_n$ *over alphabet* Σ *is a sequence of sets* t_i, $i = 0, ..., n$. *Every* t_i *is a set of pairs* $(s_j, \pi_i(s_j))$, *where* $s_j \in \Sigma$ *and* $\pi_i(s_j)$ *is the probability of having symbol* s_j *in location* i. *Formally,*

$$t_i = \{(s_i, \pi_i(s_j)) \mid s_j \neq s_\ell \text{ for } j \neq \ell, \text{ and } \sum_j \pi_i(s_j) = 1\}.$$

For a finite alphabet $\Sigma = \{a_1, ..., a_{|\Sigma|}\}$ we can view a weighted sequence as a $|\Sigma| \times n$ matrix T of numbers in $[0,1]$, where $T[j,i] = \pi_i(a_j)$. For the rest of this paper we assume a finite fixed alphabet Σ.

Definition 2. $P = p_0, ..., p_m$ *is a* solid sequence *over alphabet* Σ *if* $p_i \in \Sigma$, $i = 0, ..., m$.

We say that solid pattern P *(or simply* pattern P) occurs in location i of weighted text T with probability at least α *if* $\prod_{j=0}^{m} \pi_j(p_j) \geq \alpha$.

Definition 3. *The* exact weighted matching problem *is defined as follows:*

INPUT: Weighted text T *over alphabet* Σ, *solid pattern* P *over alphabet* Σ, *and probability* $\alpha \in [0,1]$.
OUTPUT: All locations i *in* T *where pattern* P *occurs with probability at least* α.

Using convolutions, as introduced by Fischer and Paterson [8], as well as the observation that Solid pattern P occurs in location i of weighted text T with probability at least α if $\sum_{j=0}^{m} \log \pi_j(p_j) \geq \log \alpha$. we can efficiently solve the exact weighted matching problem in time $O(|\Sigma| n \log m) = O(n \log m)$. The idea is to use the Fast-Fourier-Transform (FFT) [6] to compute the sum of the log probabilities for every pattern symbol separately. This can be done in time $O(n \log m)$, in a computational model with word size $O(\log m)$.

We are now ready for the Hamming distance in weighted sequences problem.

3 Hamming Distance – Error in Text

Computing the Hamming distance between two (solid) strings assumes that a number of symbols were replaced. The Hamming distance is the number of these replaced symbols. In the case of weighted subsequences it makes a difference where these symbols were replaced. The simpler case, which we consider in this section, assumes replacement in the text. The assumption is that some text symbols are erroneous and, in fact, there should have been a probability 1 for the symbol that happens to match the pattern, rather than the probabilities that appear in the text.

Note that by this definition, allowing enough mismatches can guarantee a match at every location, no matter how close to 1 we choose α.

Definition 4. *The* Weighted Hamming Distance with Mismatches in the Text problem *is the following:*

INPUT: Weighted text T over alphabet Σ, solid pattern P over alphabet Σ, and probability $\alpha \in [0, 1]$.
OUTPUT: For every location i in T, the minimum k such that if k text probabilities were changed to 1 then pattern P would occur at location i with probability at least α.

There does not seem to be a natural way to use the powerful constraint that the numbers in the weighted text are probabilities. However, it seems like we can solve the problem without it. We reduce the weighted Hamming distance with mismatches in the text problem to the *minimum ignored mask bits problem.* The idea is to consider a text whose elements are non-positive numbers, and a pattern which is a mask, i.e. its symbols are 0's and 1's. Suppose we are interested in finding out, for each text location i, the sum of the text numbers that are aligned with 1's in the pattern.

Clearly this is a simple convolution of the pattern and text. However, we add a complication, we also have a non-positive integer α and for every text location i we seek the smallest number of mask bits that, if set to 0, would make the sum of text numbers that are aligned with (the remaining) 1's in the pattern, be no less than α.

We formally define the problem.

Definition 5. *The* Minimum Ignored Mask Bits problem *is the following:*
INPUT: Solid text T of length $n + 1$ whose elements are non-positive integers, solid pattern P of length $m + 1$ over alphabet $\{0, 1\}$, and integer $\alpha \leq 0$.
OUTPUT: For every location i in T, the minimum k such that if k pattern bits are changed from 1 to 0, and M' is the pattern resulting from those k changes, then $\sum_{j=0}^{m} T[i + j]M[j] \geq \alpha$.

Claim. The weighted Hamming distance with mismatches in the text problem is linearly reducible to the minimum ignored mask bits problem.

Proof: Given weighted text T in matrix format, where the value in $T[i, j]$ is $\log \pi_j(s_i)$, let solid text T' be a linear listing of matrix T in column-major order, i.e. $T' = T[1, 0], T[2, 0], T[3, 0], ..., T[|\Sigma|, 0],$
$T[1, 1], T[2, 1], T[3, 1], ..., T[|\Sigma|, 1], ...,$
$T[1, n], T[2, n], T[3, n], ..., T[|\Sigma|, n]$. Let M be a string of length $|\Sigma|(m + 1)$ over $\{0, 1\}$ where M is the concatenation of strings $B(p_0), B(p_1), ...B(p_m)$. $B(a)$ is defines as follows. Let $a = s_\ell$, where $\Sigma = \{s_1, s_2, ..., s_{|\Sigma|}\}$. Then $B(a)$ is a bit string of length Σ, where the ℓ-th element is 1 and all other elements are 0.

Example: If $\Sigma = \{A, B, C, D\}$ and $P = BBAD$, then $M = 0100\,0100\,1000\,0001$.

Clearly, the reduction is linear. It is also clear that turning a 1 bit in the mask M to 0, is equivalent to changing the probability in the text position corresponding to it to 1. Thus a solution to the minimum ignored mask bits problem will provide the solution to the weighted Hamming distance with mismatches in the text problem. □

Algorithm's Idea
We consider two limited cases and show an easy efficient solution for each of them. Subsequently, we use a bounded divide-and-conquer strategy, that splits a general input into the two straightforward cases, and thus solves each separately.

The first special case is one where the domain of numbers appearing in the text is bounded, i.e. there are only r different numbers that can appear as text elements.

Since we are interested in finding the smallest number k of mask 1 bits that, when turned to 0 will make the sum greater than α, and since **all numbers are non-positive**, the following observation is crucial to the algorithm:

Observation 1. *For any location i where $\sum_{j=1}^{r} S_{i,j} < \alpha$, the solution to the minimum ignored mask bits problem can be found by sequentially adding numbers that participate in the sum starting from the ones that contribute least to decreasing it, i.e. the largest (n_1). Stop adding them when the remaining sum is no longer less than α.*

This elimination would normally require $O(m)$ work per location. However, since there are only r different values, and we know how many instances of each value participate in the sum at location i ($S_{i,j}/n_j$), we can do this in time $O(r)$ per location.

Algorithm's Time: $O(rn \log m)$
A second special case we consider is when there is no bound on the number of different text elements, but we do know that for every text substring of length m there are at most r elements greater than α. This means that for location i, there is no point in even considering all elements except those r.

Algorithm's Time: $O(nr)$
We are now ready to present our divide-and-conquer algorithm. Assume first, that the text length is at most $2m$. This is a standard assumption and can be made without loss of generality (see e.g. [1]). We now sort all text elements and split them into r blocks of size at most $2|\Sigma|\frac{m}{r}$ each.

The idea is to use *Algorithm Bounded Alphabet* on the blocks, and *Algorithm Bounded Relevant Numbers* to find the border of the numbers participating in the sum within the block that tips under α. This can be done with a twist on Abrahamson's idea and produce the final algorithm.

Algorithm's Time: The time for this algorithm is $O(rf(m)) + O(m\frac{m}{r})$, where $f(m)$ is the time it takes to compute the block information. We do it by convolutions, as in *Algorithm Bounded Alphabet* so $f(m) = m \log m$. The optimal r is then the one where $r = \sqrt{\frac{m}{\log m}}$.. Thus the algorithm's time is $O(n\sqrt{m \log m})$.

4 Hamming Distance – Error in Pattern

The situation currently addressed is one where the weighted text is assumed to be error-free. The pattern, however, may have replacemet errors, i.e. it is possible that the “true” pattern symbol was replaced by another. This situation is different in a number of ways from the one considered in section 3.

The first difference between the two Hamming distance definitions is the following. The *errors in the text* definition can guarantee a match at every location, no matter how close to 1 we choose α. This is done simply by allowing enough mismatches. At the worst case $m + 1$ mismatches give a probability of 1. This is not the case if we assume errors in the pattern.

We formally define our problem.

Definition 6. *The* Weighted Hamming Distance with Mismatches in the Pattern problem *is the following:*

INPUT: Weighted text T over alphabet Σ, solid pattern P over alphabet Σ, and probability $\alpha \in [0, 1]$.

OUTPUT: For every location i in T, the minimum k such that if k pattern symbols were replaced to create new pattern P' then pattern P' would occur at location i with probability at least α.

The difficulty presented by this definition is that we put the weight of change on the pattern, rather than the text. When a text is changed, by definition 4, that change improves the product of every match that this text location participates in. However, a pattern change may improve the probability in one occurrence but actually make it worse in another.

One may be tempted to say that even when a match is defined on the pattern, we can still tell which probability is *always* best for a given text location - the maximum probability at that location. This maximum probability will actually improve (or at least will never hurt) the probability of any location that it participates in. Perhaps, then, it is possible to sort the text by largest to smallest product improvement. Then it may be possible that the algorithm in section 4 could still be modified to find the largest possible sum of log probabilities and check if it is good enough. The idea would be the following. First make a replacement in the text location that introduces the largest unmatched text probability. Use that replacement only if it is necessary. Proceed by introducing the next largest, etc. The problem is that replacing elements in sorted order from largest to smallest does not guarantee the smallest number of replacements.

The following lemma does guarantee an order of replacement. Assume that the weighted text is given in matrix format T where $T[j, i] = \pi_i(a_j)$, where $\Sigma = \{a_1, ..., a_{|\Sigma|}$, and $\pi_i(a_j)$ is the probability of having symbol a_j at text location i. Let $\max(T[*, i])$ denote the value $\max\{T[j, i] \mid j = 1, ..., |\Sigma|\}$.

Lemma 1. *Consider text element $T[j,i]$ where $\frac{\max(T[*,i])}{T[j,i]}$ is the largest. Let P be a pattern where $a_j \in P$. Then the largest increase in the product of probabilities as a result of a single symbol replacement occurs by replacing every a_j in the pattern that matches text location i by a_ℓ, where $T[\ell,i] \geq T[j,i],\ j = 1, ..., |\Sigma|$.*

Proof: Let q be the product of probabilities at location i. Assume that the pattern has a_j at location $i+\ell$, $\ell \leq m$, but that symbol a_h has the largest text probability at location $i+\ell$. Then replacing a_j by a_h would cause the product of the probabilities at location i to be $(q/T[j,i+\ell])T[h,i+\ell]$. This means that the largest change will occur when $\frac{T[j,i+\ell]}{T[h,i+\ell]}$ is largest. □

Conclude: Let $T[j,i]$ be such that $\frac{\max(T[*,i])}{T[j,i]}$ is the largest. If we replace **text location** $[j,i]$ by the value $\max(T[*,i])$, the result will be equivalent to replacing every a_j in the pattern that matches text location i by a_ℓ, where $T[\ell,i] \geq T[j,i],\ j = 1, ..., |\Sigma|$. This leads to the idea that if we replace text elements by descending order of $\frac{\max(T[*,i])}{T[j,i]}$ (where necessary) we will guarantee the minimum number of replacements at every location.

Algorithm's Idea: Sort all $2m|\Sigma|$ text elements in non-increasing order of the ratio $\frac{\max(T[*,i])}{T[j,i]}$. As in section 3, split the text elements into $O(\frac{m}{\sqrt{m\log m}})$ groups of size $O(\sqrt{m\log m})$. For each text location i calculate the probabilities $O(\frac{m}{\sqrt{m\log m}})$ times. In the first time calculate the probability of the pattern in the text without replacements. In the second time calculate the probability of the pattern with replacing every element in the group of highest ratios. In the jth time, calculate the pattern probability with replacing every element in the $j-1$ highest ratio groups.

Each such calculation can be done by FFT in time $O(n\log m)$. In addition, we can calculate by FFT the number of replacements done in each location for the groups involved. Finally, in a manner similar to the one shown in section 3, we can fine tune the exact number of replacements for each text location i in time $O(n\sqrt{m\log m})$.

A detailed description of the algorithm will appear in the journal version.

5 Weighted Matching over Infinite Alphabets

The original motivation of weighted sequence matching was from computational biology, where the alphabets are quite small (size 4 for DNA and RNA, and size 20 for amino acids). Nevertheless, from a conceptual point of view, nothing prohibits the alphabet from being very large, or even infinite. The techniques for weighted matching need to be completely different over infinite alphabets, since we may no longer assume that all symbols appear as inputs. Rather, we only input the symbols whose probability is non-zero.

Our formal definition of weighted matching (Definition 1) did not assume a finite alphabet. We now provide an efficient algorithm for exact weighted matching over infinite alphabets. The key observation for our efficient algorithm utilizes *subset matching*. Subset matching was defined by Cole and Hariharan [4], as a tool to solve the *tree pattern matching problem* [12, 7] but meanwhile has proven to be an interesting problem in and of itself. The input of the problem is a text array of n sets totaling s elements and a pattern array of m sets totaling s' elements. There is a match of the pattern in a text location if every pattern set is a subset of the corresponding text set. Formally,

Definition 7. *The Subset Matching Problem is defined as follows.*
INPUT: Text $T = T_1, T_2, ..., T_n$ *of sets* $T_i \subseteq \Sigma, \quad i = 1, ..., n$ *and pattern* $P = P_1, P_2, ..., P_m$ *of sets* $P_i \subseteq \Sigma, \quad i = 1, ..., m$*, where* Σ *is a given alphabet.*
OUTPUT: All locations $i, \quad 1 \leq i \leq n - m + 1$ *where* $\forall \ell = 1, ..., m, \quad P_\ell \subseteq T_{i+\ell-1}$.

Algorithm's Idea: Observe that in every text location where the pattern appears with non-zero probability, there is a subset-matching of the pattern. The algoritm's main idea is, then, to first find the subset matching of the pattern in the text and then calculate the probabilities of those locations. In order to accomplish that all the non zero probabilities will be mapped to a vector with size linear in the number of non-zeros. This mapping will be done using shifting where each symbol is assigned a different shift. The same shifting will be used in both the text and the pattern, thus wherever there is a *singleton* in the text which aligned with a *singleton* in the pattern in the positions where a subset matching was found it is guaranteed to be be the same character.

```
Algorithm Outline
1. Perform subset matching
2. Linearize the input to a vector of probabilities, and calculate the probability of the pattern
   appearing in each text location
end Algorithm Outline
```

Step 1: Ignore the probabilities and consider only the symbols that have a non zero probability. This results in a set of symbols for each text location. Now run Cole and Hariharan's subset matching algorithm [5].

Time Complexity: $O(s \log s)$ where s is the total number of characters.

Step 2: Create a vector of probabilities from the text. This is done by assigning for each alphabet symbol σ a number that sets the shift of this letter. This means that for each location i where σ has a non zero probability, this probability will appear at the new vector at location $i + \mathit{shift}(\sigma)$. Each vector location where more than one value is assigned is referred to as a **multiple** location and is assigned a 0. Every position where only one value was assigned is referred to as a **singleton** and is assigned the log-probability the symbol assigned to it.

Using the same shift values we create a vector of the same size from the pattern. In the vector representing the pattern each symbol that appears as a

singleton in the pattern is replaced by a 1. Multiples are replaced by a 0. After a convolution of the text vector with the pattern vector each location holds the sum of all the probabilities where a **singleton** in the text was aligned with a **singleton** in the pattern.

Lemma 2. *For the locations where a subset matching is found, each* **singleton** *location in the text vector which is aligned with a* **singleton** *in the pattern vector, contains the probability of the letter which appeared as* **singleton** *in the pattern vector.*

Proof: In a situation where a subset matching occurs, clearly the pattern symbol is shifted to the same location as its equivalent text symbol. In a singleton text matched with a singleton pattern, if there the subset match forces the fact that the text symbol equals the pattern symbol and we do not need to verify it. For the same reason, it is impossible for a pattern multiple to be matched with a text singleton when there is a subset match, since all pattern elements should be matched at least with the appropriate text symbols. □

Corollary 1. *After convolving the text vector with the pattern vector, all the non zero probabilities which appeared as* **singletons** *are calculated for all locations.*

As a result of the corollary we can now zero the calculated probabilities of the text singletons and repeat the process using different shifts until all the non zero probabilities become zero. Our solution is the sum of the results. At the end of this process each location where there is a subset match holds the log of the probability of the pattern appearing at this location.

The question we still need to address is how many such rounds are necessary until all the non zeros probabilities appears as **singletons** at least once.

Lemma 3. $O(\log s)$ *rounds are guaranteed to complete the process in the worst case. The appropriate shift functions can be computed in time* $O(s \log^2 s)$*, where* s *is the number of text symbols with non-zero probabilities.*

Proof: The lemma can be proven using superimposed coding as in Cole and Hariharan [5]. A complete proof will be provided in the journal version. □

Time Complexity: The shift functions can be computed in time $O(s \log^2 s)$. For each shift function we perform a convolution which takes $O(s \log s)$ time. There are $O(\log s)$ such convolutions, thus the total time of this step is $O(s \log^2 s)$.

6 Hamming Distance in Weighted Matching over Infinite Alphabets

We present an algorithm for the case of *errors in the text.* The case of *errors in the pattern* is similar.

The main idea of the algorithm is to combine the algorithm devised for the Hamming distance over finite alphabet with the algorithm devised for the weighted matching over infinite alphabet. The difficulty in applying the shifting technique in the Hamming distance case is that now the property that two aligned singletons must originat from the same character no longer applies. We show two ways of overcoming this problem: one with running time dependent on the number of errors and one with running time independent of the number of errors. Both algorithms use the bounded divide-and-conquer technique. We divide the sorted list of probabilities into blocks. For each text location we calculate the sum of probabilities and the number of matches for each block. Subsequently we add the log probabilities from the largest down until the probability is smaller than the input. The number of errors (the Hamming distance) is the size of the pattern minus the number of matches.

We combine the two algorithms to achieve the minimal running time of the two solutions.

6.1 Algorithm 1

The first algorithm solves the problem in the shifting technique by checking for each block separately if there was a match. This is done for each block by replacing each empty position in the shifted text with don't care and matching the shifted pattern with the new text. If a match exist then the sum of probabilities for this block is correct. If there is no match then we need to use brute force. This means checking each character in this block against the character appearing in the aligned position in the pattern.

```
Algorithm Outline
1. Sort the probabilities in the weighted sequence
2. Divide the sorted list of probabilities into blocks of size $\sqrt{s}$
3. For each block calculate the sum of probabilities (using the shifting technique).
4. For each text position and each block: If there is a subset match use the result
   calculated in the previous stage, else use brute force.
5. For each text position add blocks probabilities from the largest down until the
   sum goes below the input threshold.
6. For each text position add probabilities within the last block until the sum
   goes below the input threshold.
end Algorithm Outline
```

Time: $O(kn\sqrt{s} + s^{3/2}\log^2 s)$

Where k is the average number of blocks per text position, where a match does not exist.

6.2 Algorithm 2

The second algorithm handles the problem in the shifting technique by dividing the symbols into *frequent* and *non-frequent* characters and dealing with each type of characters separately.

Definition 8. *Let P be a pattern of length m. A pattern symbols is* frequent *if it occurs in the pattern at least $m^{2/3}$ times, otherwise it is* rare.

For each block and each *frequent* character, the sum of probabilities is calculated using convolution. Also the number of matches is calculated using convolution where we replace the probabilities in the text with ones. Since there are at most $m^{1/3}$ *frequent* characters and $s^{1/3}$ blocks, the time complexity for the *frequent* characters is $O(s^{4/3}m^{1/3}\log s)$.
For the *non-frequent* characters, brute force is used to calculate the sum of probabilities and the number of matches. Since each *non-frequent* character can appear up to $m^{2/3}$ times in the pattern the time complexity for the *non-frequent* characters is $O(sm^{2/3}) < O(s^{4/3}m^{1/3})$.

```
Algorithm Outline
1. Sort the probabilities in the weighted sequence
2. Divide the sorted list of probabilities into blocks of size $s^{2/3}$
3. For each block calculate the sum of probabilities and count the number of matches of the non-frequent characters using brute force.
4. For each block calculate the sum of probabilities and count the number of matches of the frequent characters using convolution.
5. For each text position add blocks probabilities from the largest down until the sum goes below the input threshold.
6. For each text position add probabilities within the last block until the sum goes below the input threshold.
end Algorithm Outline
```

Time Complexity: $O(s^{4/3}m^{1/3}\log s)$.

6.3 Combining the Algorithms

In order to obtain the minimal running time of both algorithm we start with first algorithm without doing the brute force part and check the average number of blocks per text location where a match was not found. If this number is not too large then we will proceed with the first algorithm and use brute force to calculate the sum of probabilities for these blocks and eventually the Hamming distance. If the number is too large, then we will use the second algorithm.

7 Conclusion and Open Problems

This paper defined the concept of approximate weighted distances, both in terms of Hamming distance and edit distance. We also presented efficient algorithms for these definitions. The algorithms are efficient in the sense that there are no known faster algorithms for edit distance in solid strings (by definition 1) or for finding the masked Hamming distance for solid strings. Further research directions would be to find efficient algorithms for the k-mismatches or k-error problems in weighted sequences.

Acknowledgments. The first author was partially supported by NSF grant CCR-01-04494 and ISF grant 35/05. The last author was partially supported by GIF Young Scientists Program grant 2055-1168.6/2002.

References

1. A. Amir and M. Farach. Efficient 2-dimensional approximate matching of half-rectangular figures. *Information and Computation*, 118(1):1–11, April 1995.
2. A. Amir, N. Lewenstein, and M. Lewenstein. Pattern matching in hypertext. *J. of Algorithms*, 35:82–99, 2000.
3. M. Christodoulakis, C. S. Iliopoulos, L. Mouchard, and K. Tsichlas. Pattern matchnig on weighted sequences. In *Proceedings of the Algorithms and Computational Methods for Biochemical and Evolutionary Networks (CompBioNets).* KCL Publications, 2004.
4. R. Cole and R. Hariharan. Tree pattern matching and subset matching in randomized $o(n \log^3 m)$ time. *Proc. 29th ACM STOC*, pages 66–75, 1997.
5. R. Cole and R. Hariharan. Verifying candidate matches in sparse and wildcard matching. In *Proc. 34st Annual Symposium on the Theory of Computing (STOC)*, pages 592–601, 2002.
6. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press and McGraw-Hill, 1992.
7. M. Dubiner, Z. Galil, and E. Magen. Faster tree pattern matching. *J. ACM*, 41(2):205–213, 1994.
8. M.J. Fischer and M.S. Paterson. String matching and other products. *Complexity of Computation, R.M. Karp (editor), SIAM-AMS Proceedings*, 7:113–125, 1974.
9. C. S. Iliopoulos, L. Mouchard, K. Perdikuri, and A. Tsakalidis. Computing the repetitions in a weighted sequence. In *Proceeding of the Prague Stringology Conference*, pages 91–98, 2003.
10. C. S. Iliopoulos, K. Perdikuri, E. Theodoridis, A. Tsakalidis, and K. Tsichlas. Motif extraction from weighted sequences. In A. Apostolico and M. Melucci, editors, *Proc. 11th Symposium on String Processing and Information Retrieval (SPIRE)*, volume 3246 of *LNCS*, pages 286–297. Springer, 2004.
11. C.S. Iliopoulos, C. Makris, I. Panagis, K. Perdikuri, E. Theodoridis, and A. Tsakalidis. Computing the repetiotions in a weighted sequence using weighted suffix trees. In *European Conference on Computational Biology (ECCB)*, pages 539–540, 2003.
12. S. R. Kosaraju. Efficient tree pattern matching. *Proc. 30th IEEE FOCS*, pages 178–183, 1989.
13. V. I. Levenshtein. Binary codes capable of correcting, deletions, insertions and reversals. *Soviet Phys. Dokl.*, 10:707–710, 1966.
14. J.D. Thompson, D.G. Higgins, and T.J. Gibson. Clustal w: Improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. *Nucleic Acids Research*, 22:4673–4680, 1994.
15. J.C. Venter and Celera Genomics Corporation. The sequence of the human genome. *Science*, 291:1304–1351, 2001.

Algorithms for Finding a Most Similar Subforest

Jesper Jansson[1,*] and Zeshan Peng[2]

[1] Department of Computer Science and Communication Engineering,
Kyushu University, 6-10-1 Hakozaki, Higashi-ku, Fukuoka 812-8581, Japan
[2] Department of Computer Science, The University of Hong Kong,
Pokfulam Road, Hong Kong
jj@tcslab.csce.kyushu-u.ac.jp, zspeng@cs.hku.hk

Abstract. Given an ordered labeled forest F ("the target forest") and an ordered labeled forest G ("the pattern forest"), the *most similar subforest problem* is to find a subforest F of F such that the distance between F and G is minimum over all possible F . This problem generalizes several well-studied problems which have important applications in locating patterns in hierarchical structures such as RNA molecules' secondary structures and XML documents. In this paper, we present efficient algorithms for the most similar subforest problem with forest edit distance for three types of subforests: simple substructures, sibling substructures, and closed subforests.

1 Introduction

An *ordered labeled tree* is a rooted tree in which the left-to-right ordering among nodes is fixed and each node is labeled by a symbol from a given alphabet. An *ordered labeled forest* is a sequence of ordered labeled trees. Ordered labeled trees and forests are useful data structures for hierarchical data representation; for example, XML documents are essentially ordered labeled trees [2] and RNA molecules' secondary structures without pseudoknots can be represented by ordered labeled forests [4, 7, 11] (see Fig. 1(a)–(c) for an example). Below, we refer to ordered labeled trees and ordered labeled forests as *trees* and *forests*, respectively.

In this paper, we study the following problem which we call *the most similar subforest problem*: Given a forest F ("the target forest") and a forest G ("the pattern forest"), find a subforest of F which is the most similar to G. There are many ways to define "subforest" and "most similar"; here, we consider three alternative definitions of "subforest" and show how to solve all the resulting problems efficiently when the *forest edit distance* [13, 16] is used to measure similarity. Our techniques combine and extend the techniques of [4] and [16].

The most similar subforest problem generalizes several other problems. For example, in the well-studied *forest inclusion problem*, the objective is to determine whether a given forest G can be obtained from another given forest F by only deleting nodes from F, and if so, finding the smallest subforest of F in

[*] Supported by JSPS (Japan Society for the Promotion of Science).

M. Lewenstein and G. Valiente (Eds.): CPM 2006, LNCS 4009, pp. 377–388, 2006.

which G is included (this problem and a constrained variant have been studied in, e.g., [15, 8]). However, in case G is *not* included in F, one might still need to find a subforest F' of F such that G is very similar to F', or to measure how far from being included in F the pattern forest G is. This is precisely "the most similar subforest problem". As another example, consider the *string pattern problem*: given two strings S and T, find a most similar (using edit distance) substring of S to T. This problem has many applications to Stringology [3] and Bioinformatics [12]. Since a string can be represented by a tree in which all non-leaf nodes have exactly one child, the string pattern problem is just a special case of the most similar subforest problem. [1]

2 Preliminaries

Throughout this paper, we use the following notation and definitions.

Let F be any given forest. Denote the number of nodes in F by $|F|$, and define $\deg(F)$ (the *degree of* F) as the maximum number of children over all nodes in F, and $dp(F)$ (the *depth of* F) as the number of edges on the longest path from a root node in F to a leaf of F. The set of leaves in F is referred to as $L(F)$. For any node $i \in F$, define $p(i)$ as *the parent of* i, and denote the label of i by $label(i)$. Any $i_1, i_2 \in F$ are *siblings* if they have the same parent; if $i_1 \neq i_2$ also holds then i_1 and i_2 are *proper siblings.* To simplify the presentation, we assume that the roots of the trees in F share an imaginary parent node, denoted by $p(F)$, which is considered to belong to F and which is labeled by a special symbol '$\diamond$'. Define the *key nodes of* F as the set $K(F) = \{p(F) \cup i \mid i \in F \text{ has a left proper sibling}\}$. Clearly, it holds that $|K(F)| \leq |L(F)|$ (Lemma 6 in [16]).

Assume without loss of generality that the nodes of F are *numbered* according to the order in which they are visited by a left-to-right postorder traversal of F. Then, for any $i_1, i_2 \in F$, define $i_1 : i_2$ as the set of nodes whose numbers are greater than or equal to i_1 and less than or equal to i_2. For any siblings i_1 and i_2 with $i_1 \leq i_2$, define $i_1 \cdot\cdot\, i_2$ as the set of nodes consisting of i_1, i_2, and every node which is both a right sibling of i_1 and a left sibling of i_2 (if $i_1 > i_2$, define $i_1 \cdot\cdot\, i_2 = \emptyset$). Finally, for any $i \in F$, refer to the *leftmost* and *rightmost* siblings of i by $b(i)$ and $e(i)$, respectively, and define $m(i)$ as the smallest numbered node in the subtree rooted at i (note that by the left-to-right postordering of the nodes, $m(i)$ will always be the leftmost leaf in this subtree).

Forest edit distance [13, 16]**:** Define the following three *edit operations* on F:

–*Relabel:* Change the label of any node in F.

–*Delete:* Delete any node i from F by making all of i's children (if any) become children of $p(i)$ and then removing node i and the edge between i and $p(i)$.

– *Insert:* Insert a new node with any label into F (the inverse operation of delete).

[1] In fact, the running time of our algorithm for finding a most similar simple substructure in Section 3.1 with parameters $|L(F)| = 1$ and $|L(G)| = 1$ matches the fastest known algorithm for the string pattern problem.

See, e.g., [1,4,5,13,14,16] for examples of these operations. Next, define an *edit mapping* M between two forests F and G as a set of pairs (i,j), where $i \in F$ and $j \in G$, such that for any two pairs $(i_1,j_1),(i_2,j_2) \in M$, the following properties are satisfied: **(1)** $i_1 = i_2$ if and only if $j_1 = j_2$; **(2)** i_1 is an ancestor of i_2 if and only if j_1 is an ancestor of j_2; and **(3)** $i_1 < i_2$ if and only if $j_1 < j_2$. For any $(i,j) \in M$, we say that node i is *linked with* node j in M. Let M be an edit mapping between F and G. Define its *left-linked set* as $M_F = \{i \mid (i,j) \in M\}$ and its *left-unlinked set* as $R_F = F \setminus M_F$, and define its *right-linked set* M_G and *right-unlinked set* R_G analogously. An edit mapping M between F and G uniquely determines a sequence of delete and relabel operations on F and G such that the resulting forests F' and G' are identical. More precisely, every $i{\in}R_F$ means "delete node i from F", every $j{\in}R_G$ means "delete node j from G", and every $(i,j){\in}M$ with $label(i){\neq}label(j)$ means "relabel i with the label of j".

From here on, we assume that the nodes in the input forests F are G are labeled by a fixed alphabet Σ where $\diamond \notin \Sigma$ (recall that the symbol '$\diamond$' is already in use). Moreover, we assume that '$-$' is a special blank symbol not in Σ and that we are given a fixed *distance function* $\gamma : (\Sigma \cup \{\diamond, -\}) \times (\Sigma \cup \{\diamond, -\}) \rightarrow \Re$, where $\Re$ is the set of real numbers and where for any $a,b \in \Sigma$, it holds that $\gamma(a,a){\leq}0$, $\gamma(a,b){\geq}0$ if $a{\neq}b$, $\gamma(a,-){\geq}0$, $\gamma(-,b){\geq}0$, and $\gamma(-,-){>}0$. We also assume that $\gamma(a,\diamond) = 0$, $\gamma(\diamond,b) = 0$, $\gamma(\diamond,\diamond) = 0$, $\gamma(\diamond,-) \geq 0$, $\gamma(-,\diamond) \geq 0$. For any $i \in F$ and $j \in G$, define $f(i) = label(i)$ and $g(j) = label(j)$. Then, for any $i \in F$ and $j \in G$, the *distance* between i and j is defined as $\gamma(i,j){=}\gamma(f(i),g(j))$. Finally, define the *cost* of an edit mapping M as:

$$\delta(M) = \sum_{(i,j)\in M} \gamma(f(i),g(j)) + \sum_{i \in R_F} \gamma(f(i),-) + \sum_{j \in R_G} \gamma(-,g(j)).$$

An *optimal edit mapping* between two forests F and G is an edit mapping with the minimum cost: $\min\{\delta(M)\}$ over all possible M. This cost is called *the forest edit distance* between F and G, and is denoted by $\delta(F,G)$.

Subforest definitions: Let F be a forest. We define the following types of subforests of F. For any node i in F, the *subtree of F rooted at i* is the subtree consisting of i and all descendants of i, and is denoted by $F[i]$. For any siblings i_1, i_2, the set of subtrees rooted at $i_1 \cdot\cdot\, i_2$ forms a *closed subforest of F* (see also [4]). A *simple substructure of F* is any connected subgraph of F, and a *sibling substructure of F* is a set of disjoint simple substructures of F whose roots are siblings (not necessarily consecutive) in F. Finally, given any subset S of the nodes in F, the *restricted subforest* $F\|_S$ is defined as the forest obtained from F by deleting all nodes not in S. To illustrate these definitions, consider the forest F in Fig. 1(c) and the subforests of F shown in Fig. 1(d). Here, $F_1{=}F\|_S$ is a restricted subforest for $S = \{3,5,6,9,13,22,31\}$, F_2 is a simple substructure of F, F_3 is a closed subforest of F, and F_4 is a sibling substructure of F.

We say that a *most similar subforest* (using any one of the above definitions for "subforest") of a forest F to a forest G is a subforest F' of F that minimizes the forest edit distance $\delta(F',G)$ over all possible F'. The *most similar subforest*

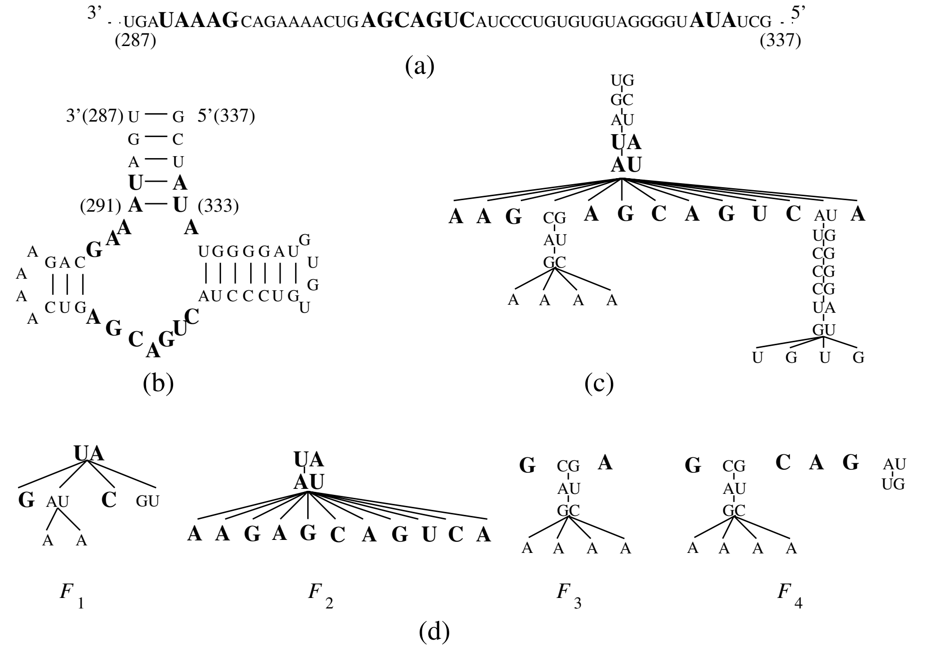


Fig. 1. (a) A segment of the primary structure of the cherry small circular viroid-like RNA molecule (accession number Y12833, GI:2347024) [10], (b) its secondary structure, (c) a forest representation F of the secondary structure (see, e.g., [4,7,11] for details), and (d) various types of subforests of F. The so-called *Hammerhead motif* [10], which corresponds to the pattern specified by subforest F_2, is marked in bold.

problem is: given two forests F and G, find a most similar subforest of F to G (again, using any one of the above definitions for "subforest").

Our contributions: In this paper, we show how to solve the most similar subforest problem efficiently, where "subforest" means "simple substructure", "sibling substructure", or "closed subforest". The time and space complexities of our algorithms are summarized in the next table.

Finding a most similar:	Complexity	Section
Simple substructure	$O(\lvert F\rvert \cdot \lvert G\rvert \cdot \min\{\lvert L(F)\rvert, dp(F)\} \cdot \min\{\lvert L(G)\rvert, dp(G)\})$ time, $O(\lvert F\rvert \cdot \lvert G\rvert)$ space	3.1
Sibling substructure	$O(\lvert F\rvert \cdot \lvert G\rvert \cdot \min\{\lvert L(F)\rvert, dp(F)\} \cdot \min\{\lvert L(G)\rvert, dp(G)\})$ time, $O(\lvert F\rvert \cdot \lvert G\rvert)$ space	3.2
Closed subforest	$O(\lvert F\rvert \cdot \lvert G\rvert \cdot \lvert L(F)\rvert \cdot \min\{\lvert L(G)\rvert, dp(G)\})$ time, $O(\lvert F\rvert \cdot \lvert G\rvert + \lvert L(F)\rvert \cdot dp(F) \cdot \lvert G\rvert + \lvert F\rvert \cdot \lvert L(G)\rvert \cdot dp(G))$ space	3.3

Related results: Tai [13] gave the first algorithm for computing the forest edit distance between two given forests F and G. Zhang and Shasha [16] gave a more efficient algorithm for this problem running in $O(|F| \cdot |G| \cdot \min\{|L(F)|, dp(F)\} \cdot \min\{|L(G)|, dp(G)\})$ time and $O(|F| \cdot |G|)$ space. (Actually, these papers assumed F and G to be *trees*, but it is simple to extend their methods to forests.) More recently, Klein [9], Chen [1], and Touzet [14] have developed algorithms that are faster for certain kinds of inputs. Zhang and Shasha's algorithm [16] computes $\delta(F[i], G[j])$ for all $i \in F$, $j \in G$, and can therefore find a subtree rooted at a node in F which is the most similar to G, i.e., a most similar rooted subtree, but none of the algorithms from [1,9,13,14,16] can be used directly to efficiently find, e.g., a most similar simple substructure of F to G (the number of simple substructures of F may be exponential in $|F|$, so it is not practical to try them all separately).

An alternative measure of the similarity between two forests is the *forest alignment distance* (see [7] for a formal definition). Although the edit distance and alignment distance are equivalent for *strings*, they are not equivalent for trees and forests [7]. The algorithm of Jiang *et al.* [7] for computing the forest alignment distance runs in $O(|F| \cdot |G| \cdot (\deg(F) + \deg(G))^2)$ time, and its running time was improved for similar inputs in [5]. Note that the algorithm of Jiang *et al.* computes an optimal *global* alignment between F and G, meaning that all nodes of F and G contribute to the cost of the final solution. Recently, Höchsmann *et al.* [4] gave an algorithm for computing an optimal *local* alignment between F and G which finds a closed subforest F' of F and a closed subforest G' of G having the minimum forest alignment distance; a more efficient algorithm for this problem (running in $O(|F| \cdot |G| \cdot (\deg(F) + \deg(G))^2)$ time and $O(|F| \cdot |G| \cdot (\deg(F) + \deg(G)))$ space) along with some extensions to other types of subforests were given in [6]. Höchsmann *et al.* [4] also considered the problem of finding a closed subforest F' of F which minimizes the alignment distance to G (i.e., the analogue of our "most similar closed subforest problem" but using alignment distance instead of edit distance), which they called the *small-in-large closed subforest similarity problem*, and showed how to solve it in $O(|F| \cdot |G| \cdot \deg(F) \cdot \deg(G) \cdot (\deg(F) + \deg(G)))$ time and $O(|F| \cdot |G| \cdot \deg(F) \cdot \deg(G))$ space.

3 Algorithms for the Most Similar Subforest Problem

In this section, we present efficient algorithms for a finding a most similar subforest (simple substructure, sibling substructure, and closed subforest, respectively) of F to G. As a preprocessing step to all our algorithms below, we calculate and store $K(F), K(G)$, $L(F)$, and $L(G)$ according to their postorders in auxiliary arrays in linear time. Moreover, $m(i)$ for all $i \in F$, $i \in G$ are also precomputed.

3.1 An Algorithm for Finding a Most Similar Simple Substructure

We first introduce some additional terminology. Let F be a forest. Define a new edit operation called the *cut operation* on F as follows: for any node i in F, *cutting node* i means removing the entire subtree $F[i]$ (along with the parent edge of i if i is not a root node) from F at cost 0. Note that the cut operation differs from the previously defined delete operation since it removes *all* the nodes in a subtree of F and is for free. For any two nodes u and v in F with $u \neq v$, we say that u and v are *consistent* if u is not a descendant of v and v is not a descendant of u. A set C of nodes from F is *consistent* if every pair of nodes in C is consistent. Denote the set of all consistent sets of nodes in F by $\mathcal{C}(F)$, and for any $C \in \mathcal{C}(F)$, let $F \ominus C$ be the forest obtained from F by cutting all nodes in C.

Suppose F' is a simple substructure of F rooted at a node i. By definition, F' is a connected subgraph of $F[i]$, which means that F' can be obtained from $F[i]$ by cutting all nodes in some (possibly empty) consistent set. We have:

Lemma 1. *Let i be a node in a forest F. F' is a simple substructure of F rooted at i if and only if $F' = F[i] \ominus C$ for some $C \in \mathcal{C}(F[i])$.*

To locate a most similar simple substructure of F to G, we look for a most similar simple substructure of $F[i]$ to G among all $i \in F$. By Lemma 1, this is equivalent to finding a $C \in \mathcal{C}(F[i])$ such that $\delta(F[i] \ominus C,\, G)$ is minimized since the cut operations do not contribute to the total cost of an edit mapping between $F[i]$ and G. (Observe that we are only allowed to cut nodes in F, and not in G by the problem definition.) For any given forests F' and G', define $\Psi(F', G') = \min_{C \in \mathcal{C}(F')}\{\delta(F' \ominus C,\, G')\}$. Then the goal of our algorithm is to compute $\min_{i \in F} \Psi(F[i], G)$. Below, we extend the techniques of [16] to derive some useful recurrences for computing certain values of Ψ.

First of all, it is easy to show that:

Lemma 2. $\Psi(\emptyset, \emptyset) = 0$*;* $\Psi(F, \emptyset) = 0$*;* $\Psi(\emptyset, G) = \sum_{j \in G} \gamma(-, g(j))$*.*

Proof. The first case is obvious since there is no cost for the empty mapping between two empty forests. For the second case, suppose $F = \langle T_1, \ldots, T_t \rangle$. We know that $F \ominus \{r(T_1), \ldots, r(T_t)\}) = \emptyset$, where $r(T)$ for any tree T refers to the root of T, so $\Psi(F, \emptyset) = \delta(\emptyset, \emptyset) = 0$. In the third case, we cannot cut any nodes, so $\Psi(\emptyset, G) = \sum_{j \in G} \gamma(-, g(j))$. □

Next, because of the left-to-right postordering of the nodes, we have $F[i] = F\|_{m(i):i}$ and $G[j] = G\|_{m(j):j}$. To compute $\Psi(F[i], G[j])$, we compute $\Psi(F\|_{m(i):x}, G\|_{m(j):y})$ for all $x \in \{m(i), \ldots, i\}$ and $y \in \{m(j), \ldots, j\}$. Intuitively, when considering nodes x and y, if the subtree $F[x]$ is very dissimilar to $G[y]$ then it will be better to cut x (i.e., remove the entire subtree $F[x]$ at once at no additional cost), in which case $F\|_{m(i):x}$ becomes just $F\|_{m(i):m(x)-1}$. On the other hand, if $F[x]$ is similar to $G[y]$ then x and y should be linked, or one of x and y should be deleted, and then the remaining parts of $F[x]$ and $G[y]$ linked.

Lemma 3. *For any $i \in F$, $j \in G$, $x \in \{m(i), \ldots, i\}$, and $y \in \{m(j), \ldots, j\}$,*

$$\Psi(F\|_{m(i):x}, G\|_{m(j):y}) = \min \begin{cases} \Psi(F\|_{m(i):m(x)-1}, G\|_{m(j):y}); \\ \Psi(F\|_{m(i):x-1}, G\|_{m(j):y}) + \gamma(f(x), -); \\ \Psi(F\|_{m(i):x}, G\|_{m(j):y-1}) + \gamma(-, g(y)); \\ \Psi(F\|_{m(i):m(x)-1}, G\|_{m(j):m(y)-1}) + \\ \qquad \Psi(F\|_{m(x):x-1}, G\|_{m(y):y-1}) + \gamma(f(x), g(y)). \end{cases}$$

Proof. Let $C \in \mathcal{C}(F\|_{m(i):x})$ be a consistent set that minimizes $\delta(F\|_{m(i):x} \ominus C, G\|_{m(j):y})$, i.e., such that $\delta(F\|_{m(i):x} \ominus C, G\|_{m(j):y}) = \Psi(F\|_{m(i):x}, G\|_{m(j):y})$, and let M be an optimal edit mapping between $F\|_{m(i):x} \ominus C$ and $G\|_{m(j):y}$. Consider nodes x and y and the set C:

- $x \in C$: In this case, x is cut and so the whole subtree $F[x]$ is removed at no cost. We get $\Psi(F\|_{m(i):x}, G\|_{m(j):y}) = \Psi(F\|_{m(i):m(x)-1}, G\|_{m(j):y})$.
- $x \notin C$: In this case, x is either deleted or linked with a node in $G\|_{m(j):y}$, and analogously for node y. There are three possible subcases:
  - $x \notin M_F$: Then x is deleted from $F\|_{m(i):x}$ in the optimal solution given by M, so $\Psi(F\|_{m(i):x}, G\|_{m(j):y}) = \Psi(F\|_{m(i):x-1}, G\|_{m(j):y}) + \gamma(f(x), -)$.
  - $y \notin M_G$: Then y is deleted from $G\|_{m(j):y}$ in the optimal solution given by M, so $\Psi(F\|_{m(i):x}, G\|_{m(j):y}) = \Psi(F\|_{m(i):x}, G\|_{m(j):y-1}) + \gamma(-, g(y))$.
  - $x \in M_F$ and $y \in M_G$: Then nodes x and y are linked in the optimal solution given by M. We get $\Psi(F\|_{m(i):x}, G\|_{m(j):y}) = \Psi(F\|_{m(i):m(x)-1}, G\|_{m(j):m(y)-1}) + \Psi(F\|_{m(x):x-1}, G\|_{m(y):y-1}) + \gamma(f(x), g(y))$. □

To simplify the implementation of the algorithm described below, rewrite the recurrence relation in Lemma 3 as follows so that $\Psi(F\|_{m(i):x}, G\|_{m(j):y})$ can be computed without needing to access the value of $\Psi(F\|_{m(x):x-1}, G\|_{m(y):y-1})$.

Lemma 4. *For any $i \in F$, $j \in G$, $x \in \{m(i), \dots, i\}$, and $y \in \{m(j), \dots, j\}$,*
1. if $m(i) = m(x)$ and $m(j) = m(y)$ then:

$$\Psi(F\|_{m(i):x}, G\|_{m(j):y}) = \min \begin{cases} \Psi(F\|_{m(i):m(x)-1}, G\|_{m(j):y}); \\ \Psi(F\|_{m(i):x-1}, G\|_{m(j):y}) + \gamma(f(x), -); \\ \Psi(F\|_{m(i):x}, G\|_{m(j):y-1}) + \gamma(-, g(y)); \\ \Psi(F\|_{m(i):x-1}, G\|_{m(j):y-1}) + \gamma(f(x), g(y)). \end{cases}$$

2. else:

$$\Psi(F\|_{m(i):x}, G\|_{m(j):y}) = \min \begin{cases} \Psi(F\|_{m(i):m(x)-1}, G\|_{m(j):y}); \\ \Psi(F\|_{m(i):x-1}, G\|_{m(j):y}) + \gamma(f(x), -); \\ \Psi(F\|_{m(i):x}, G\|_{m(j):y-1}) + \gamma(-, g(y)); \\ \Psi(F\|_{m(i):m(x)-1}, G\|_{m(j):m(y)-1}) + \Psi(F[x], G[y]). \end{cases}$$

Proof. We prove this lemma by showing that the new recurrences are equivalent to the one in Lemma 3. Note that in both cases, only the fourth term inside the min-bracket differs from Lemma 3.

1. Since $m(i) = m(x)$ and $m(j) = m(y)$, we have $\{m(i), \dots, m(x) - 1\} = \emptyset$ and $\{m(j), \dots, m(y) - 1\} = \emptyset$, and hence $\Psi(F\|_{m(i):m(x)-1}, G\|_{m(j):m(y)-1}) = 0$.
2. The definition of Ψ implies that $\Psi(F\|_{m(i):x}, G\|_{m(j):y}) \leq \Psi(F\|_{m(i):m(x)-1}, G\|_{m(j):m(y)-1}) + \Psi(F[x], G[y])$. Thus, inserting the right-hand side of this inequality into the min-expression in Lemma 3 does not affect its value, i.e., $\Psi(F\|_{m(i):x}, G\|_{m(j):y}) = \min\{\dots, \Psi(F\|_{m(i):m(x)-1}, G\|_{m(j):m(y)-1}) + \Psi(F[x], G[y])\}$ where ... denotes the four terms in Lemma 3. Now, by case 1 above, $\Psi(F[x], G[y]) \leq \Psi(F\|_{m(x):x-1}, G\|_{m(y):y-1}) + \gamma(f(x), g(y))$, so the fourth term in the new min-expression is redundant and can be deleted. □

```
Main loop:
Input: A target forest $F$ and a pattern forest $G$.
$\Psi(\emptyset, \emptyset) := 0$.
for $i_1 := 1, \ldots, |K(F)|\}$ do
    for $j_1 := 1, \ldots, |K(G)|$ do
        $i := K(F)[i_1]$; $j := K(G)[j_1]$; Call Compute_Psi$(i, j)$.
return $\min_{i \quad F} \Psi(F[i], G)$.
```

```
Procedure Compute_Psi$(i, j)$:
for $x := m(i), \ldots, i$ do $\Psi(F\|_{m(i):x}, \emptyset) := 0$.
for $y := m(j), \ldots, j$ do $\Psi(\emptyset, G\|_{m(j):y}) := \Psi(\emptyset, G\|_{m(j):y \quad 1}) + \gamma(-, g(y))$.
for $x := m(i), \ldots, i$ do
    for $y := m(j), \ldots, j$ do
        Calculate $\Psi(F\|_{m(i):x}, G\|_{m(j):y})$ according to Lemma 4.
```

Algorithm 1. Algorithm for finding a most similar simple substructure of F to G

For each $i \in F$, define $A(i)$ (*the nearest key node ancestor of* i) as follows. If $i \in K(F)$ then let $A(i) = i$; otherwise, let $A(i)$ be the nearest ancestor of i which belongs to $K(F)$. Define $A(j)$ for any node $j \in G$ analogously. We have:

Lemma 5. *For any* $i \in F$, $m(i) = m(A(i))$. *For any* $j \in G$, $m(j) = m(A(j))$.

We now describe the main algorithm (Algorithm 1) of this subsection. It calculates the minimum cost of an edit mapping between a simple substructure of F and G. The main loop considers all pairs of indices $i \in K(F)$ and $j \in K(G)$ in bottom-up order, and for each such pair of indices (i, j), it calls a procedure named Compute_Psi to obtain $\Psi(F\|_{m(i):x}, G\|_{m(j):y})$ for all $x \in \{m(i), \ldots, i\}$ and $y \in \{m(j), \ldots, j\}$ based on Lemmas 2 and 4. Finally, Algorithm 1 returns $\min_{i \in F} \Psi(F[i], G)$. The next theorem proves the correctness of this approach.

Theorem 1. *Algorithm 1 correctly computes the cost of an optimal solution.*

Proof. The correctness of all computed values follows from Lemmas 2 and 4. Let x be a node in F such that the cost of an optimal solution is given by $\Psi(F[x], G)$. We need to prove that the algorithm is guaranteed to compute $\Psi(F[x], G)$ even if $x \notin K(F)$. Observe that $\Psi(F[x], G) = \Psi(F\|_{m(x):x}, G) = \Psi(F\|_{m(A(x)):x}, G)$ by Lemma 5, and that $x \in \{m(A(x)), \ldots, A(x)\}$ because $m(A(x)) = m(x) \leq x$ and $x \leq A(x)$. Since $A(x) \in K(F)$ by the definition of $A(x)$ and since $p(G) \in K(G)$, the algorithm will always compute $\Psi(F[x], G)$. □

Theorem 2. *Algorithm 1 can be implemented to run in* $O(|F| \cdot |G| \cdot \min\{|L(F)|, dp(F)\} \cdot \min\{|L(G)|, dp(G)\})$ *time and* $O(|F| \cdot |G|)$ *space.*

Proof. Store $\Psi(F[i], G[j])$ for every $i \in F$ and $j \in G$ as soon as it is computed in a table M_1 of size $|F| \cdot |G|$. Also, allocate $(|F|+1) \cdot (|G|+1)$ additional space M_2 for Compute_Psi to temporarily store the computed values of $\Psi(F\|_{m(i):x}, G\|_{m(j):y})$ for all $x \in \{m(i) - 1, \ldots, i\}$ and $y \in \{m(j) - 1, \ldots, j\}$ for its current (i, j).

(M_2 is reused by successive calls to `Compute_Psi`.) In total, the space complexity is $O(|F| \cdot |G|)$.

To analyze the running time of this implementation, we first show that Step 5 in `Compute_Psi` (evaluating the expression in Lemma 4) for any $\Psi(F\|_{m(i):x}, G\|_{m(j):y})$ always takes $O(1)$ time. Whenever Step 5 is performed, the values of $\Psi(F\|_{m(i):m(x)-1}, G\|_{m(j):y})$, $\Psi(F\|_{m(i):x-1}, G\|_{m(j):y})$, $\Psi(F\|_{m(i):x}, G\|_{m(j):y-1})$, $\Psi(F\|_{m(i):x-1}, G\|_{m(j):y-1})$, and $\Psi(F\|_{m(i):m(x)-1}, G\|_{m(j):m(y)-1})$ are already stored in M_2. Therefore, we can directly evaluate the expression in $O(1)$ time if $m(i) = m(x)$ and $m(j) = m(y)$. If $m(i) < m(x)$ or $m(j) < m(y)$, we also need $\Psi(F[x], G[y])$ in $O(1)$ time. This value has already been computed and stored in M_1 because $m(i) < m(x)$ implies $m(i) < m(A(x))$ by Lemma 5 which in turn implies $A(x) < i$, and $m(j) < m(y)$ similarly implies $A(y) < j$; since at least one of these two conditions is true, the algorithm will have called `Compute_Psi`$(A(x), A(y))$ previously and hence already have computed $\Psi(F[x], G[y])$. Thus, Step 5 in `Compute_Psi` takes $O(1)$ time, which means that the algorithm's total running time is $O(\sum_{i \in K(F)} \sum_{j \in K(G)} |F[i]| \cdot |G[j]|)$. By Lemma 7 in [16], this sum can be rewritten as $O(|F| \cdot |G| \cdot \min\{|L(F)|, \mathit{dp}(F)\} \cdot \min\{|L(G)|, \mathit{dp}(G)\})$. The theorem follows. □

We remark that Algorithm 1 computes the *cost* of an optimal solution. Standard traceback techniques can be applied to also return a corresponding optimal edit mapping within the same asymptotic running time and space bounds.

3.2 An Algorithm for Finding a Most Similar Sibling Substructure

An algorithm for finding a most similar sibling substructure of F to G is given here. It is based on Algorithm 1 since finding a most similar sibling substructure is closely related to finding a most similar simple substructure, as shown next.

Suppose that $F' = \langle T_1, \ldots, T_s \rangle$ is a most similar sibling substructure of F to G, where $\{T_1, \ldots, T_s\}$ are simple substructures of F with roots $\{i_1, \ldots, i_s\}$, and where F' is non-empty. Then $\{i_1, \ldots, i_s\}$ are siblings in F. Let $S(i_1)$ be the set consisting of i_1 and all siblings of i_1 in F, and define $C = S(i_1) \setminus \{i_1, \ldots, i_s\}$. Note that C is a consistent set, i.e., $C \in \mathcal{C}(F)$, using the notation from Section 3.1. Consider the closed subforest $F\|_{m(b(i_1)):e(i_1)}$. It is clear that F' is also a most similar sibling substructure of $F\|_{m(b(i_1)):e(i_1)}$ to G. (This claim comes from cutting all nodes belonging to C from $F\|_{m(b(i_1)):e(i_1)}$ at zero cost.) So $\delta(F', G) = \Psi(F\|_{m(b(i_1)):e(i_1)}, G)$, and the problem turns into finding the minimum of $\Psi(F\|_{m(b(i_1)):e(i_1)}, G)$ over all $i_1 \in F$. By the left-to-right postordering of the nodes, this is equivalent to computing $\min_{i \in F} \Psi(F\|_{m(i):i-1}, G)$. [2]

We modify the implementation of Algorithm 1 given in the proof of Theorem 2 as follows. Allocate $O(|F|)$ extra space M_3 to also store the values of $\Psi(F\|_{m(i):i-1}, G)$ for all $i \in F$ as they are computed. Then, change Step 5 of the main loop to return the value $\min_{i \in F} \Psi(F\|_{m(i):i-1}, G)$ (found by checking M_3) instead. Clearly, the asymptotic time and space complexities are the same as for

[2] In contrast, recall from Section 3.1 that finding a most similar *simple* substructure is equivalent to computing $\min_{i \quad F} \Psi(F\|_{m(i):i}, G)$.

Algorithm 1. To prove the correctness of the modified algorithm, we show that all values of $\Psi(F\|_{m(i):i-1}, G)$ for $i \in F$ are indeed computed. Let i be any node in F. Recall that $A(i)$ is the nearest key node ancestor of i. Then $m(i) = m(A(i))$ by Lemma 5, and $i \leq A(i)$ and $A(i) \in K(F)$. According to the proof of Theorem 2, for any $k \in K(F)$, the algorithm will compute $\Psi(F\|_{m(k):x}, G)$ for all $x \in \{m(k), \ldots, k\}$. Now, select $k = A(i)$ and $x = i - 1$. We obtain:

Theorem 3. *Given a target forest F and a pattern forest G, we can find a most similar sibling substructure of F to G over all sibling substructures of F in $O(|F|\cdot|G|\cdot\min\{|L(F)|, dp(F)\}\cdot\min\{|L(G)|, dp(G)\})$ time and $O(|F|\cdot|G|)$ space.*

3.3 An Algorithm for Finding a Most Similar Closed Subforest

We now provide an algorithm for finding a most similar closed subforest of F to G. The algorithm of [16] as well as Algorithm 1 from Section 3.1 are not suitable for this variant of the problem because if i_1 and i_2 are siblings in F with $i_1 < i_2$, $i_1 \in K(F)$ then the value of $\delta(F\|_{m(i_1):i_2}, G)$ is not calculated, whereas $F\|_{m(i_1):i_2} = F[i_1 \cdot \cdot i_2]$ might in fact be a most similar closed subforest of F to G. Therefore, we develop a different technique in this subsection. The proofs of Lemma 6, Lemma 7, and Theorem 4 below are similar to those of Lemma 2, Lemmas 3–4, and Theorems 1– 2, respectively, and have been omitted due to space constraints.

For any forest F, any leaf $l \in L(F)$, and any node $x \in F$, write $l \preceq x$ if l is a descendant of x in F, and $l \not\preceq x$ otherwise. Since $m(x)$ is precomputed, we can immediately test if $l \preceq x$ simply by checking if $m(x) \leq l \leq x$ is true. The next lemmas state how to efficiently calculate $\delta(F\|_{l:x}, G\|_{m(j):y})$ where $l \in L(F)$, $x \in \{l, \ldots, |F|\}$, $j \in K(G)$, and $y \in \{m(j), \ldots, j\}$.

Lemma 6. $\delta(\emptyset,\emptyset)=0$; $\delta(F,\emptyset)=\sum_{i\in F}\gamma(f(i),-)$; $\delta(\emptyset,G)=\sum_{j\in G}\gamma(-,g(j))$.

Lemma 7. *For any $l \in L(F)$, $j \in K(G)$, $x \in \{l, \ldots, |F|\}$, $y \in \{m(j), \ldots, j\}$, $\delta(F\|_{l:x}, G\|_{m(j):y})$ is equal to the minimum of the following three values:*

$$\begin{cases} \delta(F\|_{l:x-1}, G\|_{m(j):y}) + \gamma(f(x), -); \\ \delta(F\|_{l:x}, G\|_{m(j):y-1}) + \gamma(-, g(y)); \\ \begin{cases} \delta(F\|_{l:x-1}, G\|_{m(j):y-1}) + \gamma(f(x), g(y)), & \textit{if } l \preceq x \textit{ and } m(j) \preceq y; \\ \delta(\emptyset, G\|_{m(j):m(y)-1}) + \delta(F\|_{l:x}, G[y]), & \textit{if } l \preceq x \textit{ and } m(j) \not\preceq y; \\ \delta(F\|_{l:m(x)-1}, \emptyset) + \delta(F[x], G\|_{m(j):y}), & \textit{if } l \not\preceq x \textit{ and } m(j) \preceq y; \\ \delta(F\|_{l:m(x)-1}, G\|_{m(j):m(y)-1}) + \delta(F[x], G[y]), & \textit{if } l \not\preceq x \textit{ and } m(j) \not\preceq y. \end{cases} \end{cases}$$

Now we are ready to describe Algorithm 2 for finding a most similar closed subforest of F to G. Its overall structure resembles that of Algorithm 1. For each leaf $l \in L(F)$ and node $j \in K(G)$, it calls a procedure named `Compute_Delta` which uses Lemma 7 to calculate $\delta(F\|_{l:x}, G\|_{m(j):y})$ for all $x \in \{l, \ldots, |F|\}$ and $y \in \{m(j), \ldots, j\}$. To enable each evaluation of Lemma 7 to be performed in $O(1)$ time, the algorithm temporarily stores the computed values of $\delta(F\|_{l:x}, G\|_{m(j):y})$ for all $x \in \{l, \ldots, |F|\}$ and $y \in \{m(j), \ldots, j\}$ until the next call to `Compute_Delta` using $O(|F| \cdot |G|)$ space; on the other hand, all computed values of the form

```
Main loop:
Input: A target forest $F$ and a pattern forest $G$.
$\delta(\emptyset, \emptyset) := 0$.
for $l_1 := |L(F)|, \ldots, 1$ do
    for $j_1 := 1, \ldots, |K(G)|$ do
        $l := L(F)[l_1]$; $j := K(G)[j_1]$; Call Compute_Delta($l, j$).
return $\min\{\delta(F\|_{m(i_1):i_2}, G) \mid i_1, i_2$ are siblings in $F\}$.
```

```
Procedure Compute_Delta($l, j$):
for $x := l, \ldots, |F|$ do $\delta(F\|_{l:x}, \emptyset) := \delta(F\|_{l:x\;\;1}, \emptyset) + \gamma(f(x), -)$.
for $y := m(j), \ldots, j$ do $\delta(\emptyset, G\|_{m(j):y}) := \delta(\emptyset, G\|_{m(j):y\;\;1}) + \gamma(-, g(y))$.
for $x := l, \ldots, |F|$ do
    for $y := m(j), \ldots, j$ do
        Calculate $\delta(F\|_{l:x}, G\|_{m(j):y})$ according to Lemma 7.
```

Algorithm 2. Algorithm for finding a most similar closed subforest of F to G

$\delta(F[x], G[y])$, $\delta(F[x], G\|_{m(j):y})$ with $m(j) \preceq y$, and $\delta(F\|_{l:x}, G[y])$ with $l \preceq x$ are stored throughout the entire execution of the algorithm using an additional $O(|F| \cdot |G| + |F| \cdot |K(G)| \cdot dp(G) + |L(F)| \cdot dp(F) \cdot |G|)$ space. Finally, the algorithm returns $\min\{\delta(F\|_{m(i_1):i_2}, G) \mid i_1$ and i_2 are siblings in $F\}$.

Theorem 4. *Given a target forest F and a pattern forest G, we can find a most similar closed subforest of F to G over all closed subforests of F in $O(|F| \cdot |G| \cdot |L(F)| \cdot \min\{|L(G)|, dp(G)\})$ time and $O(|F| \cdot |G| + |L(F)| \cdot dp(F) \cdot |G| + |F| \cdot |L(G)| \cdot dp(G))$ space.*

4 Concluding Remarks

It is straightforward to generalize our algorithms to find a subforest F' of F and a subforest G' of G that are the most similar for any combination of the types of subforests considered above. For example, if both F' and G' should be simple substructures then we can modify Algorithm 1 to allow nodes in G to be cut too.

An open question is: Is it possible to extend the algorithms in this paper to other types of subforests? For example, one might consider *gapped subforests* (introduced in [6]), where a gapped subforest of F is obtained by removing from any closed subforest F' of F a set C of closed subforests such that no two closed subforests in C have the same parent in F'.

References

1. W. Chen. New algorithm for ordered tree-to-tree correction problem. *Journal of Algorithms*, 40(2):135–158, 2001.
2. G. Cobéna, S. Abiteboul, and A. Marian. Detecting changes in XML documents. In *Proceedings of the 18th IEEE International Conference on Data Engineering* (ICDE 2002), pages 41–52, 2002.

3. M. Crochemore and W. Rytter. *Text algorithms*. Oxford University Press, 1994.
4. M. Höchsmann, T. Töller, R. Giegerich, and S. Kurtz. Local similarity in RNA secondary structures. In *Proceedings of the IEEE Computational Systems Bioinformatics Conference* (CSB 2003), pages 159–168, 2003.
5. J. Jansson and A. Lingas. A fast algorithm for optimal alignment between similar ordered trees. *Fundamenta Informaticae*, 56(1–2):105–120, 2003.
6. J. Jansson, T. H. Ngo, and W.-K. Sung. Local gapped subforest alignment and its application in finding RNA structural motifs. In *Proceedings of the 15th International Symposium on Algorithms and Computation* (ISAAC 2004), pages 569–580, 2004.
7. T. Jiang, L. Wang, and K. Zhang. Alignment of trees - an alternative to tree edit. *Theoretical Computer Science*, 143:137–148, 1995.
8. P. Kilpeläinen and H. Mannila. Ordered and unordered tree inclusion. *SIAM Journal on Computing*, 24(2):340–356, 1995.
9. P. N. Klein. Computing the edit-distance between unrooted ordered trees. In *Proceedings of the 6th European Symposium on Algorithms* (ESA 1998), pages 91–102, 1998.
10. Motifs database. `http://subviral.med.uottawa.ca/cgi-bin/motifs.cgi`.
11. B. A. Shapiro and K. Zhang. Comparing multiple RNA secondary structures using tree comparisons. *Computer Applications in the Biosciences*, 6(4):309–318, 1990.
12. T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.
13. K.-C. Tai. The tree-to-tree correction problem. *Journal of the ACM*, 26(3):422–433, 1979.
14. H. Touzet. A linear time edit distance algorithm for similar ordered trees. In *Proceedings of the 16th Symposium on Combinatorial Pattern Matching* (CPM 2005), pages 334–345, 2005.
15. G. Valiente. Constrained tree inclusion. In *Proceedings of the 14th Symposium on Combinatorial Pattern Matching* (CPM 2003), pages 361–371, 2003.
16. K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal on Computing*, 18(6):1245–1262, 1989.

Efficient Algorithms for Regular Expression Constrained Sequence Alignment

Yun-Sheng Chung[1], Chin Lung Lu[2], and Chuan Yi Tang[1],*

[1] Department of Computer Science, National Tsing Hua University,
Hsinchu, Taiwan 300, ROC
yschung@algorithm.cs.nthu.edu.tw, cytang@cs.nthu.edu.tw

[2] Department of Biological Science and Technology, National Chiao Tung University,
Hsinchu, Taiwan 300, ROC
cllu@mail.nctu.edu.tw

Abstract. Imposing constraints is an effective means to incorporate biological knowledge into alignment procedures. As in the PROSITE database, functional sites of proteins can be effectively described as regular expressions. In an alignment of protein sequences it is natural to expect that functional motifs should be aligned together. Due to this motivation, in CPM 2005 Arslan introduced the regular expression constrained sequence alignment problem and proposed an algorithm which can take time and space up to $O(|\Sigma|^2 |V|^4 n^2)$ and $O(|\Sigma|^2 |V|^4 n)$, respectively, where Σ is the alphabet, n is the sequence length, and V is the set of states in an automaton equivalent to the input regular expression. In this paper we propose a more efficient algorithm solving this problem which takes $O(|V|^3 n^2)$ time and $O(|V|^2 n)$ space in the worst case. If $|V| = O(\log n)$ we propose another algorithm with time complexity $O(|V|^2 \log |V| n^2)$. The time complexity of our algorithms is independent of Σ, which is desirable in protein applications where the formulation of this problem originates; a factor of $|\Sigma|^2 = 400$ in the time complexity of the previously proposed algorithm would significantly affect the efficiency in practice.

1 Introduction

Sequence alignment is one of the most fundamental problems in computational biology. Due to its importance, it has been extensively studied in the past (see, e.g., [1]). As biological knowledge and predictions grow, it is often desirable if one can incorporate more information into the alignment procedure in hope that the alignment result can be more biologically meaningful and reasonable. In particular, when the input sequences share some properties, one then expects that the resulting alignment should not violate these properties. Such kind of preservation is in its nature a satisfaction of properly defined constraints. Due to this need, Tang et al. [2] defined the constrained sequence alignment problem (CSA for short). They considered the alignment of RNase sequences which share

* To whom correspondence should be addressed.

M. Lewenstein and G. Valiente (Eds.): CPM 2006, LNCS 4009, pp. 389–400, 2006.

a conserved sequence of residues H, K, H. It is expected that the alignment result should have these conserved residues aligned together. A constraint is then defined in [2] as a sequence of characters. The problem requires that the alignment result must contain a sequence of columns, with length the same as the constraint, such that each character in the constraint appears in exactly one of these columns and that each column consists solely of a character. The goal is to find the best such alignment.

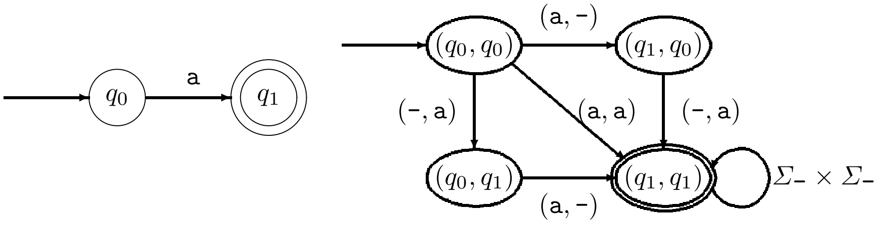
Later, Chin et al. [3] proposed an efficient algorithm for the pairwise version of CSA, along with a 2-approximation algorithm for the multiple alignment version with scoring function satisfying triangle inequality. Tsai et al. [4] generalized the definition of a constraint from a sequence of characters to a sequence of strings (patterns), and the occurrences of each pattern in the sequences need not be identical to the pattern specified in the input. Instead, the Hamming distances between each pattern and its occurrences need only to be within a user-specified threshold. This formulation enables the user to align sequences so that some known motifs are required to be aligned together. Lu and Huang [5] then reduced the memory requirement of the algorithm in [4], which significantly improves the applicability of the tool.

As indicated by Arslan [6], it is well known that proteins with similar functions often share some motifs. The PROSITE database [7] collects biologically significant sites and motifs of proteins. In particular, these motifs can be conveniently represented as regular expressions. In addition, IUPAC code [8] is also commonly used to represent motifs for both nucleic acids and amino acids sequences, and can also be expressed in regular expressions. Hence an alignment tool able to incorporate patterns in regular expressions would be useful. To fulfill this need, Arslan introduced the regular expression constrained sequence alignment problem (RECSA for short) [6]. A feasible solution of RECSA is an alignment containing a run of contiguous columns corresponding to two substrings, one for each input sequence, such that both substrings match the regular expression. The following example, constructed by Arslan [6], clearly illustrates the definition and its difference with a standard alignment without constraint:

```
T G F P S V G K T K D D - - - - A   T - - - G F P S V G K T K D D A
|   |   | |       | | |         |   |                             |
T - F - S V A - - K D D D G K S A   T F S V A K D D D G K S - - - A
                                            * * * * * * * *
```

In all examples given in this section, a match of identical symbols is scored 1 and all other cases are scored 0. The alignment shown left is an optimal alignment without constraint, while that shown in the right is an optimal constrained alignment. The constraint R is $(\mathtt{G}+\mathtt{A})\Sigma\Sigma\Sigma\Sigma\mathtt{GK}(\mathtt{S}+\mathtt{T})$, the P-loop motif. The starred columns "support" the satisfaction of constraint R: both GFPSVGKT and AKDDDGKS match R. For more descriptions about biological applications of RECSA, the reader is referred to Arslan's paper [6].

There are well-established algorithms to convert R into an ε-free NFA A (see, e.g., [9]). In [6], R is converted into A first. Then a weighted finite automaton M is constructed from A to accept all alignments satisfying R. Automaton M is essentially a product machine of A. Each state in M corresponds to a pair of states in A. The alphabet of M corresponds to edit operations, which is represented as a pair of symbols in $\Sigma_{-} = \Sigma \cup \{\texttt{-}\}$. The transitions in M and their relationship with those in A are illustrated in the following simple example, where $R = \texttt{a}$:



Shown left is A and shown right is M. Given a sequence of edit operations, or equivalently an alignment, as input to M, some states in M can be reached from the initial state (q_0, q_0). For example, given $\left[\begin{smallmatrix}\texttt{t-}\\\texttt{ta}\end{smallmatrix}\right]$, state (q_0, q_1) is reached[1], while given $\left[\begin{smallmatrix}\texttt{t-c}\\\texttt{tat}\end{smallmatrix}\right]$, no state is reached. In particular, given a feasible constrained alignment satisfying R, for example $\left[\begin{smallmatrix}\texttt{ta-}\\\texttt{tac}\end{smallmatrix}\right]$, state (q_1, q_1), the final state of M in the example, is reached; the self loop on the final state is added for this purpose. As in a standard dynamic programming alignment algorithm, Arslan's algorithm iterates over index pairs (i_1, i_2). On each (i_1, i_2), a corresponding M_{i_1,i_2} is maintained. Each state (p, q) of M_{i_1,i_2} contains the score of an optimal alignment of $S_1[1..i_1]$ and $S_2[1..i_2]$ such that (p, q) can be reached if the alignment is given to M_{i_1,i_2} as input; such (p, q) is called *active* in [6]. If no such alignment exists for a state (p, q) of M_{i_1,i_2}, its corresponding score is set to $-\infty$ and it is not active. Given an additional input symbol $(a, b) \in \Sigma_{-} \times \Sigma_{-}$, the active states in M_{i_1,i_2} may change, and the scores are updated by adding the score of the current input symbol (edit operation). The resulting weighted automaton is denoted by $M_{i_1,i_2}^{(a,b)}$. The score updating rule in Arslan's algorithm is $M_{i_1,i_2} = \max\left\{M_{i_1-1,i_2}^{(S_1[i_1],\texttt{-})}, M_{i_1,i_2-1}^{(\texttt{-},S_2[i_2])}, M_{i_1-1,i_2-1}^{(S_1[i_1],S_2[i_2])}\right\}$, where the maximum operation applied on the three weighted automata yields a weighted automaton with each state scored by the maximum score from the corresponding states in the three automata. In the above example, for the score of an optimal constrained alignment of S_1 and S_2, one simply looks up the score stored in state (q_1, q_1) in $M_{n,n}$.

In [6], the score stored in the initial state of M_{i_1,i_2} is always 0. It can be observed, however, that this makes the algorithm in [6] applicable only to cases where there exists only one substring matching R in each of S_1 and S_2. For example, suppose that $R = \texttt{a}$, $S_1 = \texttt{attac}$ and $S_2 = \texttt{ttatc}$; two matches of R can be found in S_1. The optimal constrained alignment is $\left[\begin{smallmatrix}\texttt{a}&\texttt{t}&\texttt{t}&\texttt{a}&\texttt{-}&\texttt{c}\\\texttt{-}&\texttt{t}&\texttt{t}&\texttt{a}&\texttt{t}&\texttt{c}\end{smallmatrix}\right]$ with a score of 4, but the solution output by the algorithm in [6] would be $\left[\begin{smallmatrix}\texttt{-}&\texttt{-}&\texttt{a}&\texttt{t}&\texttt{t}&\texttt{a}&\texttt{c}\\\texttt{t}&\texttt{t}&\texttt{a}&\texttt{t}&\texttt{-}&\texttt{-}&\texttt{c}\end{smallmatrix}\right]$ with a score of 3, since the alignment before the satisfaction of the constraint is ignored if the initial state always has a score of 0. Certainly, once this point

[1] The transitions alone cannot achieve this; for example, on input $\left[\begin{smallmatrix}\texttt{t-}\\\texttt{ta}\end{smallmatrix}\right]$, M actually "dies" upon seeing $(\texttt{t}, \texttt{t})$ rather than keep moving to (q_0, q_1). This is achieved by some auxiliary manipulations in [6].

is noticed, it can be easily patched by, e.g., adding a self loop labeled by all possible edit operations to the initial state so that the initial state always keeps the score of an optimal alignment without any constraint.

To update the active states and the scores, all transitions in M are examined by the algorithm in [6]. Let V be the set of states in A. There can be up to $|\Sigma|$ transitions from a state of A to another. Hence the number of transitions in A can be up to $O(|\Sigma|\,|V|^2)$, and that in M can be up to $O(|\Sigma|^2|V|^4)$ since M is made by a product of A. In addition, $O(n)$ copies of M are maintained throughout in [6], hence the space complexity can be up to $O(|\Sigma|^2|V|^4n)$. In [6] it is not mentioned how to reconstruct the optimal alignment. The stated space complexity is for the computation of the optimal score. Here we make the dependence on the alphabet size explicit since in protein applications for which RECSA is originally formulated, $|\Sigma|^2$ is 400 which would significantly affect the efficiency.

The critical information in M is the scores on the states; storing the entire M is unnecessary. In this paper, we directly use a matrix to store scores. The matrix is implemented using a simple array and can be easily manipulated. Time and space complexity of our algorithm are respectively bounded by $O(|V|^3n^2)$ and $O(|V|^2n)$ in the worst case; we also mention how the optimal alignment can be reconstructed within this space bound. When $|V| = O(\log n)$, which is not remote since in general R is much shorter than the input sequences, we propose another algorithm to take advantage of operations on $O(\log n)$-lengthed data, which takes constant time in the standard unit-cost RAM model (or the operations can be implemented by using auxiliary tables of size $O(n)$; the details are omitted). The resulting time complexity is $O(|V|^2 \log |V|\, n^2)$ in this case. Although M is not used in our algorithm, it serves as a conceptual framework facilitating our presentation. From now on, we assume that the initial state of M has a self loop labeled with all possible edit operations.

In the next section we make some notions introduced in this section precise. In Sec. 3 our algorithms are formally presented. In the last section concluding remarks are given.

2 Preliminaries

For convenience throughout this paper we assume that $|S_1| = |S_2| = n$. An edit operation can be defined to be a symbol in $\Sigma_- \times \Sigma_-$, where $\Sigma_- = \Sigma \cup \{\texttt{-}\}$. An edit operation is written as either a pair (a, b) or a column vector $\left[\begin{smallmatrix} a \\ b \end{smallmatrix}\right]$ in this paper. Define $\rho : \Sigma_-^* \to \Sigma^*$ to be the "removing spaces" operator, e.g., $\rho(\texttt{a-tc-g}) = \texttt{atcg}$. A sequence $\mathcal{A} = \left[\begin{smallmatrix} a_1 \\ b_1 \end{smallmatrix} \cdots \begin{smallmatrix} a_m \\ b_m \end{smallmatrix}\right]$ of edit operations is called an alignment of S_1 and S_2 if $\rho(a_1 \cdots a_m) = S_1$ and $\rho(b_1 \cdots b_m) = S_2$. Let γ be a real-valued scoring function defined on sequences of edit operations satisfying

$$\gamma(\left[\begin{smallmatrix} a_1 \\ b_1 \end{smallmatrix} \cdots \begin{smallmatrix} a_m \\ b_m \end{smallmatrix}\right]) = \begin{cases} 0 & \text{if } m = 0 \text{, i.e., if it is an empty alignment;} \\ \gamma(\left[\begin{smallmatrix} a_1 \\ b_1 \end{smallmatrix} \cdots \begin{smallmatrix} a_{m-1} \\ b_{m-1} \end{smallmatrix}\right]) + \gamma(\left[\begin{smallmatrix} a_m \\ b_m \end{smallmatrix}\right]) & \text{otherwise.} \end{cases}$$

Let R be a regular expression over alphabet Σ. The goal of RECSA is to find a maximum-scored alignment $\mathcal{A} = \left[\begin{smallmatrix} a_1 \\ b_1 \end{smallmatrix} \cdots \begin{smallmatrix} a_m \\ b_m \end{smallmatrix}\right]$ of S_1 and S_2 such that there exist ℓ_1 and ℓ_2 with both $\rho(a_{\ell_1} \cdots a_{\ell_2})$ and $\rho(b_{\ell_1} \cdots b_{\ell_2})$ matching R.

Let $A = (Q, \Sigma, \delta, q_0, F)$ be an ε-free NFA equivalent to R, which can be constructed manually or by any established algorithm. Then $\delta(p, \varepsilon) = \delta(p, \text{-}) = \{p\}$ for all $p \in Q$. We also define $\delta(Q', a)$ to be $\bigcup_{p \in Q'} \delta(p, a)$, where $Q' \subseteq Q$ and $a \in \Sigma$. In this paper we use Q and V interchangeably. Following the notations in [6], $M = (Q^M, \Sigma^M, \delta^M, q_0^M, F^M)$, where $Q^M = Q \times Q$, $\Sigma^M = \Sigma_{\text{-}} \times \Sigma_{\text{-}}$, $q_0^M = (q_0, q_0)$ and $F^M = F \times F$. Transition function δ^M is defined as

$$\delta^M((p,q),(a,b)) = \begin{cases} \delta(p,a) \times \delta(q,b) & \text{if } (p,q) \notin F^M \cup \{(q_0,q_0)\}; \\ \delta(p,a) \times \delta(q,b) \cup \{(p,q)\} & \text{otherwise.} \end{cases}$$

Function δ^M can be naturally extended to be defined on the cross product of Q^M and the set of all possible alignments. A state (p, q) of M_{i_1,i_2} is active iff there exists some alignment $\mathcal{A}$ of $S_1[1..i_1]$ and $S_2[1..i_2]$ such that $(p, q) \in \delta^M(q_0^M, \mathcal{A})$. Each state (p, q) in M_{i_1,i_2} has a score $W_{i_1,i_2}(p, q)$ assigned to it. If (p, q) is active, then $W_{i_1,i_2}(p, q)$ is the score of an optimal alignment $\mathcal{A}$ of $S_1[1..i_1]$ and $S_2[1..i_2]$ such that $(p, q) \in \delta^M(q_0^M, \mathcal{A})$. Otherwise no such alignment exists and $W_{i_1,i_2}(p, q) = -\infty$.

3 The Algorithms

In this section we present two algorithms, the first for the general case and the second for the case that $|V| = O(\log n)$.

3.1 The General Case

Recall that automaton M_{i_1,i_2} accepts alignments of $S_1[1..i_1]$ and $S_2[1..i_2]$ satisfying the regular expression constraint. Also recall that once we have $W_{n,n}(p, q)$ for all $p, q \in Q$, the optimal score of aligning S_1 and S_2 with the constraint satisfied can be easily found by $\max_{(p,q) \in F \times F} W_{n,n}(p, q)$. Hence if we can compute all W_{i_1,i_2} then the problem is solved. There is a simple relationship among W_{i_1,i_2}, W_{i_1-1,i_2}, W_{i_1,i_2-1} and W_{i_1-1,i_2-1} which allows one to compute W_{i_1,i_2} based on the other three. Define

$$W_{i_1,i_2}^{(a,b)}(p,q) = \max_{(p',q'):(p,q) \in \delta^M((p',q'),(a,b))} W_{i_1,i_2}(p',q') + \gamma(a,b)$$

Then for $i_1, i_2 \geq 1$, the relationship can be stated as follows:

$$W_{i_1,i_2}(p,q) = \max\{W_{i_1-1,i_2}^{(S_1[i_1],\text{-})}(p,q), W_{i_1,i_2-1}^{(\text{-},S_2[i_2])}(p,q), W_{i_1-1,i_2-1}^{(S_1[i_1],S_2[i_2])}(p,q)\} \quad (1)$$

Let $\mathcal{A} = \left[\begin{smallmatrix} a_1 \\ b_1 \end{smallmatrix} \cdots \begin{smallmatrix} a_m \\ b_m \end{smallmatrix}\right]$ be the best alignment of $S_1[1..i_1]$ and $S_2[1..i_2]$ such that $(p, q) \in \delta^M(q_0^M, \mathcal{A})$ and that $(a_m, b_m) = (S_1[i_1], \text{-})$. It can be shown that $\gamma(\mathcal{A}) = W_{i_1-1,i_2}^{(S_1[i_1],\text{-})}(p, q)$, or $W_{i_1-1,i_2}^{(S_1[i_1],\text{-})}(p, q) = -\infty$ if such $\mathcal{A}$ does not exist. Indeed, if such $\mathcal{A}$ does not exist, then for any alignment $\mathcal{A}'$ of $S_1[1..i_1-1]$ and $S_2[1..i_2]$ with $(p',q') \in \delta^M(q_0^M, \mathcal{A}')$, or equivalently, for any $W_{i_1-1,i_2}(p',q') > -\infty$ with $\mathcal{A}'$ corresponding to this score, we must have $(p,q) \notin \delta^M((p',q'),(S_1[i_1],\text{-}))$, since otherwise $\mathcal{A}' \begin{bmatrix} S_1[i_1] \\ - \end{bmatrix}$ is an alignment of $S_1[1..i_1]$ and $S_2[1..i_2]$ such that $(p,q) \in \delta(q_0^M, \mathcal{A}' \begin{bmatrix} S_1[i_1] \\ - \end{bmatrix})$, a contradiction. Taking the convention that $\max \emptyset = -\infty$ it follows that $W_{i_1-1,i_2}^{(S_1[i_1],-)}(p,q) = -\infty$. Now if such $\mathcal{A}$ exists, let $\mathcal{A}' = \begin{bmatrix} a_1 & \cdots & a_{m-1} \\ b_1 & & b_{m-1} \end{bmatrix}$. There must exist $(p',q') \in \delta^M(q_0^M, \mathcal{A}')$ such that $(p,q) \in \delta^M((p',q'),(S_1[i_1],\text{-}))$. Clearly $\gamma(\mathcal{A}') = \max_{(p',q'):(p,q)\in\delta^M((p',q'),(S_1[i_1],\text{-}))} W_{i_1-1,i_2}(p',q')$ otherwise $\mathcal{A}$ cannot be the best alignment as stated. Therefore $\gamma(\mathcal{A}) = \gamma(\mathcal{A}') + \gamma(S_1[i_1],\text{-}) = W_{i_1-1,i_2}^{(S_1[i_1],-)}(p,q)$, as desired. The other two cases can be analyzed similarly, and hence (1) follows.

The algorithm in [6] is based on (1).[2] On each (i_1,i_2), weighted automaton M_{i_1,i_2}, whose transitions are the same as those in M, is constructed such that state (p,q) in M_{i_1,i_2} carries the score $W_{i_1,i_2}(p,q)$. Given M_{i_1-1,i_2}, $M_{i_1-1,i_2}^{(S_1[i_1],-)}$ can be computed, which is also a weighted automaton where state (p,q) carries the score $W_{i_1-1,i_2}^{(S_1[i_1],-)}(p,q)$; $M_{i_1,i_2-1}^{(-,S_2[i_2])}$ and $M_{i_1-1,i_2-1}^{(S_1[i_1],S_2[i_2])}$ are similarly defined. Then M_{i_1,i_2} is computed as

$$\max\{M_{i_1-1,i_2}^{(S_1[i_1],-)}, M_{i_1,i_2-1}^{(-,S_2[i_2])}, M_{i_1-1,i_2-1}^{(S_1[i_1],S_2[i_2])}\}$$

which is a weighted automaton where state (p,q) has score as defined in the right hand side of (1). To compute $M_{i_1-1,i_2}^{(S_1[i_1],-)}$ from M_{i_1-1,i_2}, in [6] each transition of M_{i_1-1,i_2} is examined a constant number of times so that for each (p,q) the score $W_{i_1-1,i_2}^{(S_1[i_1],-)}(p,q)$ can be computed. Automata $M_{i_1,i_2-1}^{(-,S_2[i_2])}$ and $M_{i_1-1,i_2-1}^{(S_1[i_1],S_2[i_2])}$ are computed in the same way, and M_{i_1,i_2} can then be computed easily.

As mentioned before, given two states q' and q, there can be up to $|\Sigma|$ transitions from q' to q. Let E be the set of "label-free" arcs in A, i.e. $(q',q) \in E$ iff $q \in \delta(q',a)$ for some $a \in \Sigma$. In particular, there can be at most one "label-free" arc from state q' to q. Hence there can be up to $O(|\Sigma||V|^2)$ transitions in A and $O(|\Sigma|^2|V|^4)$ transitions in M. Since the structure of M is duplicated on each (i_1,i_2), it takes $O(|\Sigma|^2|E|^2n^2)$ time in the worst case to do this duplication. Asymptotically this is equivalent to $O(|E|^2n^2)$, but in practice $|\Sigma|$ can affect the efficiency significantly, hence we make the dependence explicit. In [6] it is not mentioned how M is implemented. The time taken by the computation of M_{i_1,i_2} from M_{i_1-1,i_2}, M_{i_1,i_2-1} and M_{i_1-1,i_2-1} is also $O(|\Sigma|^2|E|^2n^2)$ in the worst case if M is implemented directly as an adjacency list, in which case the whole adjacency list needs to be examined not to miss any (p',q') with $(p,q) \in \delta^M((p',q'),(a,b))$.

Clearly it is not necessary to maintain the whole machine structure on each (i_1,i_2). The relevant information is the scores $W_{i_1,i_2}(p,q)$. On each (i_1,i_2) we use table L_{i_1,i_2} to store W_{i_1,i_2}. In addition, to handle the transitions we need not

[2] In [6], (1) is stated in terms of weighted automata M_{i_1,i_2} etc.; the algorithm is then based on the computation of $M_{i_1\ 1,i_2}^{(S_1[i_1],-)}$, etc., as we now describe.

use δ^M directly; the use of δ is sufficient. We construct a table T to represent δ: $T[q', a; q] = 1$ if $q \in \delta(q', a)$, and $T[q', a; q] = 0$ otherwise. This construction is easy, taking $O(|\Sigma||V|^2)$ time and space if automaton A is originally represented as an adjacency list. The computation of $L_{i_1,i_2}[p, q] = W_{i_1,i_2}(p, q)$ can then be improved, and is detailed as follows.

For (i_1, i_2) fixed, first we consider the computation of $W^{(S_1[i_1],S_2[i_2])}_{i_1-1,i_2-1}(p, q)$, which is

$$\max_{(p',q'):(p,q)\in\delta^M((p',q'),(S_1[i_1],S_2[i_2]))} W_{i_1-1,i_2-1}(p', q') + \gamma(S_1[i_1], S_2[i_2])$$

In terms of δ instead of δ^M, this can be written as

$$\begin{cases} \max_{p',q':p\in\delta(p',S_1[i_1]) \text{ and } q\in\delta(q',S_2[i_2])} W_{i_1-1,i_2-1}(p', q') + \gamma(S_1[i_1], S_2[i_2]) \\ \hfill \text{if } (p, q) \notin F^M \cup \{(q_0, q_0)\}; \\ \max_{p',q':p\in\delta(p',S_1[i_1]) \text{ and } q\in\delta(q',S_2[i_2]) \text{ or } (p',q')=(p,q)} W_{i_1-1,i_2-1}(p', q') + \\ \quad \gamma(S_1[i_1], S_2[i_2]) \hfill \text{otherwise.} \end{cases}$$

Hence clearly the use of δ suffices. This immediately enables us to compute $L^{(S_1[i_1],S_2[i_2])}_{i_1-1,i_2-1}[p, q]$, expected to equate $W^{(S_1[i_1],S_2[i_2])}_{i_1-1,i_2-1}(p, q)$ and initialized to be $-\infty$, for all $p, q \in V$ as follows, assuming the correctness of $L_{i_1-1,i_2-1}[p, q]$:

```
for $p, q \in V$ do
    for $p' \in V$ with $T[p', S_1[i_1]; p] = 1$ and $q' \in V$ with $T[q', S_2[i_2]; q] = 1$ do
        $L^{(S_1[i_1],S_2[i_2])}_{i_1-1,i_2-1}[p, q] \leftarrow$
                $\max\{L^{(S_1[i_1],S_2[i_2])}_{i_1-1,i_2-1}[p, q], L_{i_1-1,i_2-1}[p', q'] + \gamma(S_1[i_1], S_2[i_2])\}$;
    end for
end for
for $p, q \in V$ with $(p, q) \in (F \times F) \cup \{(q_0, q_0)\}$ do
    $L^{(S_1[i_1],S_2[i_2])}_{i_1-1,i_2-1}[p, q] \leftarrow$
                $\max\{L^{(S_1[i_1],S_2[i_2])}_{i_1-1,i_2-1}[p, q], L_{i_1-1,i_2-1}[p, q] + \gamma(S_1[i_1], S_2[i_2])\}$;
end for
```

However, this takes $\Theta(|V|^4)$ time which results in a worse algorithm than that in [6]. Upon rearranging,

```
for $p', p \in V$ with $T[p', S_1[i_1]; p] = 1$ do
    for $q', q \in V$ with $T[q', S_2[i_2]; q] = 1$ do
        $\cdots$                          ▷ The same inner loop code as above
    end for
end for
$\cdots$                          ▷ Deals with the case $(p, q) \in (F \times F) \cup \{(q_0, q_0)\}$
```

This now takes $|V|^2 + \sum_{p\in V} \deg(p) \times |V|^2 + O(|E|^2) = O(|E|\,|V|^2)$ time, where E is the set of "label-free" arcs obtained from A and $\deg(p)$ is the indegree of p in graph (V, E). This, again, does not improve the algorithm in [6]. We take a further step to separate the computation in two steps. Let L be a temporary table with all entries $L[p, q]$ initialized to be $-\infty$.

```
for $p', p \in V$ with $T[p', S_1[i_1]; p] = 1$ do
    for $q' \in V$ do
        $L[p, q'] \leftarrow \max\{L[p, q'], L_{i_1-1,i_2-1}[p', q'] + \gamma(S_1[i_1], S_2[i_2])\}$;
    end for
end for
for $q', q \in V$ with $T[q', S_2[i_2]; q] = 1$ do
    for $p \in V$ do
        $L^{(S_1[i_1],S_2[i_2])}_{i_1-1,i_2-1}[p, q] \leftarrow \max\{L^{(S_1[i_1],S_2[i_2])}_{i_1-1,i_2-1}[p, q], L[p, q']\}$;
    end for
end for
···                    ▷ Deals with the case $(p, q) \in (F \times F) \cup \{(q_0, q_0)\}$
```

To see the correctness, observe that in the first part $L[p, q']$ is computed to be

$$\max_{p':p\in\delta(p',S_1[i_1])} L_{i_1-1,i_2-1}[p', q'] + \gamma(S_1[i_1], S_2[i_2])$$

In the second part $L^{(S_1[i_1],S_2[i_2])}_{i_1-1,i_2-1}[p, q]$ is computed to be $\max_{q':q\in\delta(q',S_2[i_2])} L[p, q']$, which in turn is

$$\max_{p':p\in\delta(p',S_1[i_1]),q':q\in\delta(q',S_2[i_2])} L_{i_1-1,i_2-1}[p', q'] + \gamma(S_1[i_1], S_2[i_2])$$

as desired. The time taken by the above code segment is $O\left(\sum_{p\in V} \deg(p) \times |V|\right)$, which is $O(|E|\,|V|)$.

The computation of $L^{(S_1[i_1],-)}_{i_1-1,i_2}$ is easier. By definition, $W^{(S_1[i_1],-)}_{i_1-1,i_2}(p, q)$ is

$$\max_{(p',q'):(p,q)\in\delta^M((p',q'),(S_1[i_1],-))} W_{i_1-1,i_2}(p', q') + \gamma(S_1[i_1], -)$$

In terms of δ, this is simply

$$\begin{cases} \max_{p':p\in\delta(p',S_1[i_1])} W_{i_1-1,i_2}(p', q) + \gamma(S_1[i_1], -), & \text{if } (p, q) \notin F^M \cup \{(q_0, q_0)\}; \\ \max_{p':p\in\delta(p',S_1[i_1]) \text{ or } p'=p} W_{i_1-1,i_2}(p', q) + \gamma(S_1[i_1], -), & \text{otherwise.} \end{cases}$$

This can be implemented as follows:

```
for $p', p \in V$ with $T[p', S_1[i_1]; p] = 1$ do
    for $q \in V$ do
        $L^{(S_1[i_1],-)}_{i_1-1,i_2}[p, q] \leftarrow \max\{L^{(S_1[i_1],-)}_{i_1-1,i_2}[p, q], L_{i_1-1,i_2}[p', q] + \gamma(S_1[i_1], -)\}$;
    end for
end for
for $p, q \in V$ with $(p, q) \in (F \times F) \cup \{(q_0, q_0)\}$ do
    $L^{(S_1[i_1],-)}_{i_1-1,i_2}[p, q] \leftarrow \max\{L^{(S_1[i_1],-)}_{i_1-1,i_2}[p, q], L_{i_1-1,i_2}[p, q] + \gamma(S_1[i_1], -)\}$;
end for
```

which again takes $O(|E|\,|V|)$ time. The computation of $L^{(-,S_2[i_2])}_{i_1,i_2-1}$ is the same.

Now we consider the boundary cases when $i_1 = 0$ or $i_2 = 0$. It is clear that $W_{0,0}(q_0, q_0) = 0$ and that $W_{0,0}(p, q) = -\infty$ for $(p, q) \neq q_0^M$ since the only alignment of $S_1[1..0] = \varepsilon$ and $S_2[1..0] = \varepsilon$ is an empty alignment, with a score of 0 by the definition of γ, and from q_0^M we can only reach q_0^M given an empty alignment as input to M. In addition, it can be seen that

$$W_{i_1,i_2}(p,q) = \begin{cases} W_{i_1-1,0}^{(S_1[i_1],\texttt{-})}(p,q) & \text{if } i_1 > 0 \text{ and } i_2 = 0, \\ W_{0,i_2-1}^{(\texttt{-},S_2[i_2])}(p,q) & \text{if } i_2 > 0 \text{ and } i_1 = 0, \end{cases}$$

which can be implemented as mentioned before. In [6], the weights for all the states $(p,q) \neq (q_0,q_0)$ in $M_{i_1,0}$ for $i_1 > 0$ or in M_{0,i_2} for $i_2 > 0$ are all set to $-\infty$, which, without appropriate assumptions on γ (e.g. triangle inequality; no such assumption is made in [6]), would lead to a failure of considering possible optimal constrained alignments like $\left[\begin{smallmatrix}\texttt{c} & \texttt{-}\\ \texttt{-} & \texttt{t}\end{smallmatrix}\right]$ where $R = \texttt{c} + \texttt{t}$, $S_1 = \texttt{c}$ and $S_2 = \texttt{t}$.

Now that $L_{i_1-1,i_2-1}^{(S_1[i_1],S_2[i_2])}$, $L_{i_1-1,i_2}^{(S_1[i_1],\texttt{-})}$ and $L_{i_1,i_2-1}^{(\texttt{-},S_2[i_2])}$ can be computed, by (1) it is easy to compute L_{i_1,i_2} in $O(|V|^2)$ time for a fixed (i_1,i_2). Graph (V,E) is connected if considered undirected, hence $|V| = O(|E|)$. Therefore the total time of our algorithm is $O(|V|\,|E|n^2)$.

To reconstruct the optimal constrained alignment, first recall that in a feasible constrained alignment $\mathcal{A} = \left[\begin{smallmatrix}a_1 & \cdots & a_m\\ b_1 & \cdots & b_m\end{smallmatrix}\right]$ of S_1 and S_2, there exist ℓ_1 and ℓ_2 such that $\delta(q_0, \rho(a_{\ell_1}\cdots a_{\ell_2})) \cap F \neq \emptyset$ and $\delta(q_0, \rho(b_{\ell_1}\cdots b_{\ell_2})) \cap F \neq \emptyset$. We may regard $\mathcal{A}$ as composed of three parts: an optimal unconstrained alignment of $\rho(a_1\cdots a_{\ell_1-1})$ and $\rho(b_1\cdots b_{\ell_1-1})$, that of $\rho(a_{\ell_1}\cdots a_{\ell_2})$ and $\rho(b_{\ell_1}\cdots b_{\ell_2})$ and that of $\rho(a_{\ell_2+1}\cdots a_m)$ and $\rho(b_{\ell_2+1}\cdots b_m)$. Let $j_1 = |\rho(a_1\cdots a_{\ell_1-1})|$, $j_1' = |\rho(a_1\cdots a_{\ell_2})|$, $j_2 = |\rho(b_1\cdots b_{\ell_1-1})|$ and $j_2' = |\rho(b_1\cdots b_{\ell_2})|$. Clearly if we know j_1, j_1', j_2 and j_2' then the constrained alignment $\mathcal{A}$ can be constructed in $O(n)$ space by using Hirschberg's celebrated divide-and-conquer algorithm [10] to align $S_1[1..j_1]$ and $S_2[1..j_2]$, align $S_1[j_1+1..j_1']$ and $S_2[j_2+1..j_2']$, and align $S_1[j_1'+1..n]$ and $S_2[j_2'+1..n]$. For this purpose we compute matrices $\eta_1^{i_1,i_2}$ and $\eta_2^{i_1,i_2}$ for each (i_1,i_2). Let $\mathcal{A} = \left[\begin{smallmatrix}a_1 & \cdots & a_m\\ b_1 & \cdots & b_m\end{smallmatrix}\right]$ be the alignment implied by $L_{i_1,i_2}[p,q]$. If $(p,q) \in F \times F$, then $\eta_1^{i_1,i_2}[p,q]$ stores (j_1,j_2) and $\eta_2^{i_1,i_2}[p,q]$ stores (j_1',j_2') such that $p \in \delta(q_0, S_1[j_1+1..j_1'])$ and $q \in \delta(q_0, S_2[j_2+1..j_2'])$, where $S_1[j_1+1..j_1'] = \rho(a_{\ell_1}\cdots a_{\ell_2})$ and $S_2[j_2+1..j_2'] = \rho(b_{\ell_1}\cdots b_{\ell_2})$ for some ℓ_1 and ℓ_2. If $(p,q) \notin F \times F$, then $\eta_1^{i_1,i_2}[p,q]$ stores (j_1,j_2) such that $p \in \delta(q_0, S_1[j_1+1..i_1])$ and $q \in \delta(q_0, S_2[j_1+1..i_2])$, where $S_1[j_1+1..i_1] = \rho(a_{\ell_1}\cdots a_m)$ and $S_2[j_2+1..i_2] = \rho(b_{\ell_1}\cdots b_m)$ for some ℓ_1; $\eta_2^{i_1,i_2}[p,q]$ simply stores (i_1,i_2). Note that when computing L_{i_1,i_2} we can determine the value of (a_m,b_m); for example $(a_m,b_m) = (S_1[i_1],\texttt{-})$ if $L_{i_1,i_2}[p,q] = L_{i_1-1,i_2}^{(S_1[i_1],\texttt{-})}[p,q]$.

To see how to compute these values we assume that $(a_m,b_m) = (S_1[i_1],\texttt{-})$; the other two cases are similar. Let $(p',q) \in \delta(q_0^M, \left[\begin{smallmatrix}a_1 & \cdots & a_{m-1}\\ b_1 & \cdots & b_{m-1}\end{smallmatrix}\right])$ be such that $L_{i_1-1,i_2}^{(S_1[i_1],\texttt{-})}[p,q] = L_{i_1-1,i_2}[p',q] + \gamma(S_1[i_1],\texttt{-})$; (p',q) can be determined during the computation of $L_{i_1-1,i_2}^{(S_1[i_1],\texttt{-})}[p,q]$. Consider first that $(p,q) \notin F \times F$. Then $\eta_2^{i_1,i_2}[p,q] = (i_1,i_2)$. If $(p',q) = (q_0,q_0)$, then clearly we can set $\eta_1^{i_1,i_2}[p,q] = (i_1-1,i_2)$. If $(p',q) \neq (q_0,q_0)$ then we can set $\eta_1^{i_1,i_2}[p,q] = \eta_1^{i_1-1,i_2}[p',q]$ since if $(j_1,j_2) = \eta_1^{i_1-1,i_2}[p',q]$ then $p' \in \delta(q_0, S_1[j_1+1..i_1-1])$ and $q \in \delta(q_0, S_2[j_2+1..i_2])$ where $S_1[j_1+1..i_1-1] = \rho(a_{\ell_1}\cdots a_{m-1})$ and $S_2[j_2+1..i_2] = \rho(b_{\ell_1}\cdots b_{m-1})$ for some ℓ_1. Since $(p,q) \neq (q_0,q_0)$ and $(p,q) \notin F \times F$, $(p,q) \in \delta^M((p',q),(S_1[i_1],\text{-}))$ iff $p \in \delta(p',S_1[i_1])$. Hence $p \in \delta(q_0,S_1[j_1+1..i_1])$ and $q \in \delta(q_0,S_2[j_2+1..i_2])$ with $S_1[j_1+1..i_1] = \rho(a_{\ell_1}\cdots a_m)$ and $S_2[j_2+1..i_2] = \rho(b_{\ell_1}\cdots b_m)$. Now consider $(p,q) \in F \times F$. When $(p,q) = (p',q)$, then we simply set $\eta_1^{i_1,i_2}[p,q] = \eta_1^{i_1-1,i_2}[p',q]$ and $\eta_2^{i_1,i_2}[p,q] = \eta_2^{i_1-1,i_2}[p',q]$. When $(p,q) \neq (p',q)$, then $\eta_1^{i_1,i_2}[p,q]$ is set in the same way as before, while $\eta_2^{i_1,i_2}[p,q]$ is set to (i_1,i_2).

Therefore $\eta_1^{i_1,i_2}$ and $\eta_2^{i_1,i_2}$ can be set without difficulty. For the boundary case, all entries in $\eta_1^{0,0}$ and $\eta_2^{0,0}$ are simply set to $(0,0)$. With the knowledge of $\eta_1^{n,n}[p,q]$ and $\eta_2^{n,n}[p,q]$ where $(p,q) = \arg\max_{(p',q') \in F \times F}\{L_{n,n}[p',q']\}$, an optimal constrained alignment of S_1 and S_2 can then be constructed in linear space.

When computing L_{i_1,i_2} all those $L_{i'_1,i'_2}$ with $i'_1 \leq i_1 - 2$ are not necessary. Similar arguments are also applicable to $\eta_1^{i_1,i_2}$ and $\eta_2^{i_1,i_2}$. Each of these matrices has $O(|V|^2)$ entries. Given $\eta_1^{n,n}$ and $\eta_2^{n,n}$, the reconstruction of an optimal constrained alignment takes $O(n)$ space (and $O(n^2)$ time so the time complexity stated before is not affected). Therefore the space complexity of our algorithm is $O(|V|^2\, n)$. On the other hand, since $2n$ copies of weighted automata are maintained, giving only the score of an optimal constrained alignment, the space complexity of the algorithm in [6] is $O(|\Sigma|^2|E|^2 n)$, which is less satisfactory.

3.2 An Algorithm for Short Regular Expression Constraints

In general R is much shorter than S_1 and S_2. The number of states in the automaton A equivalent to R may hence also be small relative to n. Here we present another algorithm which can be combined with the algorithm presented in the last subsection to yield a more efficient algorithm when $|V| = O(\log n)$. Under this assumption, we represent each $T[q,a]$ (regarded as a vector containing entries $T[q,a;p]$ for all $p \in V$) by a bit vector of length $|V|$; such a bit vector now takes $O(1)$ space to store. For brevity we only show the computation of $L_{i_1-1,i_2}^{(S_1[i_1],\text{-})}$. The other cases can be handled similarly.

```
for $q \in V$ do
    Compute array $X$ such that $L_{i_1-1,i_2}[X[j],q] \geq L_{i_1-1,i_2}[X[j+1],q]$;
    Initialize bit vector $Y$ and array $Z$, both with length $|V|$, to be all 0;
    for $j = 1$ to $|V|$ do
        $p' \leftarrow X[j]$;
        $T' \leftarrow T[p',S_1[i_1]] \otimes \bar{Y}$;    ▷ $T'$ is a temporary bit vector, $\otimes$ is the bitwise AND operator and $\bar{Y}$ is the bitwise complement of $Y$
        $t \leftarrow \|T'\|_1$;    ▷ The number of bits with value 1 in $T'$
        for $c \leftarrow 1$ to $t$ do
            $Z[\lambda[T']] \leftarrow p'$;    ▷ $\lambda[T']$ is the position of the leftmost bit with value 1 in $T'$
            Set bit $\lambda[T']$ of $T'$ to 0;
        end for
        $Y \leftarrow Y \oplus T'$;
    end for
    for $p \in V$ with $Z[p] \neq 0$ do
```

```
15:         $L^{(S_1[i_1],-)}_{i_1-1,i_2}[p,q] \leftarrow L_{i_1-1,i_2}[Z[p],q] + \gamma(S_1[i_1],\text{-})$;
16:     end for
17: end for
```

Now we discuss the correctness of the above code segment. For convenience bit vectors are also treated as sets in our discussion that follows. Fix state $q \in V$ of the outmost loop. Entry $Z[p]$ of array Z, if $Z[p] \neq 0$, is used to keep

$$\arg\max_{p'}\{L_{i_1-1,i_2}[p',q] : p \in \delta(p', S_1[i_1])\}$$

On the jth execution of line 12, vector Y stores the set $\bigcup_{j'=1}^{j} \delta(X[j'], S_1[i_1])$ of states, i.e., $Y[p] = 1$ iff $p \in \delta(X[j'], S_1[i_1])$ for some $j' \leq j$. Vector T' when initialized in line 6 keeps the set of states in $\delta(X[j], S_1[i_1])$ that are not in Y, hence if $j' < j$, we have that $p \in T'$ implies $p \notin \delta(X[j'], S_1[i_1])$. In particular, since $L_{i_1-1,i_2}[X[j],q] \geq L_{i_1-1,i_2}[X[j+1],q]$, $p \in T'$ implies that

$$\begin{aligned} L_{i_1-1,i_2}[X[j],q] &= \max\{L_{i_1-1,i_2}[X[j'],q] : j \leq j' \leq |V| \text{ and } p \in \delta(X[j'], S_1[i_1])\} \\ &= \max\{L_{i_1-1,i_2}[X[j'],q] : 1 \leq j' \leq |V| \text{ and } p \in \delta(X[j'], S_1[i_1])\} \\ &= \max\{L_{i_1-1,i_2}[p',q] : p \in \delta(p', S_1[i_1])\} \end{aligned}$$

If for some p, it does not belong to any T' throughout the computation, then no state p' with $p \in \delta(p', S_1[i_1])$ exists; leaving $Z[p] = 0$ in this case is correct. As lines 7–11 implements statement "For each $p \in T'$ set $Z[p] \leftarrow X[j]$," it is now clear that Z is correctly computed. The correctness of the above code segment then follows.

We now analyze the time complexity of the above code segment. Line 2 takes $O(|V|^2 \log |V|)$ total time. Line 3 takes $O(|V|^2)$ total time. Lines 5–7 take $O(1)$ time for each iteration, hence take $O(|V|^2)$ time in total. For a fixed q, line 8 is executed $O(|V|)$ times since the T' for a specific value of j is disjoint from all those corresponding to other values of j. Hence the total time taken by lines 8–11 is $O(|V|^2)$. Lines 12 and 14–16 also take $O(|V|^2)$ total time. Put this code segment into the complete algorithm, the overall time complexity for solving RECSA is $O(|V|^2 \log |V|\, n^2)$. Therefore by combining the algorithm presented in the last subsection, we can solve RECSA in time $O(\min\{|V| \log |V|, |E|\} |V|\, n^2)$.

We summarize our results as the following theorem.

Theorem 1. *RECSA can be solved in $O(|V|\,|E| n^2)$ time and $O(|V|^2 n)$ space. If $|V| = O(\log n)$, then it can be solved in $O(\min\{|E|, |V| \log |V|\}\, |V|\, n^2)$ time and $O(|V|^2 n)$ space.*

4 Conclusion and Discussion

In this paper we proposed two efficient algorithms solving the regular expression constrained sequence alignment problem. The time and space complexity of our first algorithm are respectively $O(|V|\,|E|\, n^2)$ and $O(|V|^2\, n)$. When $|V| = O(\log n)$ we are able to solve RECSA in $O(\min\{|E|, |V| \log |V|\}\, |V|\, n^2)$

time. These compare favorably with the time and space complexity in [6] which are respectively $O(rn^2)$ and $O(rn)$, where r is the number of transitions in M and $r = O(|\Sigma|^2\,|E|^2)$. When aligning protein sequences a factor of $|\Sigma|^2$ may impair the efficiency seriously. In addition to the asymptotic improvements, our algorithms are also easy to implement and due to the use of simple arrays less redundancy is involved in practice. In [6] it is not stated how to reconstruct an optimal constrained alignment. We propose a space efficient method to reconstruct it without affecting the above stated space complexity.

In practice it is important to have an algorithm to align multiple sequences with the given regular expression constraint satisfied. In [6] it is suggested to extend the structure of weighted finite automata to support multiple sequences. However the size of such automata grows exponentially with the number of sequences, becoming an exponential multiplicative factor to the exponential time complexity of a multiple sequence alignment procedure. Hence a reasonable progressive algorithm of RECSA for multiple sequences is still in demand.

Acknowledgement

The authors thank the anonymous referees for their useful comments on improvements of the presentation of this paper.

References

1. Jiang, T., Xu, Y., Zhang, M.Q., eds.: Current Topics in Computational Molecular Biology. The MIT Press (2002)
2. Tang, C.Y., Lu, C.L., Chang, M.D.T., Tsai, Y.T., Sun, Y.J., Chao, K.M., Chang, J.M., Chiou, Y.H., Wu, C.M., Chang, H.T., Chou, W.I.: Constrained sequence alignment tool development and its application to rnase family alignment. Journal of Bioinfomatics and Computational Biology **1** (2003) 267–287
3. Chin, F.Y.L., Ho, N.L., Lam, T.W., Wong, P.W.H.: Efficient constrained multiple sequence alignment with performance guarantee. Journal of Bioinformatics and Computational Biology **3**(1) (2005) 1–18
4. Tsai, Y.T., Huang, Y.P., Yu, C.T., Lu, C.L.: Music: A tool for multiple sequence alignment with constraints. Bioinformatics **20** (2004) 2309–2311
5. Lu, C.L., Huang, Y.P.: A memory-efficient algorithm for multiple sequence alignment with constraints. Bioinformatics **21**(1) (2005) 20–30
6. Arslan, A.N.: Regular expression constrained sequence alignment. In Apostolico, A., Crochemore, M., Park, K., eds.: CPM 2005. Volume 3537 of Lecture Notes in Computer Science., Springer (2005) 322–333
7. Hulo, N., Sigrist, C.J.A., Saux, V.L., Langendijk-Genevaux, P.S., Bordoli, L., Gattiker, A., Castro, E.D., Bucher, P., Bairoch, A.: Recent improvements to the prosite database. Nucleic Acids Res. **32** (2004) 134–137
8. Faisst, S., Meyer, S.: Compilation of vertebrate-encoded transcription factors. Nucleic Acids Research **20**(1) (1992) 3–26
9. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation. 2nd edn. Addison-Wesley (2001)
10. Hirschberg, D.S.: A linear space algorithm for computing maximal common subsequences. Comm. ACM **18** (1975) 341–343

Large Scale Matching for Position Weight Matrices

Aude Liefooghe, Hélène Touzet, and Jean-Stéphane Varré

LIFL – UMR CNRS 8022 – Université Lille 1
59 655 Villeneuve d'Ascq cedex, France

Abstract. This paper addresses the problem of multiple pattern matching for motifs encoded by Position Weight Matrices. We first present an algorithm that uses a multi-index table to preprocess the set of motifs, allowing a dramatically decrease of computation time. We then show how to take benefit from simlar motifs to prevent useless computations.

1 Introduction

Weighted pattern matching has attracted a lot of interest recently. We deal here more particularly with the problem of finding weighted patterns in a non-weighted text. This situation occurs in computational biology for example, when searching for transcription factor binding sites modelled by *Position Weight Matrices* [12]. Large public databases of PWMs are available, such as Transfac [14] and Jaspar [7]. The usage of PWMs goes with global bioinformatics strategies that help to elucidate regulation mechanisms: comparative genomics, identification of over-represented motifs, identification of correlation between binding sites,... Most of these methods rely on long-range exhaustive scannings for large predefined sets of PWMs (see [13, 6, 3] for recent examples). Brute force approach is no longer appropriate for this task, and there is a need for efficient algorithms.

In this article, we address the problem of searching for PWM occurrences in long sequences with a large initial set of PWMs. In Section 3, we first present a general algorithm for that question with no further assumption on the set of matrices. The foundation of this algorithm is to preprocess the matrices and to store the scores in a multi-index table. We show how to build and to use this table in a optimal way with respect to the data set.

In Section 4, we investigate the multiple PWM matching problem for *similar* matrices. The idea is to take benefit from the mutual information content in the set of matrices, and eliminate useless computations. This gives rise to two filtering algorithms. Filtering may be lossless or approximate.

2 Position Weight Matrices

The objects we deal with are weighted patterns: each character symbol is assigned a score dependent of the position in the pattern. Such weighted patterns

M. Lewenstein and G. Valiente (Eds.): CPM 2006, LNCS 4009, pp. 401–412, 2006.

are indeed simple non-branching weighted automata. They are known as *Position Weight Matrices* (PWMs for short), or equivalently *Position Specific Scoring Matrices* (PSSMs) in computational biology. We adopt this terminology here.

Definition 1 (Position Weight Matrix). *Let Σ be a finite alphabet and let n be a positive natural number. A* Position Weight Matrix *(for short PWM) of length n is a matrix indexed by $\{1,\dots,n\} \times \Sigma$ with integer coefficient values.*

Note that we consider here Position Weight Matrices composed of integer coefficients. In full generality, PWMs can contain real coefficients when they are defined as entropy matrices or log odd matrices. But in practice such matrices are rounded to integer matrices or equivalently decimal matrices.

Given a PWM M, we write $M(i,x)$ for the coefficient indexed by i in $\{1,\dots,n\}$ and x in Σ. We also use *slices* of PWMs: $M[i..j]$ denotes the submatrix of M obtained by selecting only positions from i to j, for all character symbols. By convention, if $i > j$, then $M[i..j]$ is an empty matrix.

Definition 2 (Score). *Let M be a PWM of length n and let u be a string of Σ^n. The score of M on u is defined as*

$$\texttt{Score}(u,M) = \sum_{i=1}^{n} M(i,u_i)$$

where u_i denotes the character symbol at position i in u. Given a sequence S of length greater than n, we write $\texttt{Score}(S,i,M)$ for the score of M on the factor of length n beginning at position i in S.

Definition 3 (PWM matching problem). *Let S be a sequence, M be a PWM and α be a score cutoff. The PWM matching problem consists in finding all positions i in S such that $\texttt{Score}(S,i,M)$ is greater than α.*

The score distribution of the PWM can be assigned a P-value function, which gives the statistical significance of an occurrence according to its score [2, 11]. In this article, we use the P-value as an internal parameter to speed up the computation of the score. This is an unusual application. To give a formal definition of the P-value, one needs to model the background sequence properly. We assume here that the positions in the sequence are independently and identically distributed. All results can be extended to more sophisticate random sources, such as Markov models [5]. But we think that in this context the gain is poor.

Definition 4 (P-value). *Let M be a PWM and let α be a score threshold. The P-value $\texttt{Pvalue}(M,\alpha)$ is the probability that the background model can achieve a score equal to or greater than α.*

In other words, the P-value is the proportion of strings (with respect to the background model) whose score is greater than the cutoff. The P-value can be computed with time complexity linear in the length of the PWM. The algorithm uses probability generating functions or directly dynamic programming. The P-value for the i^{th} column of the matrix is deduced from the previous position $i-1$ as follows.

$$\texttt{Pvalue}(M[1..0], \alpha) = \begin{cases} 1 & \text{if } \alpha \leq 0 \\ 0 & \text{otherwise} \end{cases}$$

$$\texttt{Pvalue}(M[1..i]), \alpha) = \frac{1}{\sigma} \sum_{x \in \Sigma} \texttt{Pvalue}(M[1..i-1], \alpha - M(i, x)) \tag{1}$$

Throughout this paper, σ denotes the cardinality of the alphabet Σ.

3 The Multiple PWM Matching Problem

We present a simple algorithm for the PWM matching problem in presence of a large set of PWMs. This algorithm takes advantage of the fact that PWMs are permanent objects that are known in advance. So it is likely to preprocess PWMs to speed up the search process. We combine two complementary strategies: pruning the columns of the PWMs and precomputing partial scores in a multi-index table. We first describe the method for a single matrix. The extension to a larger number of PWMs is straightforward and is presented in the third part of this section.

3.1 Pruning the PWM

The first idea comes from the structure of the PWMs and of the underlying motif. The score depends on the entirety of the PWM. But in most cases the information contained in a subset of the columns is sufficient to conclude to the absence of occurrence without inspecting all columns of the matrix. One can anticipate from partial scores that the final score will be lower than the predetermined cutoff. For example, for the PWM M of Figure 1 associated to the score threshold $\alpha = 1$, every word starting with the prefix *bbb* cannot be an occurence for M whatever the remaining part of the word is. The score of *bbb* is -3. It cannot be compensated for by the two last columns, whose maximal cumulated score is 2. In this context, it is not necessary to inspect the last two columns of M to be able to conclude that there is no occurrence of the PWM.

$$\begin{array}{c|ccccc|} a & 2 & 1 & 0 & -2 & 1 \\ b & -2 & -1 & 0 & 1 & 0 \end{array}$$

Fig. 1. Example of PWM

For each position of the PWM, we consider the minimal score that should be reached to be able to have an occurrence of the PWM. We call this threshold the *Greater lower bound* score.

Definition 5 (Greater lower bound). *Let M be a PWM of length n with a score threshold α. Let i be a position in $\{1, \ldots, n\}$. The* greater lower bound *for i in M, denoted $\texttt{GLB}(M, i, \alpha)$, is the largest value v such that for each $u \in \Sigma^{n-i}$, $v + \texttt{Score}(u, M[i+1..n]) < \alpha$.*

Lemma 1. *Let M be a PWM of length n with a score threshold α. Let i be a position in $\{1, \dots, n\}$. The greater lower bound for i in M equals*

$$\mathtt{GLB}(M, i, \alpha) = \alpha - \sum_{j=i+1}^{n} \max\{M(j, x);\ x \in \Sigma\}.$$

Proof. By definition 2, the score system is additive and the score of $M[i+1..n]$ is majored by $\sum_{j=i+1}^{n} \max\{M(j, x);\ x \in \Sigma\}$.

According to Definition 4, we are able to measure the probability of the set of prefixes of length i that can lead to an occurrence of M: it equals

$$\mathtt{Pvalue}(M[1..i], \mathtt{GLB}(M, i, \alpha)).$$

It means that when searching for occurrences of M, the probability to have to go beyond the position i is $\mathtt{Pvalue}(M[1..i], \mathtt{GLB}(M, i, \alpha))$. This probability decreases with increasing values of i and increasing values of α.

PWM Matching Problem with Pruning

Input: a PWM (M, α) of length n, a string u of length n
Output: is $\mathtt{Score}(u, M)$ greater than α ?

```
$s \leftarrow 0$
for $i \in \{1, \dots n\}$ and $s \geq \mathtt{GLB}(M, i-1, \alpha)$ do
    $s \leftarrow s + M(i, u_i)$
return $(s \geq \alpha)$
```

For a given matrix M of length n, the average number of visited columns is

$$\sum_{i=0}^{n-1} \mathtt{Pvalue}(M[1..i], \mathtt{GLB}(M, i, \alpha)).$$

Figure 2 shows the gain in the computation time yielded by the application of the pruning rule on a large number of PWMs retrieved from Transfac [14] and Jaspar [7] databases.

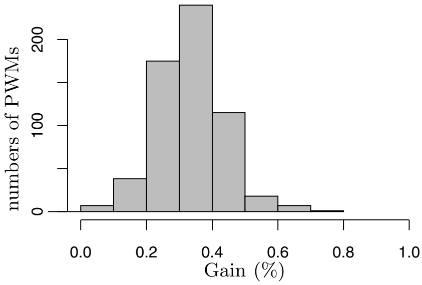


Fig. 2. Evaluation of the impact of the pruning strategy on Jaspar and Transfac vertebrate matrices (602 matrices in total). The horizontal axis indicates the mean runtime gain compared with the classical brute-force algorithm (in percentage) for one matrice. The vertical axis indicates the number of matrices for each value. The average gain, for the whole set of matrices, is 33.85%.

3.2 Preprocessing of the PWM

We now show that it is possible to further speed up the computation of the score using an index table. Let M be a position weight matrix of length n on the alphabet Σ. This matrix induces a mapping from Σ^n to $\mathbb{Z}$. So its score distribution is determined by σ^n values. Of course, for real-life PWMs, n is too large to store exhaustively all scores on Σ^n. The solution we propose here consists in dividing the matrix into a set of smaller tractable submatrices $\{M_1, \ldots, M_l\}$, that are made of contigous slices of M. For each such small submatrix, it is possible to precompute all possible scores for all possible strings and store them exhaustively in an index table. The problem of designing this series of index tables is thus defined by the following parameters:

- the number l of submatrices,
- an increasing sequence of positions $i_1 = 1, \ldots, i_k, \ldots, i_{l+1} = n + 1$.

The sequence of positions determines the bounds of the submatrices: $M_k = M[i_k..i_{k+1} - 1]$. Each submatrix M_k is associated an index table T_k whose size is $\sigma^{i_{k+1}-i_k}$: for each word $u \in \Sigma^{i_{k+1}-i_k}$, we define $T_k(h(u)) = \texttt{Score}(u, M_k)$, where h is a hash function associating to each word of Σ^* a natural number in lexicographic ordering in a classical way. The final result is obtained using the additivity property of the definition 2. For each $u \in \Sigma^n$,

$$\texttt{Score}(u, M) = \sum_{k=1}^{l} T_k(h(u(i_k..i_{k+1} - 1))) \qquad (2)$$

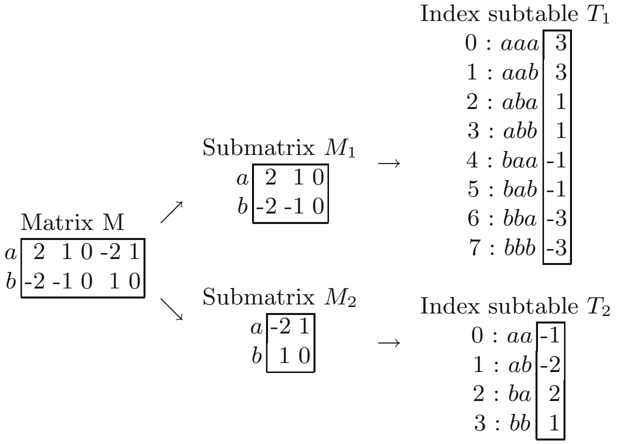
A full example of the construction of the index table for a PWM of length 5 is visible in Figure 3.

3.3 Construction of the Optimal Index Table for an Arbitrary Set of PWMs

The construction of the index table for a single matrix can easily be modified to handle a set of PWMs: each cell of the index table T_k is a vector that contains the partial score of each PWM. More formally, let $\mathcal{M}$ be a set of PWMs and let t_n be the number of matrices of length n in $\mathcal{M}$. The index table is defined as previously by

- the number l of submatrices,
- an increasing sequence of positions $i_1 = 1, \ldots, i_k, \ldots i_{l+1} = n + 1$.

The difference is that the size of each subtable T_k is $\sigma^{i_{k+1}-i_k} \sum_{y=i_k}^{i_{k+1}-1} t_y$. The index table can also be combined with the pruning strategy described in Section 3.1: pruning applies at each position corresponding to borders of subtables. So the objective is to choose the number and the size of the subtables that minimize the average runtime taking into account the structure of the PWMs. For that purpose, we use Definition 5 and Lemma 1.



Matrix M

$$\begin{array}{c|ccccc|} a & 2 & 1 & 0 & -2 & 1 \\ b & -2 & -1 & 0 & 1 & 0 \end{array}$$

Submatrix M_1

$$\begin{array}{c|ccc|} a & 2 & 1 & 0 \\ b & -2 & -1 & 0 \end{array}$$

Submatrix M_2

$$\begin{array}{c|cc|} a & -2 & 1 \\ b & 1 & 0 \end{array}$$

Index subtable T_1

0 : *aaa*	3
1 : *aab*	3
2 : *aba*	1
3 : *abb*	1
4 : *baa*	-1
5 : *bab*	-1
6 : *bba*	-3
7 : *bbb*	-3

Index subtable T_2

0 : *aa*	-1
1 : *ab*	-2
2 : *ba*	2
3 : *bb*	1

Fig. 3. Example of construction of the index table for a PWM of length 5, with alphabet $\Sigma = \{a, b\}$. The matrix M is divided into two submatrices, M_1 of length 3 and M_2 of length 2. The table T_1 stores all precomputed scores for M_1 on $\{a, b\}^3$, and, similarly all scores for M_2 on $\{a, b\}^2$ are stored in T_2. The correspondence between $\{a, b\}$ and the positions in the tables is performed with the hash function h. For instance, the final score of *abbab* is obtained as the sum of $T_1(h(abb)) = T_1(3) = 1$ and $T_2(h(ab)) = T_2(1) = -2$: it is -1.

Given e the maximal memory space capacity, we write $\phi(j, e)$ for the average number of accesses for positions $[1..j]$.

$$\phi(0, e) = \begin{cases} 0, & \text{if } e \geq 0 \\ +\infty, & \text{if } e < 0 \end{cases} \tag{3}$$

$$\phi(j, e) = \min_{0 \leq i < j} \{\phi(i, e - \sigma^{j-i} \sum_{y=i+1}^{j} t_y) + \sum_{M \in \mathcal{M}} \texttt{Pvalue}(M[1..i], \texttt{GLB}(M, i, \alpha))\}$$

The last case corresponds to the main case, with the creation of an index table for the slice $[i + 1..j]$. In this equation, by convention if the length of M is smaller than i, then $\texttt{Pvalue}(M[1..i], \texttt{GLB}(M, i, \alpha))$ equals 0. The allocated memory space for this new table is $\sigma^{j-i} \sum_{y=i+1}^{j} t_y$ and, according to the analysis of subsection 3.1, it requires $\texttt{Pvalue}(M[1..i], \texttt{GLB}(M, i, \alpha))$ accesses in average. The final value is obtained with $\phi(m, e)$, where m is the length of the longest PWM of $\mathcal{M}$ and e is the available memory space. ϕ can be computed by dynamic programming, and the associated optimal structure is then recovered by tracing back. The following pseudo-code algorithm recapitulates the steps of the preprocessing of the set of PWMS. Figure 4 gives an example of the optimal index table for the set of PWMs coming from the databases Transfac vertebrates and Jaspar.

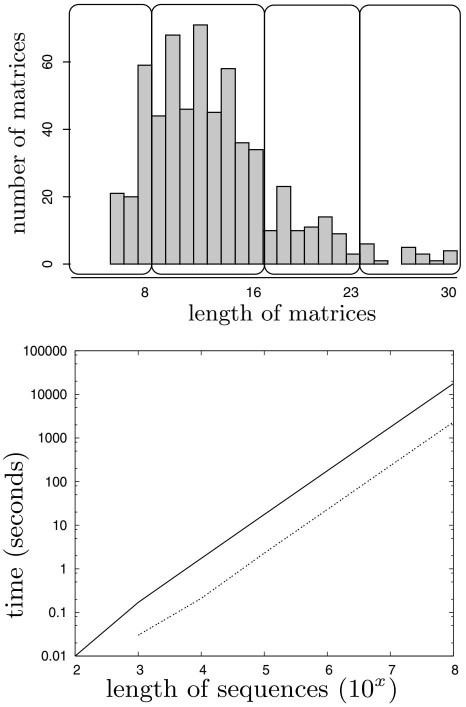


Fig. 4. The histogram represents the distribution of sizes of Transfac and Jaspar matrices (602 matrices in total). The optimal index table for this set of matrices is composed of four subtables: 1..8, 9..16, 17..23 and 24..30. The number of matrices in each subtable is respectively 602, 502, 100 and 20. After pruning with lemma 1, the corresponding average number of accesses is respectively 602, 216, 18 and 1. The second figure gives the computation time of our algorithm (dotted line) compared to the brute-force algorithm (solid line). The available memory size was 256MB. The second diagram shows that our algorithm is eight times faster than the brute-force one.

Preprocessing of the PWMs

Input: a set $\mathcal{M}$ of PWMs, a natural number e for the available memory space
Output: the optimal index table for $\mathcal{M}$ and e

```
for each $(M, \alpha) \in \mathcal{M}$ do
    for $i = 1$ to length($M$) do
        Compute $\texttt{Pvalue}(M[1..i], \texttt{GLB}(M, i, \alpha))$ with Lemma 1 and Equation 1
$m \leftarrow$max{length($M$); $M \in \mathcal{M}$}
Compute $\phi(m, e)$ with Equation 3 and deduce the optimal cutting $\{i_1, \ldots, i_l\}$
for $k = 1$ to $l$ do
    for each $u \in \Sigma^{i_{k+1} \quad i_k}$ do
        for each $M \in \mathcal{M}$ do
            $T_k(h(u))[M] \leftarrow \texttt{Score}(u, M[i_k..i_{k+1} - 1])$
```

The computation time for the greater lower bound for the set of all matrices $\texttt{Pvalue}(M[1..i], \texttt{GLB}(M, i, \alpha))$ is in $O(\sum_{M \in \mathcal{M}} \text{length}(M))$. At first sight, the time computation for ϕ is in $O(m^2)$. But for bounded values of e, the difference $j-i$ should also be bounded by $\log_\sigma(e)$. So the whole computation can be done within $O(\log_\sigma(e))$. It remains to fill up the tables : this can be done in time $O(l\sigma^{\max\{i_{k+1}-i_k, 1\leq k\leq l\}}|\mathcal{M}|)$. Once the table index is set up, the score of each PWM is computed with Equation 2. In this formula, the summation is stopped as soon as the partial score is smaller than the associated greater lower bound for the score cutoff.

4 The PWM Matching Problem for Similar Matrices

In the previous section, we designed an efficient algorithm for scanning a sequence given a set of arbitrary PWMs. If the PWMs encode for binding sites specific to a family of transcription factors, they may look similar. Figure 5 shows an example of four similar matrices extracted from Transfac. In this case, the occurrences of the PWMs are correlated. Similarity may also come from redundancy between databases. [9, 10, 8] proposed several correlation measures between two matrices. The distance is used to group matrices into clusters. The clusters are then used to analyse transcription factor sites themselves, to compare with a new matrix or to eliminate redundancy between databases. [10] also suggested to use clusters to speed-up the PWM matching problem. Indeed, a representative matrix may be defined for each cluster. The occurrences for all matrices are then deduced from the occurrences of the representative matrix. Nevertheless no method is given and moreover no measure was proposed in order to control prediction errors when a representative matrix is used to predict the occurrences of all the matrices. In this section, we address a new issue of the PWM matching problem when given a set of similar matrices.

4.1 Similar Matrices

The two algorithms are based on the common idea that if PWMs are similar, we may design a *filtering matrix* that helps to decide if a matrix M of the set has an occurence for a given position of the sequence, without computing the score of M systematically. For that task, we need to align all PWMs. Here the process of alignment is to put in correspondence similar columns of the matrices by shifting them, without introducing gaps. Each pairwise alignment induces a frameshift between two PWMs. In full generality, finding the best alignment is NP-hard: it is an alternative formulation of the maximal $k-$clique on a $k-$partite graph problem [1, 4]. Nevertheless, here involved matrices are very similar. We thus assume that the matrices have the following property.

Definition 6 (Consistent PWMs). *Let $\mathcal{M}$ be a set of PWMs. $\mathcal{M}$ is* consistent *if for each triplet of distinct PWMs M, N, O, the best alignment between M and N and the best alignment between M and O allow to deduce the best alignment between N and O.*

In the context of consistent matrices, obtaining the optimal alignment is straightforward: choose a reference matrix and compute the optimal frameshift of each matrix against this reference matrix. From now on, we consider only sets of consistent matrices. Once the matrices are aligned, it is likely to assume that the PWMs all have the same length: it is enough to fill missing columns by vectors of 0. For sake of simplicity, we assume that the matrices are aligned and of the same length in the remaining of this article.

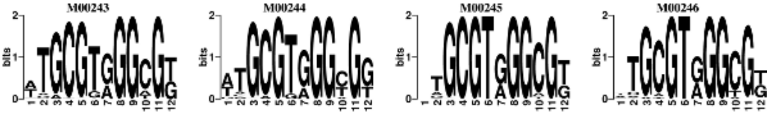


Fig. 5. Four matrices of the EGR-1 family extracted from Transfac. This is a typical case where a filtering matrix can be used. Matrices are very similar and obtaining the alignment is straightforward. Here there is no frameshift.

The choice of the filtering matrix strongly depends on the set of initial PWMs and of the properties we expect for the algorithm. We list here some examples of filtering matrices for a set $\{M_1, \dots, M_k\}$:

- *arithmetical mean* PWM:
$$\forall x \in \Sigma, \forall i \in 1..\mathrm{length}(M),\ M(i,x) = \sum_{j=1}^{k} M_j(i,x)/k$$
- *minimum* PWM:
$$\forall x \in \Sigma, \forall i \in 1..\mathrm{length}(M),\ M(i,x) = \min\{M_1(i,x), \dots, M_k(i,x)\}$$
- *maximum* PWM:
$$\forall x \in \Sigma, \forall i \in 1..\mathrm{length}(M),\ M(i,x) = \max\{M_1(i,x), \dots, M_k(i,x)\}$$

4.2 Lossless Filtering

The first algorithm for similar PWMs is an exact algorithm. Given a filtering matrix and a set of matrices with associated score cutoffs, we define the *uncertainty threshold* that allows us to decide from the filtering matrix score whether or not a PWM of the set has a chance to have a score greater than its score cutoff.

Definition 7 (Uncertainty threshold). *Let M and N be two PWMs. The* uncertainty threshold *between M and N, denoted $U(M,N)$, is defined as the smallest positive natural number x such that*

$$\forall u \in \Sigma^n\ \forall \alpha \in \mathbb{N}\ \ \mathtt{Score}(u,N) \leq \alpha - x \Rightarrow \mathtt{Score}(u,M) \leq \alpha$$

Lemma 2. $U(M,N) = \sum_{i=1}^{n} \max_{x \in \Sigma} (N(i,x) - M(i,x))$.

The first filtering algorithm can now be designed. For each position of the sequence, for each matrix M_i, we use the score of the filtering matrix N and the uncertainty threshold between M_i and N to decide whether or not the score of M_i has to be computed.

Lossless Filtering PWM Matching Algorithm

```
Input: a sequence $S$, a set of PWMs $\{(M_1, \alpha_1), \ldots, (M_k, \alpha_k)\}$
Output: all occurrences of $M_1, \ldots, M_k$ in $S$

Define a filtering matrix $N$ for $\{(M_1, \alpha_1), \ldots, (M_k, \alpha_k)\}$
for $j = 1$ to $p$ compute $U(M_j, N)$
for each position $i$ in the sequence $S$ do
    Compute $s \leftarrow \texttt{Score}(S, i, N)$ with Equation 2
    for $j = 1$ to $p$ do
        if $s > \alpha_j - U(M_j, N)$
            then compute $\texttt{Score}(S, i, M_j)$
```

Equation 1 and Lemma 1 allow us to compute a priori the average proportion of positions in the sequence where it is necessary to compute explicitly the score of the PWM M_j (last line of the algorithm) : this is $\texttt{Pvalue}(N, \alpha_j - U(M_j, N))$. For example, with the four matrices of Figure 5 and a score cutoff corresponding to a P-value of e^{-5}, if we define the filtering matrix as the arithmetical mean matrix, the average proportion of computations for M00243, M00244, M00245, M00246 is 3%, 2.3%, 2.1% and 3.5% respectively. The computation of the filtering matrix on a random sequence of length 10^6 gives rise to 105351 computations of the score of one of the four matrices, that is 2, 7%, compared to the brute-force algorithm.

4.3 Approximate Algorithm

The similarity between matrices may also be exploited to design an approximate algorithm for the PWM matching problem. The algorithm computes only the score of the filtering matrix and if an occurrence exists for the filtering matrix at position i we conclude that each matrix of the set occurs at position i. The algorithm becomes very simple.

Approximate Filtering PWM Matching Algorithm

```
Input: a sequence $S$, a set of PWMs $\{(M_1, \alpha_1), \ldots, (M_k, \alpha_k)\}$
Output: estimation of all occurrences of $M_1, \ldots, M_k$ by all occurences of a filtering
matrix $N$ in $S$

Define a filtering matrix $N$ and a score threshold $\alpha$ for $\{(M_1, \alpha_1), \ldots, (M_k, \alpha_k)\}$
for each position $i$ in the sequence $S$ do
    Compute $\texttt{Score}(S, i, N)$ with Equation 2
```

In this approximate algorithm, the choice of the filtering matrix determines the sensitivity and the selectivity of the estimation. To evaluate the relevance of the filtering matrix, we need to be able to bound the error induced by the approximation in a more accute way than with the uncertainty threshold. For that we define *false positive* and *false negative occurrences.*

Definition 8 (False positives and false negatives). *Let M and N be two PWMs with respective score threshold α_1 and α_2.*

- *u is a* false positive occurrence *for N if* $\texttt{Score}(u, M) \geq \alpha_1$ *and* $\texttt{Score}(u, N) < \alpha_2$*,*
- *u is a* false negative occurrence *for N if* $\texttt{Score}(u, M) < \alpha_1$ *and* $\texttt{Score}(u, N) \geq \alpha_2$*.*

We write $FP(M, N, \alpha_1, \alpha_2)$ for the proportion of false positive occurrences*, and $FN(M, N, \alpha_1, \alpha_2)$ for the proportion of* false negative occurrences *(assuming that the sequences are independently and identically distributed).*

For example, for the three definitions of filtering matrix given in subsection 4.1, we have the following behaviour.

- the minimal matrix with cutoff $\alpha = \max\{\alpha_1, \ldots, \alpha_k\}$ guarantees that there is no false positive occurrence,
- the maximal matrix with cutoff $\alpha = \min\{\alpha_1, \ldots, \alpha_k\}$ guarantees that there is no false negative occurrence,
- the arithmetical mean matrix yields balanced results between false positive and false negative occurrences.

The following lemma shows that it is possible to compute FP and FN exactly, and to control the error induced by the filtering matrix.

Lemma 3. $FP(M, N, \alpha_1, \alpha_2) = l(\text{length}(M), \alpha_1, \alpha_2)$ *where the function l is defined recursively as follows.*

$$l(0, s_1, s_2) = \begin{cases} 1 \textit{ if } s_1 \geq 0 \textit{ and } s_2 < 0 \\ 0 \textit{ otherwise} \end{cases}$$
$$l(i, s_1, s_2) = \textstyle\sum_{x \in \Sigma} l(i-1, s_1 - M(x, i), s_2 - N(x, i))$$

$FN(M, N, \alpha_1, \alpha_2) = l'(\text{length}(M), \alpha_1, \alpha_2)$ *where the function l' is defined recursively as follows.*

$$l'(0, s_1, s_2) = \begin{cases} 1 \textit{ if } s_1 < 0 \textit{ and } s_2 \geq 0 \\ 0 \textit{ otherwise} \end{cases}$$
$$l'(i, s_1, s_2) = \textstyle\sum_{x \in \Sigma} l'(i-1, s_1 - M(x, i), s_2 - N(x, i))$$

This Lemma implies that FP and FN can be computed by dynamic programming by recurrence on i. On the example of Figure 5, the rate of FP and FN are 19% and 21% respectively with the arithmetical mean as the filtering matrix and a P-value of e^{-5} for the score cutoff.

5 Conclusion

We proposed an efficient algorithm for the PWM matching problem in presence of a large set of PWMs, which is eight times faster than the brute-force algorithm. We also investigated the problem of PWM matching for similar matrices. In this perspective, we formulated exact relationships between the set of occurrences of PWMs, that allow to estimate the redundancy of the occurrences. We believe that these results are of a more general interest, and may be used in larger contexts for assessing the significance of multiple occurrences. This question arises frequently when studying regulatory sequences and putative transription factor binding sites. Another vertue of this analysis is that it helps to cope with redundant occurrences in a very efficient way. Redundancy between PWMs is a common problem when one works with public databases [9]. The classical approach is to eliminate overlapping binding sites afterwards using empirical similarity rules. Lemma 3 makes it possible to detect redundant PWMs prior to the searching phase, and to control the overlapping rate.

References

1. Y.S. Chung, S.L. Peng, C.Y. Tang, and J.M. Yang. Finging k-cliques on a k-partite graph. 22th Worshop on Combinatorial Mathematics and Computational Theory, 2005.
2. J.M. Claverie and S. Audic. The statistical significance of nucleotide position-weight matrix matches. *Computer Applications in the Biosciences*, 12(5):431–439, 1996.
3. R. Elkon, C. Linhart, R. Sharan, R. Shamir, and Y. Shiloh. Genome-wide in silico identification of transcriptional regulators controlling the cell cycle in human cells. *Genome Research*, 13:773–780, 2003.
4. T. Grunert, S. Irnich, H-J. Zimmermann, M. Schneider, and B. Wulfhorst. Cliques in k-partite graphs and their application in textile engineering, 2002.
5. Haiyan Huang, Ming-Chih J Kao, Xianghong Zhou, Jun S Liu, and Wing H Wong. Determination of local statistical significance of patterns in markov sequences with application to promoter element identification. *J Comput Biol*, 11(1):1–14, 2004.
6. V.D. Marinescu, I.S. Kohane, and A. Riva. The MAPPER database: a multi-genome catalog of putative transcription factor binding sites. *Nucleic Acids Research*, 33 Database issue:D91–D97, 2005.
7. A. Sandelin, W. Alkema, P. Engstrom, W.W. Wasserman, and B. Lenhard. JASPAR: an open-access database for eukaryotic transcription factor binding profiles. *Nucleic Acids Research*, Database issue:D91-4, 2004.
8. Albin Sandelin and Wyeth W. Wasserman. Constrained binding site diversity within families of transcription factors enhances pattern discovery bioinformatics. *Journal of Molecular Biology*, 338(2):207–215, 2004.
9. E. D. Schones, P. Sumazin, and M.Q. Zhang. Similarity of position frequency matrices for transcription factor binding sites. *Bioinformatics*, 21(3):307–13, 2005.
10. Kielbasa SM, Gonze D, and Herzel H. Measuring similarities between transcription factor binding sites. *BMC Bioinformatics*, 6(237), 2005.
11. R. Staden. Methods for calculating the probabilities of finding patterns in sequences. *Computer Applications in the Biosciences*, 5:89–96, 1989.
12. G.D. Stormo and D. S. Fields. Specificity, free energy and information content in protein-DNA interactions. *Trends in biochemical sciences*, 23:109–113, 1998.
13. S.J. Ho Sui, J.R. Mortimer, D.J. Arenillas, J. Brumm, C.J. Walsh, B.P. Kennedy, and W.W. Wasserman. oPOSSUM: identification of over-represented transcription factor binding sites in co-expressed genes. *Nucleic Acids Res*, 33(10):3154–64, 2005.
14. E. Wingender, X. Chen, R. Hehl, I. Karas, I. Liebich, V. Matys, T. Meinhardt, M. Pruss, I. Reuter, and F. Schacherer. TRANSFAC: an integrated system for gene expression regulation. *Nucleic Acids Research*, 28(1):316–319, 2000.

Author Index