LNCS 4580

Bin Ma
Kaizhong Zhang (Eds.)

Combinatorial Pattern Matching

18th Annual Symposium, CPM 2007
London, Canada, July 2007
Proceedings

Springer

Bin Ma Kaizhong Zhang (Eds.)

Combinatorial Pattern Matching

18th Annual Symposium, CPM 2007
London, Canada, July 9-11, 2007
Proceedings

Springer

Volume Editors

Bin Ma
Kaizhong Zhang
University of Western Ontario
Department of Computer Science
London, Ontario, N6A 5B7, Canada
E-mail: {bma; kzhang}@csd.uwo.ca

Library of Congress Control Number: 2007929746

CR Subject Classification (1998):
F.2.2, I.5.4, I.5.0, 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	3-540-73436-8 Springer Berlin Heidelberg New York
ISBN-13	978-3-540-73436-9 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12086467 06/3180 5 4 3 2 1 0

Preface

The papers contained in this volume were presented at the 18th Annual Symposium on Combinatorial Pattern Matching (CPM 2007) held at the University of Western Ontario, in London, Ontario, Canada from July 9 to 11, 2007.

All the papers presented at the conference are original research contributions on computational pattern matching and analysis, data compression and compressed text processing, suffix arrays and trees, and computational biology. They were selected from 64 submissions. Each submission was reviewed by at least three reviewers. The committee decided to accept 32 papers. The programme also included three invited talks by Tao Jiang from the University of California, Riverside, USA, S. Muthukrishnan from Rutgers University, USA, and Frances Yao from City University of Hong Kong, Hong Kong.

Combinatorial Pattern Matching addresses issues of searching and matching strings and more complicated patterns such as trees, regular expressions, graphs, point sets, and arrays. The goal is to derive non-trivial combinatorial properties of such structures and to exploit these properties in order to either achieve superior performance for the corresponding computational problems or pinpoint conditions under which searches cannot be performed efficiently.

The Annual Symposium on Combinatorial Pattern Matching started in 1990, and has since taken place every year. The objective of the annual CPM meetings is to provide an international forum for research in combinatorial pattern matching and related applications. Previous CPM meetings were held in Paris, London, Tucson, Padova, Asilomar, Helsinki, Laguna Beach, Aarhus, Piscataway, Warwick, Montreal, Jerusalem, Fukuoka, Morelia, Istanbul, Jeju Island, and Barcelona. Selected papers from the first meeting appeared in volume 92 of *Theoretical Computer Science*, from the 11th meeting in volume 2 of the *Journal of Discrete Algorithms*, from the 12th meeting in volume 146 of *Discrete Applied Mathematics*, and from the 14th meeting in volume 3 of the *Journal of Discrete Algorithms*. Starting from the 3rd meeting, the proceedings of all meetings have been published in the LNCS series, volumes 644, 684, 807, 937, 1075, 1264, 1448, 1645, 1848, 2089, 2373, 2676, 3109, 3537, 4009, and 4580.

The whole submission and review process, as well as the production of this volume, was carried out with the help of the EasyChair system. The conference was sponsored by the University of Western Ontario and by the Fields Institute.

April 2007

Bin Ma
Kaizhong Zhang

Conference Organization

Programme Chairs

Bin Ma
Kaizhong Zhang

Programme Committee

Tatsuya Akutsu
Amihood Amir
Raffaele Giancarlo
Gregory Kucherov
Ming Li
Guohui Lin
Heikki Mannila
Gonzalo Navarro
Ron Pinter
Mathieu Raffinot
Cenk Sahinalp
David Sankoff
Steve Skiena
James Storer
Masayuki Takeda
Gabriel Valiente
Martin Vingron
Lusheng Wang

Local Organization

Meg Borthwick
Jingping Liu
Cheryl McGrath
Roberto Solis-Oba (Co-chair)
Kaizhong Zhang (Co-chair)

External Reviewers

Cagri Aksay	Guillaume Blin
Hideo Bannai	Marie-Pierre Béal
Petra Berenbrink	Cedric Chauve

Shihyen Chen
Phuong Dao
Guillaume Fertin
Tom Friedetzky
Liliana Félix
Aris Gionis
Stefan Haas
Niina Haiminen
Iman Hajiresouliha
Tzvika Hartman
Rodrigo Hausen
Hannes Heikinheimo
Yasuto Higa
Fereydoun Hormozdiari
Lucian Ilie
Shunsuke Inenaga
Hossein Jowhari
Oren Kapah
Emre Karakoc
Orgad Keller
Takuya Kida
Roman Kolpakov
Tsvi Kopelowitz
Dennis Kostka
Juha Kärkkäinen
Gad Landau
Michael Lappe
Thierry Lecroq
Avivit Levy
Weiming Li
Yury Lifshits
Jingping Liu
Mercè Llabrés
Antoni Lozano
Giovanni Manzini
Conrado Martínez
Igor Nor
Pasi Rastas
Tobias Rausch
David Reese
Hugues Richard
Jairo Rocha
Oleg Rokhlenko
Francesc Rosselló
Dominique Rossin
Wojciech Rytter
Kunihiko Sadakane
Hiroshi Sakamoto
Rahaleh Salari
Marcel Schulz
Jouni Seppanen
Dina Sokol
Jens Stoye
Takeyuki Tamura
Eric Tannier
Helene Touzet
Antti Ukkonen
Tomas Vinar
Robert Warren
Lei Xin

Table of Contents

A Combinatorial Approach to Genome-Wide Ortholog Assignment: Beyond Sequence Similarity Search

Tao Jiang

Computer Science Department, University of California - Riverside
jiang@cs.ucr.edu

Abstract. The assignment of orthologous genes between a pair of genomes is a fundamental and challenging problem in comparative genomics. Existing methods that assign orthologs based on the similarity between DNA or protein sequences may make erroneous assignments when sequence similarity does not clearly delineate the evolutionary relationship among genes of the same families. In this paper, we present a new approach to ortholog assignment that takes into account both sequence similarity and evolutionary events at genome level, where orthologous genes are assumed to correspond to each other in the most parsimonious evolving scenario under genome rearrangement and gene duplication. It is then formulated as a problem of computing the signed reversal distance with duplicates between two genomes of interest, for which an efficient heuristic algorithm was constructed based on solutions to two new optimization problems, minimum common partition and maximum cycle decomposition. Following this approach, we have implemented a high-throughput system for assigning orthologs on a genome scale, called MSOAR, and tested it on both simulated data and real genome sequence data. Our predicted orthologs between the human and mouse genomes are strongly supported by ortholog and protein function information in authoritative databases, and predictions made by other key ortholog assignment methods such as Ensembl, Homologene, INPARANOID, and HGNC. The simulation results demonstrate that MSOAR in general performs better than the iterated exemplar algorithm of D. Sankoff's in terms of identifying true exemplar genes.

This is joint work with X. Chen (Nanyang Tech. Univ., Singapore), Z. Fu (UCR), J. Zheng (NCBI), V. Vacic (UCR), P. Nan (SCBIT), Y. Zhong (SCBIT), and S. Lonardi (UCR).

Stringology: Some Classic and Some Modern Problems

S. Muthukrishnan

Department of Computer Science, Rutgers University
and
Google Inc.
muthu@cs.rutgers.edu

Abstract. We examine some of the classic problems related to suffix trees from 70's and show some recent results on sorting suffixes with small space and suffix selection. Further, we introduce modern versions of suffix sorting and their application to XML processing. The study of combinatorial aspects of strings continues to flourish, and we present several open problems with modern applications.

Algorithmic Problems in Scheduling Jobs on Variable-Speed Processors

Frances F. Yao

Department of Computer Science,
City University of Hong Kong
Hong Kong SAR, China
csfyao@cityu.edu.hk

Abstract. Power and heat have become two of the major concerns for the computer industry, which struggles to cope with the energy and cooling costs for servers, as well as the short battery life of portable devices. Dynamic Voltage Scaling (DVS) has emerged as a useful technique: e.g. Intel's newest Foxton technology enables a chip to run at 64 different speed levels. Equipped with DVS technology, the operating system can then save CPU's energy consumption by scheduling tasks wisely. A schedule that finishes the given tasks within their timing constraints while using minimum total energy (among all feasible schedules) is called an optimal DVS schedule. A theoretical model for DVS scheduling was proposed in a paper by Yao, Demers and Shenker in 1995, along with a well-formed characterization of the optimum and an algorithm for computing it. This algorithm has remained as the most efficient known despite many investigations of this model. In this talk, we will first give an overview of the DVS scheduling problem, and then present the latest improved results for computing the optimal schedule in both the finite and the continuous (infinite speed levels) models. Related results on efficient on-line scheduling heuristics will also be discussed.

Speeding Up HMM Decoding and Training by Exploiting Sequence Repetitions

Shay Mozes[1,⋆], Oren Weimann[1], and Michal Ziv-Ukelson[2,⋆⋆]

[1] MIT Computer Science and Artificial Intelligence Laboratory, 32 Vassar Street, Cambridge, MA 02139, USA
shaymozes@gmail.com,oweimann@mit.edu
[2] School of Computer Science, Tel-Aviv University, Tel-Aviv 69978, Israel
michaluz@post.tau.ac.il

Abstract. We present a method to speed up the dynamic program algorithms used for solving the HMM decoding and training problems for discrete time-independent HMMs. We discuss the application of our method to Viterbi's decoding and training algorithms [21], as well as to the forward-backward and Baum-Welch [4] algorithms. Our approach is based on identifying repeated substrings in the observed input sequence. We describe three algorithms based alternatively on byte pair encoding (BPE) [19], run length encoding (RLE) and Lempel-Ziv (LZ78) parsing [22]. Compared to Viterbi's algorithm, we achieve a speedup of $\Omega(r)$ using BPE, a speedup of $\Omega(\frac{r}{\log r})$ using RLE, and a speedup of $\Omega(\frac{\log n}{k})$ using LZ78, where k is the number of hidden states, n is the length of the observed sequence and r is its compression ratio (under each compression scheme). Our experimental results demonstrate that our new algorithms are indeed faster in practice. Furthermore, unlike Viterbi's algorithm, our algorithms are highly parallelizable.

Keywords: HMM, Viterbi, dynamic programming, compression.

1 Introduction

Over the last few decades, Hidden Markov Models (HMMs) proved to be an extremely useful framework for modeling processes in diverse areas such as error-correction in communication links [21], speech recognition [6], optical character recognition [2], computational linguistics [17], and bioinformatics [12].

The core HMM-based applications fall in the domain of classification methods and are technically divided into two stages: a training stage and a decoding stage. During the *training* stage, the emission and transition probabilities of an HMM are estimated, based on an input set of observed sequences. This stage is usually executed once as a preprocessing stage and the generated ("trained") models are stored in a database. Then, a *decoding* stage is run, again and again, in order to

⋆ Work conducted while visiting MIT.

⋆⋆ Work supported by an **Eshkol grant** of the Israeli Ministry of Science and Technology.

classify input sequences. The objective of this stage is to find the most probable sequence of states to have generated each input sequence given each model, as illustrated in Fig. 1.

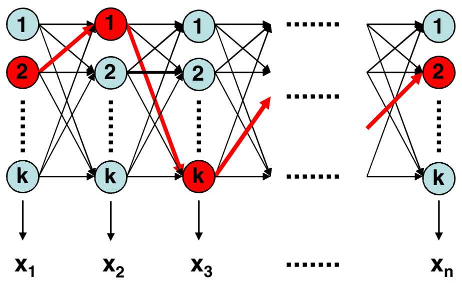


Fig. 1. The HMM on the observed sequence $X = x_1, x_2, \ldots, x_n$ and states $1, 2, \ldots, k$. The highlighted path is a possible path of states that generate the observed sequence. VA finds the path with highest probability.

Obviously, the training problem is more difficult to solve than the decoding problem. However, the techniques used for decoding serve as basic ingredients in solving the training problem. The Viterbi algorithm (VA) [21] is the best known tool for solving the decoding problem. Following its invention in 1967, several other algorithms have been devised for the decoding and training problems, such as the forward-backward and Baum-Welch [4] algorithms. These algorithms are all based on dynamic programs whose running times depend linearly on the length of the observed sequence. The challenge of speeding up VA by utilizing HMM topology was posed in 1997 by Buchsbaum and Giancarlo [6] as a major open problem. In this contribution, we address this open problem by using text compression and present the first provable speedup of these algorithms.

The traditional aim of text compression is the efficient use of resources such as storage and bandwidth. Here, however, we compress the observed sequences in order to speed up HMM algorithms. This approach, denoted "acceleration by text-compression", was previously applied to some classical problems on strings. Various compression schemes, such as LZ77, LZW-LZ78, Huffman coding, Byte Pair Encoding (BPE) and Run Length Encoding (RLE), were employed to accelerate exact and approximate pattern matching [14,16,19,1,13,18] and sequence alignment [3,7,11,15]. In light of the practical importance of HMM-based classification methods in state-of-the-art research, and in view of the fact that such techniques are also based on dynamic programming, we set out to answer the following question: can "acceleration by text compression" be applied to HMM decoding and training algorithms?

Our results. Let X denote the input sequence and let n denote its length. Let k denote the number of states in the HMM. For any given compression scheme, let n' denote the number of parsed blocks in X and let $r = n/n'$ denote the compression ratio. Our results are as follows.

1. BPE is used to accelerate decoding by a factor of $\Omega(r)$.
2. RLE is used to accelerate decoding by a factor of $\Omega(\frac{r}{logr})$.
3. Using LZ78, we accelerate decoding by a factor of $\Omega(\frac{\log n}{k})$. Our algorithm guarantees no degradation in efficiency even when $k > \log n$ and is experimentally more than five times faster than VA.
4. The same speedup factors apply to the Viterbi training algorithm.
5. For the Baum-Welch training algorithm, we show how to preprocess a repeated substring of size ℓ once in $O(\ell k^4)$ time so that we may replace the usual $O(\ell k^2)$ processing work for each occurrence of this substring with an alternative $O(k^4)$ computation. This is beneficial for any repeat with λ non-overlapping occurrences, such that $\lambda > \frac{\ell k^2}{\ell - k^2}$.
6. As opposed to VA, our algorithms are highly parallelizable. This is discussed in the full version of this paper.

Roadmap. The rest of the paper is organized as follows. In section 2 we give a unified presentation of the HMM dynamic programs. We then show in section 3 how these algorithms can be improved by identifying repeated substrings. Two compressed decoding algorithms are given in sections 4 and 5. In section 6 we show how to adapt the algorithms to the training problem. Finally, experimental results are presented in Section 7.

2 Preliminaries

Let Σ denote a finite alphabet and let $X \in \Sigma^n$, $X = x_1, x_2, \ldots, x_n$ be a sequence of observed letters. A Markov *model* is a set of k states, along with emission probabilities $e_k(\sigma)$ - the probability to observe $\sigma \in \Sigma$ given that the state is k, and transition probabilities $P_{i,j}$ - the probability to make a transition to state i from state j.

The Viterbi Algorithm. The Viterbi algorithm (VA) finds the most probable sequence of hidden states given the model and the observed sequence. i.e., the sequence of states $s_1, s_2, \ldots, s_n$ which maximize

$$\prod_{i=1}^{n} e_{s_i}(x_i) P_{s_i, s_{i-1}} \tag{1}$$

The dynamic program of VA calculates a vector $v_t[i]$ which is the probability of the most probable sequence of states emitting $x_1, \ldots, x_t$ and ending with the state i at time t. v_0 is usually taken to be the vector of uniform probabilities (i.e., $v_0[i] = \frac{1}{k}$). v_{t+1} is calculated from v_t according to

$$v_{t+1}[i] = e_i(x_{t+1}) \cdot \max_j \{P_{i,j} \cdot v_t[j]\} \tag{2}$$

Definition 1 (Viterbi Step). *We call the computation of v_{t+1} from v_t a Viterbi step.*

Clearly, each Viterbi step requires $O(k^2)$ time. Therefore, the total runtime required to compute the vector v_n is $O(nk^2)$. The probability of the most likely sequence of states is the maximal element in v_n. The actual sequence of states can be then reconstructed in linear time.

It is useful for our purposes to rewrite VA in a slightly different way. Let M^σ be a $k \times k$ matrix with elements $M^\sigma_{i,j} = e_i(\sigma) \cdot P_{i,j}$. We can now express v_n as:

$$v_n = M^{x_n} \odot M^{x_{n-1}} \odot \cdots \odot M^{x_2} \odot M^{x_1} \odot v_0 \tag{3}$$

where $(A \odot B)_{i,j} = \max_k\{A_{i,k} \cdot B_{k,j}\}$ is the so called max-times matrix multiplication. VA computes v_n using (3) from right to left in $O(nk^2)$ time. Notice that if (3) is evaluated from left to right the computation would take $O(nk^3)$ time (matrix-vector multiplication vs. matrix-matrix multiplication). Throughout, we assume that the max-times matrix-matrix multiplications are done naïvely in $O(k^3)$. Faster methods for max-times matrix multiplication [8] and standard matrix multiplication [20,10] can be used to reduce the k^3 term. However, for small values of k this is not profitable.

The Forward-Backward Algorithms. The *forward-backward* algorithms are closely related to VA and are based on very similar dynamic programs. In contrast to VA, these algorithms apply standard matrix multiplication instead of max-times multiplication. The forward algorithm calculates $f_t[i]$, the probability to observe the sequence $x_1, x_2, \ldots, x_t$ requiring that $s_t = i$ as follows:

$$f_t = M^{x_t} \cdot M^{x_{t-1}} \cdot \cdots \cdot M^{x_2} \cdot M^{x_1} \cdot f_0 \tag{4}$$

The backward algorithm calculates $b_t[i]$, the probability to observe the sequence $x_{t+1}, x_{t+2}, \ldots, x_n$ given that $s_t = i$ as follows:

$$b_t = b_n \cdot M^{x_n} \cdot M^{x_{n-1}} \cdot \cdots \cdot M^{x_{t+2}} \cdot M^{x_{t+1}} \tag{5}$$

Another algorithm which is used in the training stage and employs the forward-backward algorithm as a subroutine, is the Baum-Welch algorithm, to be further discussed in Section 6.

A motivating example. We briefly describe one concrete example from computational biology to which our algorithms naturally apply. CpG islands [5] are regions of DNA with a large concentration of the nucleotide pair CG. These regions are typically a few hundred to a few thousand nucleotides long, located around the promoters of many genes. As such, they are useful landmarks for the identification of genes. The observed sequence (X) is a long DNA sequence composed of four possible nucleotides ($\Sigma = \{A, C, G, T\}$). The length of this sequence is typically a few millions nucleotides ($n \simeq 2^{25}$). A well-studied classification problem is that of parsing a given DNA sequence into CpG islands and non CpG regions. Previous work on CpG island classification used Markov models with either 8 or 2 states ($k = 8$ or $k = 2$) [9,12].

3 Exploiting Repeated Substrings in the Decoding Stage

Consider a substring $W = w_1, w_2, \ldots, w_\ell$ of X, and define

$$M(W) = M^{w_\ell} \odot M^{w_{\ell-1}} \odot \cdots \odot M^{w_2} \odot M^{w_1} \tag{6}$$

Intuitively, $M_{i,j}(W)$ is the probability of the most likely path starting with state j, making a transition into some other state, emitting w_1, then making a transition into yet another state and emitting w_2 and so on until making a final transition into state i and emitting w_ℓ.

In the core of our method stands the following observation, which is immediate from the associative nature of matrix multiplication.

Observation 1. *We may replace any occurrence of* $M^{w_\ell} \odot M^{w_{\ell-1}} \odot \cdots \odot M^{w_1}$ *in eq.* (3) *with* $M(W)$*.*

The application of observation 1 to the computation of equation (3) saves $\ell - 1$ Viterbi steps *each* time W appears in X, but incurs the additional cost of computing $M(W)$ once.

An intuitive exercise. Let λ denote the number of times a given word W appears, in non-overlapping occurrences, in the input string X. Suppose we naïvely compute $M(W)$ using $(|W| - 1)$ max-times matrix multiplications, and then apply observation 1 to all occurrences of W before running VA. We gain some speedup in doing so if

$$\begin{aligned} (|W| - 1)k^3 + \lambda k^2 &< \lambda |W| k^2 \\ \lambda &> k \end{aligned} \tag{7}$$

Hence, if there are at least k non-overlapping occurrences of W in the input sequence, then it is worthwhile to naïvely precompute $M(W)$, regardless of it's size $|W|$.

Definition 2 (Good Substring). *We call a substring W good if we decide to compute $M(W)$.*

We can now give a general four-step framework of our method:

(I) *Dictionary Selection:* choose the set $D = \{W_i\}$ of good substrings.
(II) *Encoding:* precompute the matrices $M(W_i)$ for every $W_i \in D$.
(III) *Parsing:* partition the input sequence X into consecutive good substrings $X = W_{i_1} W_{i_2} \cdots W_{i_{n''}}$ and let X' denote the compressed representation of this parsing of X, such that $X' = i_1 i_2 \cdots i_{n''}$.
(IV) *Propagation:* run VA on X', using the matrices $M(W_i)$.

The above framework introduces the challenge of how to select the set of good substrings (step I) and how to efficiently compute their matrices (step II). In the next two sections we show how the RLE and LZ78 compression schemes can be applied to address this challenge. The utilization of the BPE compression scheme is discussed in the full version of this paper. Another challenge is how to parse the sequence X (step III) in order to maximize acceleration. We show that, surprisingly, this optimal parsing may differ from the initial parsing induced by the selected compression scheme. To our knowledge, this feature was not applied by previous "acceleration by compression" algorithms.

Throughout this paper we focus on computing path probabilities rather than the paths themselves. The actual paths can be reconstructed in linear time as described in the full version of this paper.

4 Acceleration Via Run-Length Encoding

In this section we obtain an $\Omega(\frac{r}{logr})$ speedup for decoding an observed sequence with run-length compression ratio r. A string S is *run-length encoded* if it is described as an ordered sequence of pairs (σ, i), often denoted "σ^i". Each pair corresponds to a *run* in S, consisting of i consecutive occurrences of the character σ. For example, the string *aaabbcccccc* is encoded as $a^3b^2c^6$. Run-length encoding serves as a popular image compression technique, since many classes of images (e.g., binary images in facsimile transmission or for use in optical character recognition) typically contain large patches of identically-valued pixels. The four-step framework described in section 3 is applied as follows.

(I) *Dictionary Selection:* for every $\sigma \in \Sigma$ and every $i = 1, 2, \ldots, \log n$ we choose σ^{2^i} as a *good substring.*

(II) *Encoding:* since $M(\sigma^{2^i}) = M(\sigma^{2^{i-1}}) \odot M(\sigma^{2^{i-1}})$, we can compute the matrices using repeated squaring.

(III) *Parsing:* Let $W_1 W_2 \cdots W_{n'}$ be the RLE of X, where each W_i is a run of some $\sigma \in \Sigma$. X' is obtained by further parsing each W_i into at most $\log |W_i|$ good substrings of the form σ^{2^j}.

(IV) *Propagation:* run VA on X', as described in Section 3.

Time and Space Complexity Analysis. The offline preprocessing stage consists of steps I and II. The time complexity of step II is $O(|\Sigma|k^3 \log n)$ by applying max-times repeated squaring in $O(k^3)$ time per multiplication. The space complexity is $O(|\Sigma|k^2 \log n)$. This work is done offline once, during the training stage, in advance for all sequences to come. Furthermore, for typical applications, the $O(|\Sigma|k^3 \log n)$ term is much smaller than the $O(nk^2)$ term of VA.

Steps III and IV both apply one operation per occurrence of a good substring in X': step III computes, in constant time, the index of the next parsing-comma, and step IV applies a single Viterbi step in k^2 time. Since $|X'| = \sum_{i=1}^{n'} log|W_i|$, the complexity is

$$\sum_{i=1}^{n'} k^2 log|W_i| = k^2 log(|W_1| \cdot |W_2| \cdots |W_{n'}|) \leq k^2 log((n/n')^{n'}) = O(n'k^2 log\frac{n}{n'}).$$

Thus, the speedup compared to the $O(nk^2)$ time of VA is $\Omega(\frac{\frac{n}{n'}}{log\frac{n}{n'}}) = \Omega(\frac{r}{logr})$.

5 Acceleration Via LZ78 Parsing

In this section we obtain an $\Omega(\frac{\log n}{k})$ speedup for decoding, and a constant speedup in the case where $k > \log n$. We show how to use the LZ78 [22] (henceforth LZ) parsing to find good substrings and how to use the incremental nature of the LZ parse to compute $M(W)$ for a good substring W in $O(k^3)$ time.

LZ parses the string X into substrings (LZ-words) in a single pass over X. Each LZ-word is composed of the longest LZ-word previously seen plus a single letter. More formally, LZ begins with an empty dictionary and parses according to the following rule: when parsing location i, look for the longest LZ-word W starting at position i which already appears in the dictionary. Read one more letter σ and insert $W\sigma$ into the dictionary. Continue parsing from position $i + |W| + 1$. For example, the string "AACGACG" is parsed into four words: A, AC, G, ACG. Asymptotically, LZ parses a string of length n into $O(hn/\log n)$ words [22], where $0 \leq h \leq 1$ is the entropy of the string. The LZ parse is performed in linear time by maintaining the dictionary in a trie. Each node in the trie corresponds to an LZ-word. The four-step framework described in section 3 is applied as follows.

(I) *Dictionary Selection:* the good substrings are all the LZ-words in the LZ-parse of X.
(II) *Encoding:* construct the matrices incrementally, according to their order in the LZ-trie, $M(W\sigma) = M(W) \odot M^{\sigma}$.
(III) *Parsing:* X' is the LZ-parsing of X.
(IV) *Propagation:* run VA on X', as described in section 3.

Time and Space Complexity Analysis. Steps I and III were already conducted offline during the pre-processing compression of the input sequences (in any case LZ parsing is linear). In step II, computing $M(W\sigma) = M(W) \odot M^{\sigma}$, takes $O(k^3)$ time since $M(W)$ was already computed for the good substring W. Since there are $O(n/\log n)$ LZ-words, calculating the matrices $M(W)$ for all Ws takes $O(k^3 n/\log n)$. Running VA on X' (step IV) takes just $O(k^2 n/\log n)$ time. Therefore, the overall runtime is dominated by $O(k^3 n/\log n)$. The space complexity is $O(k^2 n/\log n)$.

The above algorithm is useful in many applications, such as CpG island classification, where $k < \log n$. However, in those applications where $k > \log n$ such an algorithm may actually slow down VA.

We next show an adaptive variant that is guaranteed to speed up VA, regardless of the values of n and k. This graceful degradation retains the asymptotic $\Omega(\frac{\log n}{k})$ acceleration when $k < \log n$.

5.1 An Improved Algorithm

Recall that given $M(W)$ for a good substring W, it takes k^3 time to calculate $M(W\sigma)$. This calculation saves k^2 operations each time $W\sigma$ occurs in X in comparison to the situation where only $M(W)$ is computed. Therefore, in step

I we should include in D, as good substrings, only words that appear as a prefix of at least k LZ-words. Finding these words can be done in a single traversal of the trie. The following observation is immediate from the prefix monotonicity of occurrence tries.

Observation 2. *Words that appear as a prefix of at least k LZ-words are represented by trie nodes whose subtrees contain at least k nodes.*

In the previous case it was straightforward to transform X into X', since each phrase p in the parsed sequence corresponded to a good substring. Now, however, X does not divide into just good substrings and it is unclear what is the optimal way to construct X' (in step III). Our approach for constructing X' is to first parse X into all LZ-words and then apply the following greedy parsing to each LZ-word W: using the trie, find the longest good substring $w' \in D$ that is a prefix of W, place a parsing comma immediately after w' and repeat the process for the remainder of W.

Time and Space Complexity Analysis. The improved algorithm utilizes substrings that guarantee acceleration (with respect to VA) so it is therefore faster than VA even when $k = \Omega(\log n)$. In addition, in spite of the fact that this algorithm re-parses the original LZ partition, the algorithm still guarantees an $\Omega(\frac{\log n}{k})$ speedup over VA as shown by the following lemma.

Lemma 1. *The running time of the above algorithm is bounded by $O(k^3 n/\log n)$.*

Proof. The running time of step II is at most $O(k^3 n/\log n)$. This is because the size of the entire LZ-trie is $O(n/\log n)$ and we construct the matrices, in $O(k^3)$ time each, for just a subset of the trie nodes. The running time of step IV depends on the number of new phrases (commas) that result from the re-parsing of each LZ-word W. We next prove that this number is at most k for each word.

Consider the first iteration of the greedy procedure on some LZ-word W. Let w' be the longest prefix of W that is represented by a trie node with at least k descendants. Assume, contrary to fact, that $|W| - |w'| > k$. This means that w'', the child of w', satisfies $|W| - |w''| \geq k$, in contradiction to the definition of w'. We have established that $|W| - |w'| \leq k$ and therefore the number of re-parsed words is bounded by $k + 1$. The propagation step IV thus takes $O(k^3)$ time for each one of the $O(n/\log n)$ LZ-words. So the total time complexity remains $O(k^3 n/\log n)$. □

Based on Lemma 1, and assuming that steps I and III are pre-computed offline, the running time of the above algorithm is $O(nk^2/e)$ where $e = \Omega(\max(1, \frac{\log n}{k}))$. The space complexity is $O(k^2 n/logn)$.

6 The Training Problem

In the training problem we are given as input the number of states in the HMM and an observed training sequence X. The aim is to find a set of model parameters θ (i.e., the emission and transition probabilities) that maximize the likelihood to observe the given sequence $P(X|\theta)$. The most commonly used training algorithms for HMMs are based on the concept of Expectation Maximization. This is an iterative process in which each iteration is composed of two steps. The first step solves the decoding problem given the current model parameters. The second step uses the results of the decoding process to update the model parameters. These iterative processes are guaranteed to converge to a local maximum. It is important to note that since the dictionary selection step (I) and the parsing step (III) of our algorithm are independent of the model parameters, we only need run them once, and repeat just the encoding step (II) and the propagation step (IV) when the decoding process is performed in each iteration.

6.1 Viterbi Training

The first step of Viterbi training [12] uses VA to find the most likely sequence of states given the current set of parameters (i.e., decoding). Let A_{ij} denote the number of times the state i follows the state j in the most likely sequence of states. Similarly, let $E_i(\sigma)$ denote the number of times the letter σ is emitted by the state i in the most likely sequence. The updated parameters are given by:

$$P_{ij} = \frac{A_{ij}}{\sum_{i'} A_{i'j}} \text{ and } e_i(\sigma) = \frac{E_i(\sigma)}{\sum_{\sigma'} E_i(\sigma')} \tag{8}$$

Note that the Viterbi training algorithm does not converge to the set of parameters that maximizes the likelihood to observe the given sequence $P(X|\theta)$, but rather the set of parameters that locally maximizes the contribution to the likelihood from the most probable sequence of states [12]. It is easy to see that the time complexity of each Viterbi training iteration is $O(k^2n+n) = O(k^2n)$ so it is dominated by the running time of VA. Therefore, we can immediately apply our compressed decoding algorithms from sections 4 and 5 to obtain a better running time per iteration.

6.2 Baum-Welch Training

The Baum-Welch training algorithm [4,12] converges to a set of parameters that locally maximize the likelihood to observe the given sequence $P(X|\theta)$, and is the most commonly used method for model training. We give here a brief explanation of the algorithm and of our acceleration approach. The complete details appear in the full version of this paper.

Recall the forward-backward matrices: $f_t[i]$ is the probability to observe the sequence $x_1, x_2, \ldots, x_t$ requiring that the t'th state is i and that $b_t[i]$ is the probability to observe the sequence $x_{t+1}, x_{t+2}, \ldots, x_n$ given that the t'th state is i. The first step of Baum-Welch calculates $f_t[i]$ and $b_t[i]$ for every $1 \leq t \leq n$ and every $1 \leq i \leq k$. This is achieved by applying the forward and backward algorithms to the input data in $O(nk^2)$ time (see eqs. (4) and (5)). The second step recalculates A and E according to

$$A_{i,j} = \sum_t P(s_t = j, s_{t+1} = i | X, \theta)$$
$$E_i(\sigma) = \sum_{t|x_t=\sigma} P(s_t = i | X, \theta) \tag{9}$$

where $P(s_t = j, s_{t+1} = i|X, \theta)$ is the probability that a transition from state j to state i occurred in position t in the sequence X, and $P(s_t = i|X, \theta)$ is the probability for the t'th state to be i in the sequence X. These quantities are given by:

$$P(s_t = j, s_{t+1} = i|X, \theta) = \frac{f_t[j] \cdot P_{i,j} \cdot e_i(x_{t+1}) \cdot b_{t+1}[i]}{\sum_i f_n[i]} \tag{10}$$

and

$$P(s_t = i|X, \theta) = \frac{f_t[i] \cdot b_t[i]}{\sum_i f_n[i]}. \tag{11}$$

Finally, after the matrices A and E are recalculated, Baum-Welch updates the model parameters according to equation (8).

We next describe how to accelerate the Baum-Welch algorithm. Note that in the first step of Baum-Welch, our algorithms to accelerate VA (Sections 4 and 5) can be used to accelerate the forward-backward algorithms by replacing the max-times matrix multiplication with regular matrix multiplication. However, the accelerated algorithms only compute f_t and b_t on the boundaries of good substrings. In order to solve this problem and speed up the second step of Baum-Welch as well, we observe that when accumulating the contribution of some appearance of a good substring W of length $|W| = \ell$ to A, Baum-Welch performs $O(\ell k^2)$ operations, but updates at most k^2 entries (the size of A). Hence, it is possible to obtain a speedup by precalculating the contribution of each good substring to A and E. For brevity the details are omitted here and will appear in the full version of this paper. To summarize the results, preprocessing a good substring W requires $O(\ell k^4)$ time and $O(k^4)$ space. Using the preprocessed information and the values of f_t and b_t on the boundaries of good substrings, we can update A and E in $O(k^4)$ time per good substring (instead of ℓk^2). To get a speedup we need λ, the number of times the good substring W appears in X to satisfy:

$$\ell k^4 + \lambda k^4 < \lambda \ell k^2$$
$$\lambda > \frac{\ell k^2}{\ell - k^2} \tag{12}$$

This is reasonable if k is small. If $\ell = 2k^2$, for example, then we need λ to be greater than $2k^2$. In the CpG islands problem, if $k = 2$ then any substrings of length eight is good if it appears more than eight times in the text.

7 Experimental Results

We implemented both a variant of our improved LZ-compressed algorithm from subsection 5.1 and classical VA in C++ and compared their execution times on a

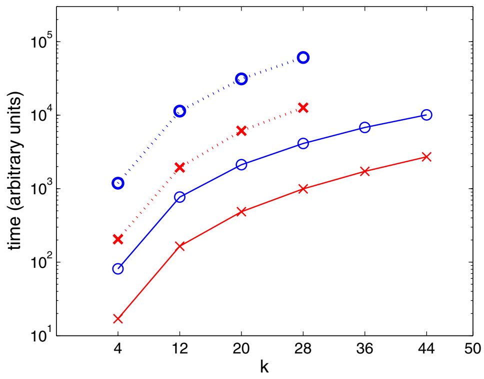


Fig. 2. Comparison of the cumulative running time of steps II and IV of our algorithm (marked x) with the running time of VA (marked o), for different values of k. Time is shown in arbitrary units on a logarithmic scale. Runs on the 1.5Mbp chromosome 4 of S. cerevisiae are in solid lines. Runs on the 22Mbp human Y-chromosome are in dotted lines. The roughly uniform difference between corresponding pairs of curves reflects a speedup factor of more than five.

sequence of approximately 22,000,000 nucleotides from the human Y chromosome and on a sequence of approximately 1,500,000 nucleotides from chromosome 4 of S. Cerevisiae obtained from the UCSC genome database. The benchmarks were performed on a single processor of a SunFire V880 server with 8 UltraSPARC-IV processors and 16GB main memory. The implementation is just for calculating the probability of the most likely sequence of states, and does not traceback the optimal sequence itself. As we have seen, this is the time consuming part of the algorithm. We measured the running times for different values of k. As we explained in the previous sections we are only interested in the running time of the encoding and the propagation steps (II and IV) since the combined parsing/dictionary-selections steps (I and III) may be performed in advance and are not repeated by the training and decoding algorithms. The results are shown in Fig. 2. Our algorithm performs faster than VA even for surprisingly large values of k. For example, for $k = 60$ our algorithm is roughly three times faster than VA.

References

1. Benson, G., Amir, A., Farach, M.: Let sleeping files lie: Pattern matching in Z-compressed files. Journal of Comp. and Sys. Sciences 52(2), 299–307 (1996)
2. Agazzi, O., Kuo, S.: HMM based optical character recognition in the presence of deterministic transformations. Pattern recognition 26, 1813–1826 (1993)
3. Apostolico, A., Landau, G.M., Skiena, S.: Matching for run length encoded strings. Journal of Complexity 15(1), 4–16 (1999)
4. Baum, L.E.: An inequality and associated maximization technique in statistical estimation for probabilistic functions of a Markov process. Inequalities 3, 1–8 (1972)

5. Bird, A.P.: Cpg-rich islands as gene markers in the vertebrate nucleus. Trends in Genetics 3, 342–347 (1987)
6. Buchsbaum, A.L., Giancarlo, R.: Algorithmic aspects in speech recognition: An introduction. ACM Journal of Experimental Algorithms, 2(1) (1997)
7. Bunke, H., Csirik, J.: An improved algorithm for computing the edit distance of run length coded strings. Information Processing Letters 54, 93–96 (1995)
8. Chan, T.M.: All-pairs shortest paths with real weights in $O(n^3/logn)$ time. In: Proc. 9th Workshop on Algorithms and Data Structures, pp. 318–324 (2005)
9. Churchill, G.A.: Hidden Markov chains and the analysis of genome structure. Computers Chem. 16, 107–115 (1992)
10. Coppersmith, D., Winograd, S.: Matrix multiplication via arithmetical progressions. Journal of Symbolic Computation 9, 251–280 (1990)
11. Crochemore, M., Landau, G., Ziv-Ukelson, M.: A sub-quadratic sequence alignment algorithm for unrestricted cost matrices. In: Proc. 13th Annual ACMSIAM Symposium on Discrete Algorithms, pp. 679–688 (2002)
12. Durbin, R., Eddy, S., Krigh, A., Mitcheson, G.: Biological Sequence Analysis. Cambridge University Press, Cambridge (1998)
13. Karkkainen, J., Navarro, G., Ukkonen, E.: Approximate string matching over Ziv-Lempel compressed text. In: Giancarlo, R., Sankoff, D. (eds.) CPM 2000. LNCS, vol. 1848, pp. 195–209. Springer, Heidelberg (2000)
14. Karkkainen, J., Ukkonen, E.: Lempel-Ziv parsing and sublinear-size index structures for string matching. In: Proc. Third South American Workshop on String Processing (WSP), pp. 141–155 (1996)
15. Makinen, V., Navarro, G., Ukkonen, E.: Approximate matching of run-length compressed strings. In: Proc. 12th Annual Symposium On Combinatorial Pattern Matching (CPM). LNCS, vol. 1645, pp. 1–13. Springer, Heidelberg (1999)
16. Manber, U.: A text compression scheme that allows fast searching directly in the compressed file. In: CPM 2001. LNCS, vol. 2089, pp. 31–49. Springer, Heidelberg (2001)
17. Manning, C., Schutze, H.: Statistical Natural Language Processing. MIT Press, Cambridge (1999)
18. Navarro, G., Kida, T., Takeda, M., Shinohara, A., Arikawa, S.: Faster approximate string matching over compressed text. In: Proc. Data Compression Conference (DCC), pp. 459–468 (2001)
19. Shibata, Y., Kida, T., Fukamachi, S., Takeda, M., Shinohara, A., Shinohara, T., Arikawa, S.: Speeding up pattern matching by text compression. In: Bongiovanni, G., Petreschi, R., Gambosi, G. (eds.) CIAC 2000. LNCS, vol. 1767, pp. 306–315. Springer, Heidelberg (2000)
20. Strassen, V.: Gaussian elimination is not optimal. Numerische Mathematik 13, 354–356 (1969)
21. Viterbi, A.: Error bounds for convolutional codes and an asymptotically optimal decoding algorithm. IEEE Transactions on Information Theory IT-13, 260–269 (1967)
22. Ziv, J., Lempel, A.: On the complexity of finite sequences. IEEE Transactions on Information Theory 22(1), 75–81 (1976)

On Demand String Sorting over Unbounded Alphabets

Carmel Kent[1], Moshe Lewenstein[2], and Dafna Sheinwald[1]

[1] IBM Research Lab, Haifa 31905, Israel
[2] Department of Computer Science, Bar-Ilan University, Ramat-Gan 52900, Israel

Abstract. On-demand string sorting is the problem of preprocessing a set of n strings to allow subsequent queries of finding the $k < n$ lexicographically smallest strings (and afterwards the next k etc.) This on-demand variant strongly resembles the search engine queries which give you the best k-ranked pages recurringly.

We present a data structure that supports this in $O(n)$ preprocessing time, and answers queries in $O(\log n)$ time. There is also a cost of $O(N)$ time amortized over all operations, where N is the total length of the strings.

Our data structure is a heap of strings, which supports heapify and delete-mins. As it turns out, implementing a full heap with all operations is not that simple. For the sake of completeness we propose a heap with full operations based on balanced indexing trees that supports the heap operations in optimal times.

1 Introduction

Sorting strings is a fundamental algorithmic task that has been intensively researched in various flavors. The classical problem appears in textbooks [1,16], and variants of the problem have attracted much interest over the years, e.g. multikey sorting [5,21,7], parallel string sorting [12,11,13] and cache-aware string sorting [4,22]. Even integer sorting can be considered a special case of this problem.

Over the last few years there has also been much interest in indexing structures, such as suffix trees [8,19,23,25] and suffix arrays [18]. The strong connection between suffix tree construction and suffix sorting was stressed in [8] and in the extended journal version [9]. In fact, a suffix array is an array containing a lexicographic ordering of the suffixes. One of the exciting results of the last several years is a linear time algorithm to construct suffix arrays, see [14,15,17]. These results followed along the line of the suffix tree construction of Farach [8] (note the leaves of the suffix tree also represent a lexicographic ordering of the suffixes of the string). Nevertheless, the linear time results hold for alphabets which can be sorted in linear time. For unbounded alphabets the time to sort the strings is still $O(n \log n)$, where n is the string length.

While all suffixes of an n length string (even over an unbounded alphabet) can be sorted in $O(n \log n)$ time, when it comes to sorting strings one needs to take into consideration also the overall length of the strings, which we denote N. Nevertheless, it is known that one can sort strings in time $O(n \log n + N)$ which matches the lower bound in the comparison model. To achieve this time bound one may use a weight balanced ternary search trie [20] or adapted mergesort [3] (where the time bounds are implicit) or use balanced indexing structures [2], among others. The adapted mergesort technique [3] has the advantage of using very little extra memory and works well in cache. The weight balanced ternary search tries [20] and balanced indexing structures [2] have the advantage of being dynamic. There have also been studies of more practically efficient methods [7,6] who adapt quicksort.

In the on demand setting it is required to preprocess a collection of items for output where the user controls the number of elements it desires to see in a well-defined order. For example, a search engine query may have a long list of hits which gets ranked in some ordering of importance. However, the search engine will return a limited (digestible) number, say k, of the best hits to the user. If the user desires another k these will be fetched and returned. (See, e.g., [26])

In *on demand string sorting* one is given a collection of strings and desires to preprocess the strings so that subsequent queries of “return the next k smallest lexicographically strings” will execute fast, consuming time proportional to k.

One way to solve this is to first sort the strings in one of the $O(n \log n + N)$ methods and then simply return the next k in $O(k)$ time. It is also possible to concatenate all the strings (separated by external symbols) to one large string S and then to create a suffix array for S. This will work well if the alphabet size is $O(|S|) = O(N)$ since the suffix array can then be created in $O(N)$ time and the desired ordering can be extracted from the suffix array ordering. However, if the alphabet is unbounded the suffix array construction will take $O(N \log N)$ time, which is worse than the previous solutions.

In on demand sorting of numbers a heap is a natural data structure to use. We propose to do the same for strings. We propose to use a heap where the elements of the heap are strings. However, a simple implementation will lead us to running times which can be even worse than we have just mentioned. We propose a construction of the heap in a careful manner utilizing the *longest common prefixes* (lcp's) among pairs of strings. In fact, lcp's have been used in suffix arrays, quicksort of strings [7,6], mergesort of strings [3] and in balanced indexing structures [2] (although they were not used in the ternary digital search tries [20]). However, handling the lcp's require special care with regards to heaps and we elaborate on this later.

Note that it is sufficient to support heapify and delete-mins to capture the application of the on demand string sorting, which we show how to support. Nevertheless, allowing insertions together with delete-mins does not work well with the lcp solution. To solve this we use a balanced indexing structure [2].

The roadmap to our paper is as follows: in section 2 we give preliminaries and definitions. In section 3 we recall the balanced indexing data structure [2] and show what needs to be adapted for it to sort strings (instead of suffixes mentioned there). In section 4 we introduce the heap (of strings) data structure and show how to implement heapify, delete-mins, insertions, and delete-min and insertion together. In section 5 we show how to support a heap fully using a balanced indexing structure.

2 Definitions and Preliminaries

Alphabet Σ is a fully ordered set of letters. Σ is unbounded, yet, we assume that any two letters thereof can be compared in a single computational operation, consuming a constant amount of time. A string S of *length* m over Σ is a sequence of m characters $S[1], S[2], \ldots, S[m]$, each being a member of Σ. The length, m, of S is denoted $|S|$. For $1 \leq i \leq j \leq m$, we denote by $S[i,j]$ the substring $S[i], S[i+1], \ldots, S[j]$ of length $j-i+1$. When $i > j$, substring $S[i,j]$ is defined to be the empty string of length 0. We say that $S_1 = S_2$, if $|S_1| = |S_2|$, and $S_1[i] = S_2[i]$ for all $1 \leq i \leq |S_1|$. We say that $S_1 < S_2$, if S_1 precedes S_2 in the lexicographic order. Formally, $S_1 < S_2$, if there exists an index $1 \leq j < \min\{|S_1|, |S_2|\}$ such that $S_1[1,j] = S_2[1,j]$ and $S_1[j+1] < S_2[j+1]$, or if $|S_1| < |S_2|$ and $S_1 = S_2[1, |S_1|]$.

For ease of description, we assume that the strings in the underlying dataset are distinct. Modifications needed to also allow equal strings are rather obvious.

2.1 Longest Common Prefix

Definition 1. *Given strings S_1 and S_2, the largest $i \leq \min\{|S_1|, |S_2|\}$, such that $S_1[1,i] = S_2[1,i]$ is denoted by $\boldsymbol{lcp}(S_1, S_2)$, and called the length of the* longest common prefix *of S_1 and S_2.*

The following well-known folklore lemma has been widely used for string algorithms. For brevity, we omit its proof.

Lemma 1. *Given strings $S_1 \leq S_2 \leq \cdots \leq S_m$, then*
$\boldsymbol{lcp}(S_1, S_m) = \min_{1 \leq i < m} \boldsymbol{lcp}(S_i, S_{i+1})$.

Corollary 1. *Given strings S_1, S_2, S_3, with $S_1 \leq S_2$ and $S_1 \leq S_3$, then:*
[i] *$S_2 \leq S_3$ implies $\boldsymbol{lcp}(S_1, S_2) \geq \boldsymbol{lcp}(S_1, S_3)$ and $\boldsymbol{lcp}(S_2, S_3) \geq \boldsymbol{lcp}(S_1, S_3)$.*
Equivalently, *$\boldsymbol{lcp}(S_1, S_2) < \boldsymbol{lcp}(S_1, S_3)$ implies $S_2 > S_3$.*
[ii] *$\boldsymbol{lcp}(S_1, S_2) > \boldsymbol{lcp}(S_1, S_3)$ implies $\boldsymbol{lcp}(S_2, S_3) = \boldsymbol{lcp}(S_1, S_3)$.*

Proposition 1. *Given strings S_1, S_2, S_3, then:*
[i] *Identifying the smaller of S_1, S_2 and computing $\boldsymbol{lcp}(S_1, S_2)$ can be done in $O(\boldsymbol{lcp}(S_1, S_2))$ time.*
[ii] *Identifying j such that S_j is the smallest of S_1, S_2, S_3, and finding $\boldsymbol{lcp}(S_i, S_j)$ for $i \neq j$, can be done in $O(\max_{i \neq j} \boldsymbol{lcp}(S_i, S_j))$ time.*

Proof. Directly from Definition 1, by simply comparing the strings character by character (two characters for [i], and three at a time for [ii]). Note that Σ need not be bounded. □

3 Balanced Indexing Structures

The *Balanced Indexing Structure* (BIS, for short) [2] is an AVL search tree that indexes into online text T, by containing one suffix (of T) in each node. It is shown in [2] to allow the insertion of suffixes in an online manner, spending $O(\log n)$ time inserting each new suffix, and to support finding all occ occurrences of a pattern P in T in $O(|P| + \log |T| + occ)$ time.

BIS is *lcp*-based. Specifically, a BIS node v, associated with a suffix $S(v)$, maintains pointers to parent, $pa(v)$, and children, $le(v)$ and $ri(v)$, in the BIS, and to predecessor, $pr(v)$, and successor, $su(v)$, in the lexicographic order of all the suffixes in the BIS, and to the smallest and largest suffixes, $sm(v)$ and $la(v)$, in the subtree rooted at v. In addition, v maintains $lcp\_pr(v)$ being $\boldsymbol{lcp}(S(v), S(pr(v)))$, and $lcp\_extreme(v)$ being $\boldsymbol{lcp}(sm(v), la(v))$.

While suffixes of a given text are strongly correlated, the different strings in a collection are not definitely correlated. Moreover, when seeing a string for the first time we must at least read the string, whereas, suffixes are handled one after the other, so we are really seeing only the next character. Nevertheless, we show here that with a bit of adaptation, we can utilize the procedures of [2], and have the BIS work similarly as a data structure for maintaining a collection of n strings, where insert of a string S will cost $O(\log n + |S|)$, search of a string S will also cost $O(\log n + |S|)$ and deleting a string from the collection will cost $O(\log n)$.

The following lemma of the BIS is crucial to our setting. It is applicable to any collection of strings, and is only based on the data items described above that are maintained in the nodes.

Lemma 2. *(Based on Lemma 8 of [2]) Let p be a path leading from a node v_1 to any descendant v_2 thereof, via pointers $le(v)$ and $ri(v)$, then $\boldsymbol{lcp}(S(v_1), S(v_2))$ can be computed in $O(|p|)$ time.*

Proof. (sketch) (i) Let $u = le(v)$ (in analogy for the right child). Let $w = sm(ri(u))$ and $z = la(ri(u))$, then $w = su(u)$ and $v = su(z)$, implying $S(u) < S(w) < S(z) < S(v)$. Now, $lcp(S(u), S(w)) = lcp\_pr(w)$; $lcp(S(w), S(z)) = lcp\_exptreme(ri(u))$; and $lcp(S(z), S(v)) = lcp\_pr(v)$. With Lemma 1, we gain $lcp(S(v), S(le(v)))$ in constant time. (ii) If v_2 is reached from v_1 via $le()$ pointers only, then the lemma follows by repeated application of (i) and Lemma 1. (iii) Let y be any node reached from u via $ri()$ pointers only. Denote $w' = sm(ri(y))$, and note that $z = la(ri(y))$. Then, $w' = su(y)$ and $v = su(z)$, implying, as in (i), $S(y) < S(w') < S(z) < S(v)$, and thus yielding $lcp(S(v), S(ri(ri(\cdots ri(le(v)) \cdots)$ in constant time. (iv) Breaking path p into a zig-zag of alternating directions (left or right), and applying the above repeatedly yields the lemma. □

Inserting string S to a BIS rooted by string R thus starts with computing $l \leftarrow \boldsymbol{lcp}(S, R)$, while determining whether $S < R$, which indicates in which of R's subtree S continues. The strings in that subtree, as well as S, are either all smaller than R or all larger than R. Hence for each node v visited in the subtree, comparing l against $\boldsymbol{lcp}(S(v), R)$ – the *lcp* computed by Lemma 2, as

S goes down the tree – suffices, in case $l \neq \boldsymbol{lcp}(S(v), R)$, to determine whether S continues down left or down right from v. If $l = \boldsymbol{lcp}(S(v), R)$ then further characters, of S and $S(v)$, are read from position $l+1$ and on until $\boldsymbol{lcp}(S, S(v))$ is computed while whether $S < S(v)$ is determined, which indicates in which of v's subtree S continues. The strings in that subtree, as well as S, are either all smaller than $S(v)$ or all larger. Hence, as before, with $l \leftarrow \boldsymbol{lcp}(S, S(v))$, a comparison of l against $\boldsymbol{lcp}(S(u), S(v))$ for any node u in this subtree, can tell how S should go down from u, etc. until S is added to the BIS as a leaf. This all is done in $O(\log n + |S|)$ time, for a BIS of size n.

Removal of a string and extraction of the smallest string are not given in [2]. However, these only require the balancing procedure for the BIS, which was shown there for insertion, and can be done in $O(\log n)$ time. We thus have:

Lemma 3. *An (adapted) BIS maintains a collection of strings, where insertion takes $O(\log n + |S|)$, removal of a string takes $O(\log n)$ and extracting minimum takes $O(\log n)$.*

Corollary 2. *n strings of total length N can be sorted in time $O(n \log n + N)$.*

4 Heap Sorting of Strings

The well-known heap data structure is a full, balanced binary tree where each node's value is larger than its parent's value.

Definition 2. *The* heap of strings*, over a collection C of strings, is a full, balanced binary tree where each node v maintains a string $S(v) \in C$ and a field $lcp(v)$, satisfying the* heap of strings property*: $S(v) > S(parent(v))$ and $lcp(v) = \boldsymbol{lcp}(S(v), S(parent(v)))$. (If v is the root, $lcp(v)$ can be of any value.)*

A naive adjustment of the integer heap operations to string heap operations would replace each comparison of two integers by a comparison of two strings, employing a sequence of as many character comparisons as the lengths of the compared strings, in the worst case. This, however, **multiplies** runtime by about the average string length.

We will observe that our algorithms have each string "wander around" the heap always accompanied by the same *lcp* field, whose value never decreases. We will present an effective use of the *lcp* fields in the nodes, through which we can accomplish all the needed string comparisons by only **adding** a total of $O(N)$ steps.

4.1 Heapify

Commonly, a full balanced binary tree of n nodes is implemented by an array T of length n. The left and right children of node $T[i]$ (if any) are $T[2i]$, and $T[2i+1]$. Given an array T of n strings, the heapify procedure changes the positions of the strings, and their associated *lcp* fields, in T, so that the resulting tree is a *heap of strings*. When we swap two nodes' strings and *lcp* fields, we

say that we swap these nodes. As with the classic $O(n)$ time heapifying of an integer array, our algorithm proceeds for nodes $T[n], T[n-1], \ldots, T[1]$, making a *heap of strings* from the subtree rooted at the current node $v = T[i]$, as follows.
If v is a leaf, **return**.
If v has one child $u = T[2i]$ (which must be a leaf), compute $l = \boldsymbol{lcp}(S(v), S(u))$, assign l in the *lcp* field of the larger string owner. If that owner is v, swap u and v. **return**
Here v has two children, $u = T[2i]$ and $w = T[2i+1]$, and by the order we process the nodes, each child now roots a *heap of strings.* **find** the smallest, S, of the three strings $S(v), S(u), S(w)$, and compute the *lcp* of S with each of the other two. Assign the newly computed *lcp*s to the other two owners.
If $S = S(v)$, **return**
If $S = S(u)$ (the case $S = S(w)$ is analogous), swap v and u. Note that now w still roots a *heap of strings*, and in addition it satisfies the *heap of strings property.* In its new position, v and each of its children, now maintains an *lcp* of their string with S (now $= S(u)$, in the former position of v), which is smaller than **all** the strings in the subtree (now) rooted at v. Invoking SiftDown(v) from the new position of v ensures that the subtree it now roots is a *heap of strings.*

SiftDown(v)
We denote by $ch1(v)$ and $ch2(v)$ both left and right children of v, as applicable by the conditions on their field values, and its parent – by $pa(v)$. $S(v)$ and $lcp(v)$ are the string and *lcp* fields of node v.
Given: (1) All nodes in v's subtree have strings larger than $S(pa(v))$; (2) All these nodes, except, possibly, one or two children of v, satisfy the *heap of strings property*; and (3) v and each of its children have their *lcp* field being the *lcp* of their respective strings with $S(pa(v))$. **Swap** nodes and update *lcp* fields, making the resulting subtree rooted by v a *heap of strings.*

```
if $v$ has no children or $lcp(v)$ is greater than each child's $lcp$
    return /* $S(v)$ is a clear smallest, and all $lcp$ fields are OK */
if $lcp(v) < lcp(ch1(v))$, and: either $ch1(v)$ is $v$'s only child
    or $lcp(ch2(v)) < lcp(ch1(v))$ /* $ch1$ is a clear smallest */
    swap $v$ and $ch1(v)$, and in its new position apply SiftDown($v$) and return
if $lcp(v) = lcp(ch1(v)) = l$ and if $ch2$ exists then $lcp(ch2(v)) < lcp(ch1(v))$,
    or $lcp(v) < lcp(ch1(v)) = lcp(ch2(v)) = l$ /* smallest has its $lcp = l$ */
    read $S_1$ and $S_2$, the strings of the equal $lcp$-s owners, from position $l+1$ on,
    character by character, until $l' = \boldsymbol{lcp}(S_1, S_2)$ is computed while the smaller, $S$,
    of two strings is determined. Assign $l'$ to the $lcp$ field of the larger string owner.
    If $S$ is not $v$'s, replace $v$ with $S$'s owner, and there apply SiftDown($v$).
    return
/* $v$ has two children and $lcp(v) = lcp(ch1(v)) = lcp(ch2(v))$ */
    read $S(v)$, $S(ch1(v))$, and $S(ch2(v))$, in parallel, from position $lcp(v)+1$ on,
    until the smallest, $S$, of the three is determined, as the $lcp$ of it with each of
    the other strings is computed. Assign newly computed $lcp$-s to respective
    owners of the other two strings.
    if $S$ is not $v$'s, swap $v$ and $S$'s owner and there apply SiftDown($v$).
```

Correctness of our algorithm follows from the known heapify for integers and Corollary 1. As to runtime, note that we process $O(n)$ nodes (including the nodes processed in the recursive calls), in each we compare *lcp* values and if needed, continue the comparison of two or three strings. In such a comparison of strings, two or three characters (as the number of strings compared) are compared at a time. The number of such character comparisons exceeds by one the extent by which one or two *lcp* values are increased (because we pass over all the equal characters until reaching inequality). We thus have:

Proposition 2. [i] *for every (two or three) string comparison, at least one of these strings has its associated lcp field increased by the number of character comparisons incurred, minus 1.* [ii] *Once a string has its associated lcp field assigned the value of l, none of its characters in positions $1, 2, \ldots, l$ participates in any further character comparison.*

Corollary 3. *Heapifying into a heap of strings can be done in $O(n + N)$ time.*

4.2 Extracting Minimal String

Having built a *heap of strings* H of size n, we now extract the smallest string therefrom, which resides at the root of H, and invoke **PumpUp**($root(H)$) which discards one node from tree H, while organizing the remaining nodes in a tree, with each of them satisfying the *heap of strings property.*

PumpUp(v)
We denote by $ch1(v)$ and $ch2(v)$ both left and right children of v, as applicable by the conditions on their field values. $S(v)$ and $lcp(v)$ are the string and *lcp* fields of node v.
Given: (1) v has a null string and a non relevant *lcp* (2) All nodes in v's subtree, other than v and its children, satisfy the *heap of strings property.* (3) The *lcp* in each of v's children reflects the *lcp* of their string with the string formerly residing in v, which is smaller than each child's string. **Arrange** such that: (1) along a path from v down to one leaf, strings and their associated *lcp*-s climb up one node, so that one node becomes redundant and can be discarded from the tree. (2) All remaining nodes satisfy the *heap of strings property.*

```
if $v$ is a leaf, discard $v$ from the tree and return.
if $v$ has one child, discard $v$ and have this child take its place. return
if $lcp(ch1(v)) > lcp(ch2(v))$ /* ch1 is smaller */
    swap $v$ and $ch1(v)$, and in its new position apply PumpUp($v$), and return.
/* $v$ has two children, with equal $lcp$-s: $l = lcp(ch1(v)) = lcp(ch2(v))$ */
    read $S(ch1(v))$ and $S(ch2(v))$ from position $l + 1$ on, character by character, until
    $l' = \boldsymbol{lcp}(S(ch1(v)), S(ch2(v)))$ is computed, and the smaller of the two strings is
    determined. Assign $l'$ as the $lcp$ of the larger string owner, say $ch1$.
    swap $v$ and $ch2(v)$, and in its new position apply PumpUp($v$), and return.
```

Correctness of **PumpUp** follows directly from Corollary 1. Observe that now H is not necessarily full and not necessarily balanced, but its height is $O(\log n)$, and hence any further extractions of minimal string does not process more than

$O(\log n)$ nodes. Note that, as with our heapify procedure, in each node we compare *lcp*-s and if needed, continue the comparison of two strings by sequentially comparing characters. The number of such character comparisons exceeds by one the extent by which one *lcp* increases. None of the *lcp* fields decreases from the value it had at the end of the heapify process.

In fact, if we start with heapify, and then extract smallest string by smallest (over the remaining) string, we lexicographically sort the set of n input strings.

Corollary 4. *Sorting of n strings of total length N, over unbounded alphabet, can be done in $O(n \log n + N)$ worst case time, using $O(n)$ auxiliary space.*

4.3 On-Demand Sorting

As construction time for a *heap of strings* is $O(n)$, smaller than the $O(n \log n)$ for BIS, the *heap of strings* is better suited for cases where we will need only an (unknown) fraction of the strings from the lexicographic order.

Corollary 5. *On Demand Sorting of n strings of total length N can be done with the retrieval of the first result in $O(n + N_1)$ time, after which the retrieval of further results in $O(\log n + N_i)$ time for the i-th result, with $\sum_i N_i \leq N$.*

Proof. Using *heap of strings*, the first result can be extracted immediately after heapifying. Further results are extracted each following a **PumpUp** that rearranges the *heap of strings*, of height $O(\log n)$, following the previous extraction. Through the whole process *lcp* fields never decrease, and each character comparison (except for one per node) incurs an *lcp* increase. □

4.4 Find the Smallest k Strings

When we know in advance that we will only need the $k < n$ smallest strings from the input set of n strings of total length N, we can use a *heap of strings* of size k, and achieve the goal in $O(n \log k + N)$ time as follows. We use a heap of size k where parents hold **larger** strings than their children, and hence the largest string in the heap resides at the root node. We heapify the first k strings of the input set into such a heap, in analogy with our heapify process above. Then, for each string S of the remaining $n - k$ strings in the input, we compare S with string R at the root, while finding $\boldsymbol{lcp}(S, R)$. If S is found greater than R, it is discarded. Otherwise, R is discarded, and S finds its place in the heap using procedure **SiftDown** adopted for heaps where parents hold larger strings than their children.

After this process, the heap of size k holds the smallest k strings in the collection. We can now sequentially extract the (next) largest string therefrom, retrieving the k smallest strings in decreasing lexicographic order, using **PumpUp** adopted for heaps where parents hold larger strings than their children. As here, too, *lcp* fields only grow, we conclude that:

Corollary 6. *Finding (and sorting) the smallest k strings of a given input set of n strings of total length N, can be done in $O(n \log k + N)$ time, using $O(k)$ auxiliary space.*

4.5 Insertion of Strings to a Heap of Strings

Heap build-up linearity in number of nodes does not apply for post build-up insertions. Namely, inserting an additional element to an existing heap of size n incurs the processing of $O(\log n)$ nodes, not $O(1)$. Moreover, insertion of data elements to a heap by the known heap algorithm causes some nodes to have their parents replace their data elements by a smaller one, without them (the children) replacing their data elements. Hence, with a *heap of strings*, by Corollary 1, the *lcp* field of such a child node, which needs to reflect the *lcp* of its string with its parent's string, might decrease.

More specifically, here is how we can insert a string, S, to a *heap of strings*, H, of size n, in $O(\log n + |S|)$ time (as with BIS). H remains a full balanced *heap of strings*. Only a few *lcp* fields therein might decrease. We first add an empty leaf node, denoted $leaf$, to H, such that H is still balanced. We then invoke procedure **InsertString** which finds the right position for S in the path leading from H's root to $leaf$, and pushes the suffix of the path from that point down, making a room for S, while updating *lcp* fields of nodes out of the path whose parents now hold smaller strings than before.

InsertString(H, S)
Given: A *heap of strings* H of size n plus a newly added empty leaf, $leaf$, whose *lcp* is set to -1, and a string S. **Insert** S in a position along *path*, being the path $root = n_1, n_2, \ldots, n_m = leaf$, making H a *heap of strings* of $n+1$ nodes.

```
1 Compare $S$ with the root string $R$, while computing $l = \textbf{\textit{lcp}}(S, R)$.
2 if $S > R$
3     for $i \leftarrow 2$ to $m$ while $l \leq lcp(n_i)$ /* $S \geq S(n_i)$ */
4         if $l = lcp(n_i)$ read strings $S$ and $S(n_i)$ from position $l+1$ on, until
                $l' \leftarrow \textbf{\textit{lcp}}(S, S(n_i))$ is computed and whether $S < S(n_i)$ is determined.
5             if $S < S(n_i)$ PushDown$(H, S, path, i, l', l)$ and then return
6             $l \leftarrow l'$
7     PushDown$(H, S, path, i, lcp(n_i), l)$
8 else /* $S < R$ */ PushDown$(H, S, path, 1, l, 0)$
```

Procedure **PushDown** actually modifies the heap the same as does the classic Sift-up process of inserting a new element to an existing heap. In addition, **PushDown** also updates *lcp* fields as necessary. This is where some of these fields might decrease.

The decrease in *lcp* values prevents from ensuring overall $O(N)$ characters comparisons for heap buildup followed by some string insertions, followed by smallest string extractions. Number of nodes processed, however, remains $O(n)$ for heap buildup, followed by $O(m \log(n+m))$ for the insertion of additional m strings, followed by $O(\log(m+n))$ for each smallest string extraction.

PushDown$(H, S, path, i, l_i, l_{i-1})$
Given A *heap of strings* H of size n plus an empty leaf, $leaf$; a string S; path $root = n_1, n_2, \ldots, n_m = leaf$; an index i for S to be inserted between n_{i-1} and n_i in the path; *lcp* value $l_{i-1} = \textbf{\textit{lcp}}(S(n_{i-1}), S)$; and $l_i = \textbf{\textit{lcp}}(S(n_i), S)$. **Push** nodes

$n_i, n_{i+1}, \ldots, n_{m-1}$ down the path, and **place** S in the evacuated position, while maintaining H a *heap of strings*.

```
for $j = m$ down to $i + 1$ /* push down end of path */
    $S(n_j) \leftarrow S(n_{j-1})$; $lcp(n_j) \leftarrow lcp(n_{j-1})$
    $lcp(sibling(n_{j+1})) \leftarrow \min\{lcp(sibling(n_{j+1})), lcp(n_{j+1})\}$ /* $sibling(n_{j+1})$ has its
        /* grandparent become its new parent, which might decrease its $lcp$ */
$lcp(n_{i+1}) \leftarrow l_i$; $S(n_i)) \leftarrow S$; and $lcp(n_i) \leftarrow l_{i-1}$
$lcp(sibling(n_{i+1})) \leftarrow \min\{lcp(sibling(n_{i+1})), l_i\}$ /* $S$ is new $sibling(n_{i+1})$'s parent */
```

4.6 Inserting Strings Without Decreasing *lcp*-s

One way to avoid decreasing of *lcp*-s is to insert single-childed nodes. This comes, however, at the expense of tree balancing. Before invoking InsertString, do not prepare an empty leaf node. Rather, once the right point in *path*, between n_{i-1} and n_i is found by InsertString, instead of invoking PushDown, add a new, empty node to H, and place S there, making it n_{i-1}'s child instead of n_i, and making n_i its only child. As before, number of characters processing does not exceed $|S|$. Now, however, no *lcp* field needs to decrease. Let n denote the size of the heap upon its build up, and m the number of strings thus inserted the heap post build up. It is easy to see that further extractions, using our PumpUp above, would not incur more than $O(\log n)$ node processing, since this procedure stops once a single-childed node is reached. Further insertions, however, although not exceeding $O(|S|)$ character comparisons, might incur the processing of $O(\log n + m)$ nodes, much greater than $O(\log(n + m))$ which BIS processes.

5 Heap of Strings Embedded with Binary Search Trees

In this section we circumvent the non-balancing introduced in Subsection 4.6. Generally speaking, we embed chains of single-childed nodes on a BIS, which is *lcp* based. We start with heapify a *heap of strings* from the initial set of n strings. Then, for each string arriving post heapifying, we insert it as in Subsection 4.6. The path of single-childed newly born nodes, which hold strings in increasing order, is arranged on a BIS and is **not** considered part of the *heap of strings*.

The BIS thus born is pointed at from both nodes of the *heap of strings*, above and below it. The smallest node in that BIS will maintain, in its *lcp_prev* field, the *lcp* of its string and the string of the *heap of strings* node above the BIS.

As in Subsection 4.6, before calling InsertString we do not prepare an empty leaf node. Rather, when InsertString finds the right point to insert S, between n_{i-1} and n_i of the *heap of strings*, instead of invoking PushDown, S will join the BIS that resides between these two nodes (or start a new BIS if there is not any) as described in Section 3 (See Fig. 1). When we are about to insert S to the BIS "hanging" between n_{i-1} and n_i, we have already computed $l = \boldsymbol{lcp}(S, S(n_{i-1}))$. All the BIS strings, as well as S, are larger than $S(n_{i-1})$, and the *lcp* fields maintained in the BIS suffice to compute, in constant time, the value of $\boldsymbol{lcp}(R, S(n_{i-1}))$, for the BIS root R. Hence, the first step of inserting S to the BIS, namely the comparison against R, can be done without reading $S[1,l]$ again. Thus, the insertion of string S to a *heap of strings* that was heapified with n strings, incurs $O(\log n + \log m + |S|)$ time, with m denoting the size of the BIS to include S.

Extracting of strings from the root of the heap can continue as before, using PumpUp, which now, if visits a node v of the *heap of strings* which has a BIS leading from it to its child in the *heap of strings*, sees the smallest node of the BIS as v's single child, and pumps this child up to take the place of v. This child is removed from the BIS and becomes a member of the *heap of strings*, and this completes the execution of PumpUp (it does not go further down). The removal of a node from BIS of size m takes $O(\log m)$ time, and our PumpUp incurs the processing of $O(\log n)$ nodes plus reading as many characters as needed, while increasing *lcp* fields.

Note that a string arriving post heapify may spend "some time" in a BIS, but once removed from there and joined the *heap of strings*, it will never go into any BIS again. Only newly arriving strings may go into a BIS. We conclude that:

Theorem 1. *It is possible to construct a heap of n strings in $O(n)$ time and support string insertion and smallest string extraction in $O(\log(n) + \log(m))$ time, with an additional $O(N)$ amortized time over the whole sequence of operations, where m is the number of strings inserted post heapifying and were not extracted yet.*

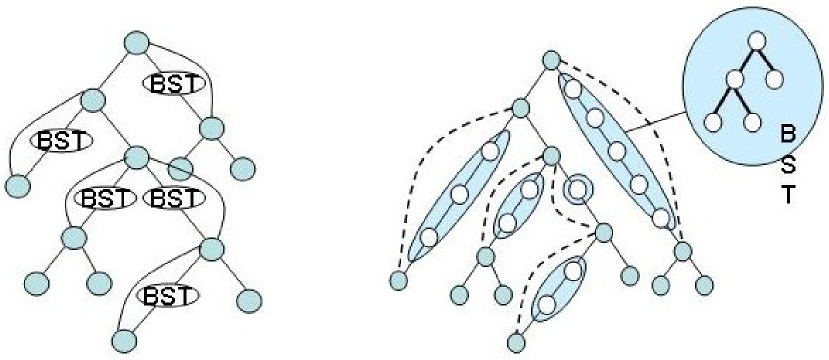


Fig. 1. *heap of strings* embedded with *Balanced Indexing Structure*

References

1. Aho, A., Hopcroft, J., Ullman, J.: The Design and Analysis of Computer Algorithms. Addison-Wesley, Reading, MA (1974)
2. Amir, A., Kopelowitz, T., Lewenstein, M., Lewenstein, N.: Towards Real-Time Suffix Tree Construction. In: Proc. of Symp. on String Processing and Information Retrieval (SPIRE), pp. 67–78 (2005)
3. Iyer, B.R.: Hardware assisted sorting in IBM's DB2 DBMS. In: International Conference on Management of Data, COMAD 2005b, Hyderabad, India (December 20-22, 2005)

4. Arge, L., Ferragina, P., Grossi, R., Vitter, J.S.: On sorting strings in external memory. In: Symposium of Theory of Computing (STOC), pp. 540–548 (1997)
5. Gonnet, G.H., Baeza-Yates, R.: Handbook of Algorithms and Data Structures. Addison-Wesley, Reading (1991)
6. Baer, J.-L., Lin, Y.-B.: Improving Quicksort Performance with a Codeword Data Structure. IEEE Transactions on Software Engineering 15, 622–631 (1989)
7. Bentley, J.L., Sedgewick, R.: Fast algorithms for sorting and searching strings. In: Proc. of Symposium on Discrete Algorithms (SODA), pp. 360–369 (1997)
8. Farach, M.: Optimal suffix tree construction with large alphabets. In: Proc. 38th IEEE Symposium on Foundations of Computer Science, pp. 137–143 (1997)
9. Farach-Colton, M., Ferragina, P., Muthukrishnan, S.: On the sorting-complexity of suffix tree construction. J. of the ACM 47(6), 987–1011 (2000)
10. Grossi, R., Italiano, G.F.: Efficient techniques for maintaining multidimensional keys in linked data structures. In: Wiedermann, J., van Emde Boas, P., Nielsen, M. (eds.) ICALP 1999. LNCS, vol. 1644, pp. 372–381. Springer, Heidelberg (1999)
11. Hagerup, T.: Optimal parallel string algorithms: sorting, merging and computing the minimum. In: Proc. of Symposium on Theory of Computing (STOC), pp. 382–391 (1994)
12. Hagerup, T., Petersson, O.: Merging and Sorting Strings in Parallel. Mathematical Foundations of Computer Science (MFCS), pp. 298–306 (1992)
13. JaJa, J.F., Ryu, K.W., Vishkin, U.: Sorting strings and constructing difital search tries in parallel. Theoretical Computer Science 154(2), 225–245 (1996)
14. Kärkkäinen, J., Sanders, P.: Simple linear work suffix array construction. In: Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.) ICALP 2003. LNCS, vol. 2719, pp. 943–955. Springer, Heidelberg (2003)
15. Kim, D.K., Sim, J.S., Park, H., Park, K.: Linear-time construction of suffix arrays. In: Baeza-Yates, R.A., Chávez, E., Crochemore, M. (eds.) CPM 2003. LNCS, vol. 2676, pp. 186–199. Springer, Heidelberg (2003)
16. Knuth, D.: the Art of Computer Programming. Sorting and Searching, vol. 3. Addison-Wesley, Reading, MA (1973)
17. Ko, P., Aluru, S.: Space efficient linear time construction of suffix arrays. In: Baeza-Yates, R.A., Chávez, E., Crochemore, M. (eds.) CPM 2003. LNCS, vol. 2676, pp. 200–210. Springer, Heidelberg (2003)
18. Manber, U., Myers, E.W.: Suffix arrays: A new method for on-line string searches. SIAM J. on Computing 22(5), 935–948 (1993)
19. McCreight, E.M.: A space-economical suffix tree construction algorithm. J. of the ACM 23, 262–272 (1976)
20. Mehlhorn, K.: Dynamic Binary Search. SIAM J. Comput. 8(2), 175–198 (1979)
21. Munro, J.I., Raman, V.: Sorting multisets and vectors inplace. In: Proc. of Workshop on Algorithms and Data Structures (WADS), pp. 473–479 (1991)
22. Sinha, R., Zobel, J., Ring, D.: Cache-efficient string sorting using copying. J. Exp. Algorithmics 11, 1084–6654 (2006)
23. Ukkonen, E.: On-line construction of suffix trees. Algorithmica 14, 249–260 (1995)
24. Scott Vitter, J.: External memory algorithms. In: Handbook of massive data sets, pp. 359–416. Kluwer Academic Publishers, Norwell, MA, USA (2002)
25. Weiner, P.: Linear pattern matching algorithm. In: Proc. 14th IEEE Symposium on Switching and Automata Theory, pp. 1–11 (1973)
26. IBM OmniFind Enterprise Edition: Programming Guide and API Reference for Enterprise Search. Sorting by relevance, date, numeric fields, or text fields, p. 30. http://publibfp.boulder.ibm.com/epubs/pdf/c1892843.pdf

Finding Witnesses by Peeling

Yonatan Aumann[1], Moshe Lewenstein[1], Noa Lewenstein[2], and Dekel Tsur[3]

[1] Bar-Ilan University
{aumann,moshe}@cs.biu.ac.il
[2] Netanya College
noa@netanya.ac.il
[3] Ben-Gurion University
dekelts@cs.bgu.ac.il

Abstract. In the k-matches problem, we are given a pattern and a text, and for each text location the goal is to list all, but not more than k, matches between the pattern and the text. This problem is one of several string matching problems that ask to not only to find where the pattern matches the text, under different "match" definitions, but also to provide *witnesses* to the match. Other such problems include: k-aligned ones [4], k-witnesses, and k-mismatches [18]. In addition, the solution to several other string matching problems relies on the efficient solution of the witness finding problems.

In this paper we provide a general efficient method for solving such witness finding problems. We do so by casting the problem as a generalization of group testing, which we then solve by a process which we call *peeling*. Using this general framework we obtain improved results for all of the above problems. We also show that our method also solves a couple of problems outside the pattern matching domain.

1 Introduction

Pattern matching is a well studied domain with a large collection of problems. In classical pattern matching one desires to find all appearances of a pattern with a text [14]. One avenue of research within pattern matching is the extension of this problem to allow for more general matching criteria. Another is to allow a bounded number of errors to occur. In both of these extensions we announce the locations that match, or the locations which mismatch, under the extended criteria or other limitations.

In several applications it is not sufficient to announce the locations which matched or mismatched. Rather we will want a *witness* to the match or mismatch. For example consider the classical pattern matching problem. If we have a pattern p = 'aba' and a text t = 'aabaa'. Then the pattern matches at text location 2 but not at any other text location. The $|p|$-length substring beginning at text location 1 is 'aab'. A witness to the mismatch at location 1 is the second character of the pattern as it is 'b' in p and 'a' in 'aab'. Of course, match is a general term which is application dependent and also witness needs can change per application. Finding witnesses arises in several applications which we mention below. However, doing so efficiently is not a simple task.

The concept of finding a "witness" was independently introduced for applications of *boolean matrix multiplication*, specifically for the all-pairs shortest path. Formally let C be the (boolean) multiplication of A and B. Then if $c_{i,j} = 1$, we say that t is a *witness* if $A_{i,t} = 1$ and $B_{t,j} = 1$.

We now bring several applications where finding k witnesses is desired. To solve these problems we define a generalization of group testing which is a process in which we "peel witnesses", called the *reconstruction problem.* We solve several variants of the reconstruction problem in this paper and utilize them to solve our witness problems. We believe that the peeling process is interesting in its own right, although we view the main contribution of the paper as a generalized setting and solution for the set of problems mentioned below. We show, at the end of the introduction how these problems fit into the setting of our problem.

Aligned Ones and Matches. Let $t = t_0, \ldots, t_{n-1}$ be a text and $p = p_0, \ldots, p_{m-1}$ be a pattern. For a text location i, we say that there is a *match* at position j if $t_{i+j} = p_j$, i.e. when aligning the beginning of p at location i of the text, the j-th position of the pattern matches that of the text. The *k-aligned ones problem*, defined in [4] and further explored in [18], is the following. For text and pattern both over the alphabet $\Sigma = \{0, 1\}$, for each text location i where there are at most k matches of the character 1, output all of these matches (in this problem, we ignore matches of the character 0). In the *k-witnesses problem*, we need to find k matches of the character 1 for every text location i (if there are fewer than k matches, we need to find all of the matches). The solution in [4] to the k-aligned ones problem is a very interesting, but a rather involved and costly procedure. In [18] a simpler solution was suggested and it hints at the direction we take.

In the *k-matches problem* for each text location i, output at least k matches; if there are less than k matches, output all matches. Here the problem is over general alphabets (i.e. not restricted to $\{0, 1\}$). This problem has appeared in a couple of applications and is a useful tool in pattern matching.

Mismatches under General Matching Relations. A similar problem to the above is the k-mismatches problem. Here the problem is to find all text location where the number of mismatches does not exceed k. This problem is well-studied (e.g. [1,5,11,16]), and the most efficient algorithm [5] runs in time $O(n\sqrt{k \log k})$. The algorithms of [11,16], which run in time $O(nk)$, allow to also report the positions of these mismatches.

All of the above works assume that the matching relation (between pattern elements and text elements) is the equality relation. In many application, however, other "matching" relations may arise (e.g. matching between color pictures with different color coding schemes). In such cases, the matching is defined by a $\Sigma \times \Sigma$ matching matrix M, where Σ is the alphabet. The *k-mismatches problem* [18] is the problem of finding for each text location i, at least k mismatches (under M); if there are less than k mismatches, find all mismatches.

For general matching relations M the suffix tree methods of [11,16] do not work. Muthukrishnan [18, Theorem 5] shows a relationship between the k-mismatches problem and the k-witnesses problem.

Half-Rectangular Matching. In half-rectangular matching one is given a 2-dimensional text t and a 2-dimensional pattern p which contains variable length rows (the half-rectangular property). Searching for the appearances of p in t is known as the half-rectangular matching problem [4].

Amir and Farach [4] provide an efficient algorithm for the half-rectangular matching problem, which uses an algorithm for the k-aligned ones problem as a subroutine. We will obtain improved algorithms for the half-rectangular matching problem based on improving the k-aligned-ones problem.

Matrix Multiplication and All-Pairs Shortest Path Problem. Let A and B be two boolean $n \times n$ matrices, and let $C = A \cdot B$, where multiplication is the logical $\wedge$ and addition the logical $\vee$. The *witness for boolean matrix multiplication problem* is for each i, j such that $c_{i,j} = 1$ find a t such that $a_{i,t} = b_{t,j} = 1$. Alon, Galil, Margalit, and Naor [2] and Seidel [19] independently discovered a simple randomized algorithm to solve the problem. Both consider the concept of witness for boolean matrix multiplication problem to solve the all-pairs shortest path problem on undirected unweighted graphs.

A natural extension to the all-pairs shortest path problem is the problem of the *all-pairs k-shortest paths*. That is for every pair of vertices u, v we desire to output the k (not necessarily disjoint) shortest paths between u and v (if they exist). Naturally, one way to solve this problem is to extend the witness problem to the *k-witnesses for boolean matrix multiplication problem*: Let $C = A \cdot B$ with multiplication over the reals. For each i, j, find $k' = \min(k, c_{i,j})$ "witnesses" for $c_{i,j}$, i.e. indexes $t_1, \ldots, t_{k'}$, such that for each $t \in \{t_1, \ldots, t_{k'}\}$, $a_{i,t} = b_{t,j} = 1$. Hence, the all-pairs k-shortest paths problem can be solved in the same time complexity as the k-witnesses for boolean matrix multiplication problem. However, one cannot simply reiterate the procedure of the one witness problem as it may rediscover the same witness once again.

1.1 The Reconstruction Problem

We first give a group testing setting and then return to show how this helps to solve our above-mentioned problems.

Let U be a universe of m distinct items, all taken from some commutative group G. In *group testing* one desires to determine the members of a set $S \subseteq U, |S| \leq k$, by queries of the form "does A contain an element of S?", where A is some subset of U. The goal of group testing is to determine S with a minimal number of queries.

Many variants of group testing have been considered. One of these variants is a change of query asking for $|S \cap A|$ rather than whether $S \cap A$ is empty or not. This problem of producing a series of non-adaptive queries $A_1, \ldots, A_l$ is known as *quantitative group testing*, see [10], or *interpolation set* [12,13]. In *additive group testing*, a query asks for the value of $\sum_{u \in S \cap A} u$ for some set A.

Here we consider a generalization of the above. Consider a collection of unknown sets $S_1, \ldots, S_n \subseteq U$, and suppose that we are interested in reconstructing $S_1, \ldots, S_n$, or in finding k elements of each S_i (for some parameter k). We are provided with procedures such that for any set $A \subseteq U$, we can compute:

– Intersection Cardinality: $\text{ISize}(S_1, \ldots, S_n, A) = \langle |S_1 \cap A|, \ldots, |S_n \cap A| \rangle$.
– Intersection Sum: $\text{ISum}(S_1, \ldots, S_n, A) = \langle \sum_{u \in S_1 \cap A} u, \ldots, \sum_{u \in S_n \cap A} u \rangle$.

Clearly, given a sufficient number of calls to these procedures, it is possible to fully reconstruct $S_1, \ldots, S_n$. However, we aim at doing so with a minimal amount of work. We note that an efficient algorithm for this problem must provide:

1. The list or sequence of sets A on which to compute $\text{ISize}(S_1, \ldots, S_n, A)$ and/or $\text{ISum}(S_1, \ldots, S_n, A)$ (note that for determining the A's we do not have explicit knowledge of $S_1, \ldots, S_n$),
2. An efficient procedure to reconstruct k elements of each S_i based on the information provided by these computations.

In this paper we consider two variants of the general setting outlined above. In the first variant, the size of $S_1, \ldots, S_n$ can be arbitrary, and we are interested in finding $\min(k, |S_i|)$ elements of S_i for all i. We call this the *k-reconstruction problem*. In the second variant, the size of each S_i is known to be bounded by k, and we are interested in fully reconstructing $S_1, \ldots, S_n$. We call this the *bounded k-reconstruction problem*. We present efficient solutions to both these problems. The solutions are based upon a combinatorial structure, which we call *peeling collections*, and a matching algorithmic procedure, which we call the *peeling procedure*. The basic idea is as follows. If a set $A \subseteq U$ satisfies $|S_i \cap A| = 1$, then we can "peel" the unique element of the intersection of S_i by querying $\sum_{u \in S_i \cap A} u$. A *peeling collection* is a collection of sets such that for any S its elements can be peeled one by one by peeling and updating the other sets to reflect that an element has been peeled. The full details appear in section 2.

We point out the following differences from previous works:

1. Most group testing codes are universal in the sense that the elements of any set S can be recovered using the codes. In [8] a lower bound of $\Omega(k \log m)$ was given on the construction of k-selective codes equivalent to the deterministic bounded-reconstruction (see section 2). Here we consider reconstructing a single set S (which one can view as a special code for one unknown S) with high probability beating the lower bounds for the k-selective codes.
2. The problem is an n-set problem as opposed to the one-set problems in the previous variants. While the problem can be solved by finding separate tests for each set, this sometimes can be done more efficiently directly. Moreover, we show applications that fall nicely into this setting.
3. The peeling method. This gives an adaptive flavor to a non-adaptive setting. It should be mentioned that the peeling method appears implicitly in [13] yet is rather inefficient and unexploited.

As far as we know, the k-reconstruction problems have not been studied explicitly before. However, in previous works, these problems are solved implicitly:

Amir et al. [4] implicitly give a deterministic algorithm that solves the bounded k-reconstruction problem. A deterministic algorithm for this problem can also be obtained from [13] with some simple observations.

For the unbounded problem, Seidel [19] and Alon and Naor [3] implicitly solve the unbounded 1-reconstruction problem. Muthukrishnan [18] implicitly gives a randomized algorithm for solving the unbounded k-reconstruction problem.

New results. We now describe our results for the reconstruction problem, which improve the previous (implicit) results. Suppose that computing $\textsc{ISize}(S_1, \ldots, S_n, A)$ or $\textsc{ISum}(S_1, \ldots, S_n, A)$ takes $O(f)$ steps. For the bounded k-reconstruction problem we present a deterministic algorithm that solves the problem in $O(k \cdot \mathrm{polylog}(m)(f+n))$ steps, and a randomized algorithm that solves the problem in expected $O(f(k + \log k \cdot \log n) + nk \log(mn))$ steps. For the (unbounded) k-reconstruction problem, we give a deterministic algorithm with time complexity $O(k \cdot \mathrm{polylog}(mn)(f+n))$, and a randomized algorithm with expected time complexity $O(f(k(\log m + \log k \log\log m) + \log n \log m) + nk \log(mn))$.

Given the efficient solutions to the general problems, we show problems can be cast as special cases of this general framework. Accordingly, we provide improved algorithms for these problems. We now list the new results. In the following, let $T_b(n, m, k, f)$ (resp., $T_u(n, m, k, f)$) be the time needed to solve the bounded (resp., unbounded) k-reconstruction problem with parameters n, m, k, and f.

1.2 Our Results

Aligned Ones and Matches. For the k-aligned ones problem, Amir and Farach [4] give an $O(nk^3 \log m \log k)$ time deterministic algorithm, and Muthukrishnan [18] provides an $O(nk^2 \log^4 m)$ randomized algorithm. Muthukrishnan [17] improved this to an $O(nk \log^4 m)$ randomized algorithm.

We show that the k-aligned ones problem can be solved in time $O(\frac{n}{m} T_b(2m, m, k, m \log m))$. Therefore, we obtain an $O(nk \cdot \mathrm{polylog}(m))$ deterministic algorithm, and an $O(n \log m(k + \log k \cdot \log m))$ randomized algorithm for the problem. Moreover, the k-matches problem can be solved in $O(|\Sigma| \frac{n}{m} T_u(2m, m, k, m \log m))$ time.

Mismatches under General Matching Relations. Muthukrishnan[18, Theorem 5] shows that the k-mismatches problem over arbitrary M can be solved with $cpn(G)$ calls to the k-witnesses problem, where G is the conflict graph (i.e. the bipartite graph corresponding to the complement of M) and $cpn(G)$ is the minimum number of edge-disjoint bipartite cliques of G. Additionally, [18] provides an $O(n\sqrt{km \log m})$ deterministic algorithm, and an expected $O(nk^2 \log^3 m \log n)$ randomized algorithm to the k-witnesses problem. An open problem presented in [18] was to find a deterministic algorithm for the k-witnesses problem with time comparable to the randomized algorithm.

The k-witnesses problem can be solved in $O(\frac{n}{m} T_u(2m, m, k, m \log m))$ time, and therefore, we obtain an $O(nk \cdot \mathrm{polylog}(m))$ deterministic algorithm and an $O(n \log m(k(\log m + \log k \log\log m) + \log^2 m))$ randomized algorithm for this problem.

Matrix Multiplication Witnesses and All-Pairs k-Shortest Paths. Alon, Galil, Margalit and Naor [2] and Seidel [19] independently discovered a simple randomized algorithm to solve the problem of finding a witness for Boolean matrix multiplication that runs in expected $O(M(n)\log n)$ time, where $M(n)$ is the time of the fastest algorithm for matrix multiplication. Alon and Naor [3] derandomized the algorithm providing an $O(M(n)\cdot \text{polylog}(n))$ deterministic algorithm for this problem.

Using our construction we solve the k-witness problem for matrices in $O(M(n)\cdot k\cdot \text{polylog}(n))$ deterministic time, and $O(M(n)(k+\log n))$ randomized time. This yields a solution to the all-pairs k-shortest paths problem with the same time complexity as the k-witnesses for the Boolean matrix multiplication problem.

2 The Peeling Processes

2.1 Peeling Collections and the Peeling Process

Consider the k-reconstruction problem which was defined in the introduction. Recall that a call to the procedures ISize$(S_1,\ldots,S_n,A)$ or ISum$(S_1,\ldots,S_n,A)$ takes $O(f)$ steps. We are interested in finding k elements of each set S_i with minimal work. Thus, we seek an algorithm that both uses few calls to these procedures and allows for efficient reconstruction of the k elements based on the results obtained from these calls.

Separating Collections. For sets S and A we say that A *peels* S if $|S\cap A|=1$. If $S\cap A=\{s\}$ we say that A peels the element s.

Definition 1. *Let S be a set and F a collection of sets. We say that F is a* k-separator for S *if there are sets $A_1,\ldots,A_{\min(k,|S|)}\in F$ such that:*

1. *for each i, A_i peels S,*
2. *for $i\neq j$, $S\cap A_i\neq S\cap A_j$ (i.e. the sets peel different elements of S).*

For a collection of sets $\mathcal{S}=\{S_1,\ldots,S_n\}$ we say that F is a k-separator for $\mathcal{S}$ *if it is a k-separator for each $S\in\mathcal{S}$.*

Suppose that F is a k-separator for S. Then, it is possible to reconstruct $k'=\min(k,|S|)$ elements of S by simply invoking the procedure ISize(S,A) for all $A\in F$. For each A such that ISize$(S,A)=1$ compute $s_A=$ ISum(S,A). The element s_A is necessarily a member of S, and since F is a k-separator, there must be at least k' distinct such s_A's.

Peeling Collections and the Peeling Procedure. A separating collection requires that for each element $s\in S$ to be recovered, there is a *distinct* set $A\in F$ that peels s. We now show how to reconstruct k' elements of S with a weaker condition. We do so by iteratively "peeling off" elements from S.

Let $P=(A_1,\ldots,A_k)$ be a sequence of sets. We say that P is a *peeling sequence for S* if for $i=1,\ldots,k$, A_i peels $(S-\cup_{j=1}^{i-1}A_j)$. Let F be a collection of sets, and $P=(A_1,\ldots,A_k)$ a sequence of sets. We say that F *contains* P if $A_1,\ldots,A_k\in F$.

Definition 2. *Let S be a set and F a collection of sets. We say that F is a* k-peeler for S *if F contains a peeling sequence for S of length $k' = \min(k, |S|)$. We say that F is a* bounded k-peeler for S *if it is a k-peeler for S provided that $|S| \le k$.*

For a collection of sets $\mathcal{S} = \{S_1, \ldots, S_n\}$ we say that F is a (bounded) k-peeler for $\mathcal{S}$ *if it is a (resp., bounded) k-peeler for each $S \in \mathcal{S}$.*

Note that a k-separator for a set S is also a k-peeler for S. Let F be a k-peeler for S. Consider the following process, which we call *the Peeling Procedure*:

```
1  For each $A \in F$ assign $z_A \leftarrow \textsc{ISize}(S, A)$ and $m_A \leftarrow \textsc{ISum}(S, A)$
2  $\textsc{Ones} \leftarrow \{A \in F : z_A = 1\}$
3  While $\textsc{Ones}$ is not empty do
4      Choose any $A \in \textsc{Ones}$ and remove $A$ from $\textsc{Ones}$
5      $s_i \leftarrow m_A$
6      For each $A' \in F$ such that $s_i \in A'$ do
7          $z_{A'} \leftarrow z_{A'} - 1$
8          If $z_{A'} = 1$ then add $A'$ to $\textsc{Ones}$
9          $m_{A'} \leftarrow m_{A'} - s_i$
```

At each iteration of lines 3–9, the procedure finds one element of S and “peels” it off S (lines 4–5). In addition, the procedure also updates the corresponding values of the $\textsc{ISize}(S, A)$ and $\textsc{ISum}(S, A)$, for all $A \in F$.

We assume that the sets of F are represented by linked lists of the elements of the set. Thus, in time $O(\sum_{A \in F} |A|)$ we can create for every element $s \in U$ a list of all the sets of F that contain s, and use these lists for the loop in line 6.

Since F is a k-peeler for S, if the procedure happens to choose (in line 4) exactly the A’s from the peeling sequence contained in F, and in exactly the right order, then the process will necessarily find at least $k' = \min(k, |S|)$ elements of S. What happens if other A’s are chosen? or in a different order? The following lemma proves that k' elements are necessarily found, regardless of the choice of the A’s and their ordering.

Lemma 1. *Suppose that F is a k-peeler for S. Then the peeling procedure necessarily finds at least $k' = \min(k, |S|)$ elements of S.*

Proof. Let $(B_1, \ldots, B_{k'})$ be the peeling sequence for S contained in F. Consider an invocation of the peeling procedure. Let $P = (A_1, \ldots, A_t)$ be the sequence of sets chosen so far in the peeling process. We show that if $t < k'$ then P can necessarily be extended. For $i = 1, \ldots, k'$ let $b_i = (S - \cup_{j=1}^{i-1} B_j) \cap B_i$ (i.e. b_i is the element of S “peeled” by B_i) and for $i = 1, \ldots, t$ let $a_i = (S - \cup_{j=1}^{i-1} A_j) \cap A_i$. Let i_0 be smallest index such that $b_{i_0} \notin \{a_1, \ldots, a_t\}$ (such an index necessarily exists since $t < k'$ and all b_i’s are distinct). Then,

$$B_{i_0} \cap (S - \cup_{j=1}^{t} A_j) = b_{i_0}$$

Hence, the peeling process can be continued by choosing B_{i_0}. □

The Peeling Procedure can be extended for reconstructing a collection of sets $\mathcal{S} = \{S_1, \ldots, S_n\}$ with a k-peeler F for $\mathcal{S}$: In step 1 compute $\langle z_{A,1}, \ldots, z_{A,n}\rangle \leftarrow$ ISIZE$(S_1, \ldots, S_n, A)$ and $\langle m_{A,1}, \ldots, m_{A,n}\rangle \leftarrow$ ISUM$(S_1, \ldots, S_n, A)$. Then, perform steps 2–9 for every set S_i separately.

Two factors determine the time complexity of the Peeling Procedure. The one is the size of the peeling collection F. The other is the total number of times lines 6–9 are performed during the peeling process. This number can be bounded as follows. For an element $u \in U$ and a peeling collection F, let $\textsc{Occ}_u(F)$ be the number of sets in F that contain u. Define $\textsc{MaxOcc}(F) = \max_{u\in U} \{\textsc{Occ}_u(F)\}$. The number of times lines 6–9 are performed is bounded by $k \cdot \textsc{MaxOcc}(F)$.

Hence, we seek small peeling collections with a small $\textsc{MaxOcc}(F)$. Clearly, for any set S of size k there is a trivial k-peeler with $|F| = k$ and $\textsc{MaxOcc}(F) = 1$. However, since we are not provided with S explicitly, we must construct the peeling collection without explicit knowledge of S.

The notions of k-separator and k-peeler are similar to k-selective collections [7,13]. A collection F is *k-selective* if for any set S, with $|S| \leq k$, there is a set $A \in F$ such that A peels S. Note that a k-selective collection is *universal*, namely the property is satisfied for any set $S \subseteq U$, while our definitions of k-separator and k-peeler are with respect to a given set S or a collection of sets $\mathcal{S}$. k-selective collections were also been studied in a non-universal setting [6]. In the universal setting, a collection F is k-selective iff it is a universal bounded k-peeler (a universal bounded k-peeler is a collection of sets which is a bounded k-peeler for any set $S \subseteq U$). However, in the non-universal setting the properties of being k-selective and being bounded k-peeler are different: Every bounded k-peeler is also k-selective, but the converse is not true.

Bounded k-peelers are very similar to the universal lists in [15]. The universal lists are defined for the universal setting. Therefore, the construction in [15], which is similar to the construction in Section 2.3, is less efficient than ours.

2.2 Deterministic Constructions

Bounded *k*-Peelers. Consider the problem of constructing a universal bounded k-peeler, or equivalently, constructing a k-selective collection. A probabilistic argument provides that there is a k-selective collection of size $O(k \log m)$ and $\textsc{MaxOcc}(F) = O(\log m)$. Indyk [13] shows how to deterministically construct a k-selective collection with $|F| = O(k \cdot \text{polylog}(m))$. His construction also gives $\textsc{MaxOcc}(F) = O(\text{polylog}(m))$. We note that there is no efficient construction of universal (unbounded) k-peeler. More precisely, the size of a universal k-peeler must be at least m (even for $k = 1$) [9].

***k*-Separators.** Let $\mathcal{S} = \{S_1, \ldots, S_n\}$ be a collection of sets, each of size at least $k \log^c m$, for some constant c to be determined later. We construct a k-separator for $\mathcal{S}$.

The construction uses the algorithm of Alon and Naor [3]. Using the terminology presented here, [3] provides an algorithm to construct a 1-separator for $\mathcal{S}$.

The algorithm uses $O(k \cdot \text{polylog}(nm))$ calls to the procedure ISIZE$(\cdot, \cdot)$, and outputs a collection F of size $O(\text{polylog}(nm))$. We use this algorithm to construct the k-separator.

The following Lemma follows from the explicit disperser constructions of [20].

Lemma 2. *There is a constant c such that for every U, there is an explicit construction of a collection H of subsets of U that satisfies*

1. *For every subset S of U such that $|S| \geq k \log^c m$, there are disjoint sets $A_1, \ldots, A_k \in H$ such that $S \cap A_i$ is not empty for all i, and*
2. *$|H| = O(k \cdot \text{polylog}(m))$.*

To construct the k-separator for $S_1, \ldots, S_n$ we do the following. Let H be the collection as provided by Lemma 2. For every $A \in H$, use the Alon-Naor algorithm to construct a 1-separator for $S_1 \cap A, \ldots, S_n \cap A$. The union of these 1-separators is a k-separator for $S_1, \ldots, S_n$. We obtain:

Theorem 1. *There exists a constant c such that the following holds. Let $\mathcal{S} = \{S_1, \ldots, S_n\}$ be a collection of sets each of size at least $k \log^c m$. It is possible to construct a k-separator collection F for $\mathcal{S}$ such that $|F| = O(k \cdot \text{polylog}(mn))$. Constructing F requires $O(k \cdot \text{polylog}(mn))$ calls to the procedure* ISIZE$(\cdot, \cdot)$.

2.3 Randomized Constructions

Preliminaries. For $p \in [0, 1]$, consider the process of choosing each element of U independently at random with probability p. We call the resulting set a *p-random set.* Let $\alpha = 1/(2e)$.

Lemma 3. *Let A be a $1/t$-random set, and let S be a set with $t/2 \leq |S| \leq t$. Then,* $\Pr[A \textit{ peels } S] \geq \alpha$.

Consider a set S of size t and a collection of sets F. We say that F *peels S down to t'* ($t' < t$) if F is a k-peeler for S with $k = t - t'$.

Corollary 1. *Let F be a collection of r $1/t$-random sets, and let S be a set with $t/2 \leq |S| \leq t$. Then* $\Pr[F \textit{ does not peel } S \textit{ down to } t/2] \leq 2^t(1-\alpha)^r$.

Bounded k-Peelers. Let $\mathcal{S}$ be a collection of n sets. We now show how to construct a collection F such that with probability $\geq 1 - \frac{1}{2}n^{-1}$, F is a bounded k-peeler for $\mathcal{S}$. W.l.o.g. $k \geq 2$. We construct F as a union of collections $F^{(j)}$, $j = 0, \ldots, \lceil \log k \rceil$ (all logarithms in this section are base 2). $F^{(0)} = \{U\}$. For $j > 0$, the collection $F^{(j)}$ consists of r_j $\frac{1}{2^j}$-random sets, with $r_j = \frac{2^j + 3\log(nk)}{\log(1/(1-\alpha))}$ (to simpify the presentation, we omit the ceiling function in the definition of r_j). Consider the following process. Given a set S of size k, use $F^{(\lceil \log k \rceil)}$ to peel S down to a set of size $2^{\lceil \log k \rceil - 1}$. Denote the resulting set by $S^{(\lceil \log k \rceil - 1)}$. Use $F^{(\lceil \log k \rceil - 1)}$ to peel $S^{(\lceil \log k \rceil - 1)}$ down to size $2^{\lceil \log k \rceil - 2}$. Denote the resulting set by $S^{(\lceil \log k \rceil - 2)}$. Continue in this way until the entire set S is peeled.

Theorem 2. *Let $\mathcal{S}$ be a collection of n sets. With probability $\geq 1 - \frac{1}{2}n^{-1}$, F is a bounded k-peeler for $\mathcal{S}$, and* $\textsc{MaxOcc}(F) = O(\log(nm))$*. The size of F is $O(k + \log k \cdot \log n)$.*

Proof. Consider some set $S \in \mathcal{S}$ of size at most k. By Corollary 1, the probability that $F^{(j)}$ fails to peel $S^{(j)}$ down to 2^{j-1} is $\leq 2^{2^j}(1-\alpha)^{r_j} = (nk)^{-3}$. Thus, the probability that F is not a bounded k-peeler for S is $\leq \rceil \log k \lceil \cdot (nk)^{-3} \leq \frac{1}{4}n^{-2}$. Therefore, the probability that F is not a bounded k-peeler for $\mathcal{S}$ is $\leq \frac{1}{4}n^{-1}$.

The size of F is

$$|F| = 1 + \sum_{j=1}^{\lceil \log k \rceil} \frac{2^j + 3\log(nk)}{\log(1/(1-\alpha))} = O(k + \log k \cdot \log(nk)) = O(k + \log k \cdot \log n).$$

Next we bound $\textsc{MaxOcc}(F)$. We have that

$$E(\textsc{Occ}_u(F)) = 1 + \sum_{j=1}^{\lceil \log k \rceil} 2^{-j} \frac{2^j + 3\log(nk)}{\log(1/(1-\alpha))} \leq 1 + 4\lceil \log k \rceil + 11\log(nk) \leq 20\log(nk).$$

Hence, by Hoeffding's inequality, $\Pr[\textsc{Occ}_u(F) > 40\log(nm)] \leq \frac{1}{4}(nm)^{-1}$. There are m different u's, so $\Pr[\textsc{MaxOcc}(F) > 40\log(nm)] \leq \frac{1}{4}n^{-1}$. □

***k*-Separators.** Let $\mathcal{S}$ be a collection of n sets, each of size $\geq 4k$. We show how to construct a collection F such that with probability $\geq 1 - \frac{1}{2}n^{-1}$, F is a k-separator for $\mathcal{S}$. For $j = \lceil \log(4k) \rceil, \ldots, \lceil \log m \rceil$, let $F^{(j)}$ be a set of r_j $\frac{1}{2^j}$-random sets, with $r_j = \frac{16}{\alpha}\log(4n) + \frac{2}{\alpha}kj/(j-1-\log k)$. Let $F = \bigcup_{j=\lceil \log 4k \rceil}^{\lceil \log m \rceil} F^{(j)}$.

Theorem 3. *Let $\mathcal{S}$ be collection of n sets, each of size $\geq 4k$. With probability $\geq 1 - \frac{1}{2}n^{-1}$, F is a k-separator for $\mathcal{S}$. The size of F is $O(k(\log m + \log k \log\log m) + \log n \log m)$.*

Proof. Consider a set $S \in \mathcal{S}$, and let j be an integer such that $2^{j-1} < |S| \leq 2^j$. By Lemma 3, the probability that a fixed set $A \in F^{(j)}$ peels S is $\geq \alpha$. Hence, the expected number sets in $F^{(j)}$ that peel S is $\geq \alpha r$. By Hoeffding's inequality, with probability $\geq 1 - \frac{1}{4}n^{-2}$, there are $\geq \frac{1}{2}\alpha r'$ sets that peel S. For an element $s \in S$, we say that s is *peeled* if there is a set A that peels s. Assuming there are $\geq \frac{1}{2}\alpha r'$ sets that peel S,

$$\Pr[\text{there are} < k \text{ peeled elements}] \leq |S|^{k-1}\left(\frac{k-1}{|S|}\right)^{\frac{\alpha}{2}r} \leq 2^{jk}\left(\frac{k}{2^{j-1}}\right)^{\frac{\alpha}{2}r} \leq \frac{1}{4n^2}.$$

Thus, the probability that F is not a k-separator for S is $\leq \frac{1}{2}n^{-2}$. Hence, the probability that F is not a k-separator of $\mathcal{S}$ is $\leq \frac{1}{2}n^{-1}$. The bound on $|F|$ stated in the theorem is straightforward. □

2.4 Solving the k-Reconstruction Problem

We are now ready to show how we use the separating and peeling collections to solve the reconstruction problems presented in the introduction. Recall that given a collection $\mathcal{S}$ containing n subsets of U, the *k-reconstruction problem* is: For each $S \in \mathcal{S}$, find $\min(k, |S|)$ elements of S. The *bounded k-reconstruction problem* is: Fully reconstruct every set $S \in \mathcal{S}$ of size at most k.

For the bounded k-reconstruction problem, bounded k-peelers fully solve the problem. Thus, we obtain:

Theorem 4. *Suppose that computing* $\textsc{ISize}(S, A)$ *or* $\textsc{ISum}(S, A)$ *for all $S \in \mathcal{S}$ and one set A takes $O(f)$ steps. Then, there exist deterministic and randomized algorithms for the bounded k-reconstruction problem with the following running times:*

- *Deterministic:* $O(k \cdot \text{polylog}(m)(f + n))$ *steps.*
- *Randomized:* $O(f(k + \log k \cdot \log n) + nk \log(mn))$ *steps.*

The randomized algorithm solves the reconstruction problem with probability at least 1/2. We can reapetedly run the algorithm until all sets are reconstructed, and the bound above is the expected number of steps.

For the (unbounded) k-reconstruction problem we do the following. We choose some integer k', and construct both a bounded k'-peeler, which will fully reconstruct sets with at most k' elements, and a k-separator which will reconstruct k elements from sets with at least k' elements. For the deterministic setting we choose $k' = k \log^c m$, and use the results of Section 2.2 for the k'-peeler and that of Theorem 1 for the k-separating collection. For the randomized construction we choose $k' = 4k$, and use Theorem 2 for the k'-peeler and Theorem 3 for the k-separators.

Theorem 5. *Suppose that computing* $\textsc{ISize}(S, A)$ *or* $\textsc{ISum}(S, A)$ *for all $S \in \mathcal{S}$ and one set A takes $O(f)$ steps. Then, there exist deterministic and randomized algorithms for the k-reconstruction problem with the following running times:*

- *Deterministic:* $O(k \cdot \text{polylog}(mn)(f + n))$ *steps.*
- *Randomized:* $O(f(k(\log m + \log k \log \log m) + \log n \log m) + nk \log(mn))$ *steps.*

We note that except for the deterministic result for the (unbounded) k-reconstruction problem, the other algorithms are *non-adaptive*, in sense that the set of A's on which the procedures $\textsc{ISize}(S, A)$ and $\textsc{ISum}(S, A)$ are performed is determined independently of the outcomes of any other such calls.

Due to lack of space, we omit the applications of the results above.

References

1. Abrahamson, K.: Generalized string matching. SIAM J. on Computing 16(6), 1039–1051 (1987)
2. Alon, N., Galil, Z., Margalit, O., Naor, M.: Witnesses for boolean matrix multiplication and for shortest paths. In: Symposium on Foundations of Computer Science (FOCS), pp. 417–426 (1992)

3. Alon, N., Naor, M.: Derandomization, witnesses for boolean matrix multiplication and construction of perfect hash functions. Algorithmica 16, 434–449 (1996)
4. Amir, A., Farach, M.: Efficient 2-dimensional approximate matching of half-rectangular figures. Information and Computation 118(1), 1–11 (1995)
5. Amir, A., Lewenstein, M., Porat, E.: Faster algorithms for string matching with k mismatches. J. of Algorithms 50(2), 257–275 (2004)
6. Brito, C., Gafni, E., Vaya, S.: An information theoretic lower bound for broadcasting in radio networks. In: Proc. 21st Annual Symposium on Theoretical Aspects of Computer Science (STACS), pp. 534–546 (2004)
7. Chlebus, B.S., Gasieniec, L., Gibbons, A., Pelc, A., Rytter, W.: Deterministic broadcasting in ad hoc radio networks. Distributed Computing 15(1), 27–38 (2002)
8. Clementi, A.E.F., Monti, A., Silvestri, R.: Selective families, superimposed codes, and broadcasting on unknown radio networks. In: Proc. 13th Symposium on Discrete Algorithms(SODA), pp. 709–718 (2001)
9. Damaschke, P.: Randomized group testing for mutually obscuring defectives. Information Processing Letters 67, 131–135 (1998)
10. Du, D.Z., Hwang, F.K.: Combinatorial group testing and its applications. World Scientific (2000)
11. Galil, Z., Giancarlo, R.: Improved string matching with k mismatches. SIGACT News. 17(4), 52–54 (1986)
12. Indyk, P.: Interpolation of symmetric functions and a new type of combinatorial design. In: Proc. of Symposium on Theory of Computing (STOC), pp. 736–740 (1999)
13. Indyk, P.: Explicit constructions of selectors and related combinatorial structures, with applications. In: Proc. 13th Symposium on Discrete Algorithms (SODA), pp. 697–704 (2002)
14. Knuth, D.E., Morris, J.H., Pratt, V.R.: Fast pattern matching in strings. SIAM J. Comp. 6, 323–350 (1977)
15. Komlós, J., Greenberg, A.G.: An asymptotically nonadaptive algorithm for conflict resolution in multiple-access channels. IEEE Trans. on Information Theory 31(2), 302–306 (1985)
16. Landau, G.M., Vishkin, U.: Efficient string matching with k mismatches. Theoretical Computer Science 43, 239–249 (1986)
17. Muthukrishnan, S.: Personal communication
18. Muthukrishnan, S.: New results and open problems related to non-standard stringology. In: Galil, Z., Ukkonen, E. (eds.) Combinatorial Pattern Matching. LNCS, vol. 937, pp. 298–317. Springer, Heidelberg (1995)
19. Seidel, R.: On the all-pairs-shortest-path problem in unweighted undirected graphs. J. of Computer and System Sciences 51, 400–403 (1995)
20. Ta-Shma, A., Umans, C., Zuckerman, D.: Loss-less condensers, unbalanced expanders, and extractors. In: Proc. 33th Symposium on the Theory of Computing (STOC), pp. 143–152 (2001)

Cache-Oblivious Index for Approximate String Matching*

Wing-Kai Hon[1], Tak-Wah Lam[2], Rahul Shah[3],
Siu-Lung Tam[2], and Jeffrey Scott Vitter[3]

[1] Department of Computer Science, National Tsing Hua University, Taiwan
wkhon@cs.nthu.edu.tw

[2] Department of Computer Science, The University of Hong Kong, Hong Kong
{twlam,sltam}@cs.hku.hk

[3] Department of Computer Sciences, Purdue University, Indiana, USA
{rahul,jsv}@cs.purdue.edu

Abstract. This paper revisits the problem of indexing a text for approximate string matching. Specifically, given a text T of length n and a positive integer k, we want to construct an index of T such that for any input pattern P, we can find all its k-error matches in T efficiently. This problem is well-studied in the internal-memory setting. Here, we extend some of these recent results to external-memory solutions, which are also cache-oblivious. Our first index occupies $O((n \log^k n)/B)$ disk pages and finds all k-error matches with $O((|P| + occ)/B + \log^k n \log\log_B n)$ I/Os, where B denotes the number of words in a disk page. To the best of our knowledge, this index is the first external-memory data structure that does not require $\Omega(|P| + occ + \mathrm{poly}(\log n))$ I/Os. The second index reduces the space to $O((n \log n)/B)$ disk pages, and the I/O complexity is $O((|P| + occ)/B + \log^{k(k+1)} n \log\log n)$.

1 Introduction

Recent years have witnessed a huge growth in the amount of data produced in various disciplines. Well-known examples include DNA sequences, financial time-series, sensor data, and web files. Due to the limited capacity of main memory, traditional data structures and algorithms that perform optimally in main memory become inadequate in many applications. For example, the suffix tree [19,25] is an efficient data structure for indexing a text T for exact pattern matching; given a pattern P, it takes $O(|P| + occ)$ time to report all occurrences of P in T, where occ denotes the number of occurrences. However, if we apply a suffix tree to index DNA, for example, the human genome which has 3 billion characters, at least 64G bytes of main memory would be needed.

* Research of T.W. Lam is supported by the Hong Kong RGC Grant 7140/06E. Research of R. Shah and J.S. Vitter is supported by NSF Grants IIS–0415097 and CCF–0621457, and ARO Grant DAAD 20–03–1–0321. Part of the work was done while W.K. Hon was at Purdue University.

To solve the problem in dealing with these massive data sets, a natural way is to exploit the external memory as an extension of main memory. In this paradigm of computation, data can be transferred in and out of main memory through an I/O operation. In practice, an I/O operation takes much more time than an operation in main memory. Therefore, it is more important to minimize the number of I/Os.

Aggarwal and Vitter [2] proposed a widely accepted *two-level I/O-model* for analyzing the I/O complexity. In their model, the memory hierarchy consists of a main memory of M words and an external memory. Data reside in external memory initially (as they exceed the capacity of main memory), and computation can be performed only when the required data are present in main memory. With one I/O operation, a disk page with B contiguous words can be read from external memory to main memory, or B words from main memory can be written to a disk page in external memory; the I/O complexity of an algorithm counts only the number of I/O operations involved. To reduce the I/O complexity, an algorithm must be able to exploit the locality of data in external memory. For instance, under this model, sorting a set of n numbers can be done in $O\left((\frac{n}{B}\log\frac{n}{B})/\log(\frac{M}{B})\right)$ I/Os, and this bound is proven to be optimal. (See [24] for more algorithms and data structures in the two-level I/O model.)

Later, Frigo et al. [15] introduced the notion of *cache-obliviousness*, in which we do not have advance knowledge of M or B in designing data structures and algorithms for external memory; instead, we require the data structures and algorithms to work for any given M and B. Furthermore, we would like to match the I/O complexity when M and B are known in advance. Among others, cache-obliviousness implies that the algorithms and data structures will readily work well under different machines, without the need of fine tuning the algorithm (or recompilation) or rebuilding the data structures. Many optimal cache-oblivious algorithms and data structures are proposed over the recent years, including algorithms for sorting [20] and matrix transposition [20], and data structures like priority queues [8] and B-trees [7].

For string matching, the recent data structure proposed by Brodal and Fagerberg [9] can index a text T in $O(n/B)$ disk pages[1] and find all occurrences of a given pattern P in T in $O((|P| + occ)/B + \log_B n)$ I/Os. This index works in a cache-oblivious manner, improving the String-B tree, which is an earlier work by Ferragina and Grossi [14] that achieves the same space and I/O bounds but requires the knowledge of B to operate.[2] In this paper, we consider the *approximate string matching* problem defined as follows:

> Given a text T of length n and a fixed positive integer k, construct an index on T such that for any input pattern P, we can find all *k-error matches* of P in T, where a k-error match of P is a string that can be transformed to P using at most k character insertions, deletions, or replacements.

[1] Under the cache-oblivious model, the index occupies $O(n)$ contiguous words in the external memory. The value of B is arbitrary, which is considered only in the analysis.

[2] Recently Bender et al. [6] have devised the cache-oblivious string B-tree, which is for other pattern matching queries such as prefix matching and range searching.

The above problem is well-studied in the internal-memory setting [21,12,3,10,17,11]. Recently, Cole et al. [13] proposed an index that occupies $O(n \log^k n)$ words of space, and can find all k-error matches of a pattern P in $O(|P| + \log^k n \log\log n + occ)$ time. This is the first solution with time complexity linear to $|P|$; in contrast, the time complexity of other existing solutions depends on $|P|^k$. Chan et al. [11] later gave another index that requires only $O(n)$ space, and the time complexity increases to $O(|P| + \log^{k(k+1)} n \log\log n + occ)$. In this paper, we extend these two results to the external-memory setting. In addition, our solution is cache-oblivious.

The main difficulty in extending Cole et al.'s index to the external-memory setting lies in how to answer the longest common prefix (LCP) query for an arbitrary suffix of a pattern P using a few I/Os. More specifically, given a suffix P_i, we want to find a substring of T that is the longest prefix of P_i. In the internal memory setting, we can compute all possible LCP values in advance in $O(|P|)$ time (there are $|P|$ such values) by exploiting the suffix links in the suffix tree of T. Then each LCP query can be answered in $O(1)$ time. In the external memory setting, a naive implementation would require $\Omega(\min\{|P|^2/B, |P|\})$ I/Os to compute all LCP values. To circumvent this bottleneck, we target to compute only some "useful" LCP values in advance (using $O(|P|/B + k \log_B n)$ I/Os), so that each subsequent LCP query can still be answered efficiently (in $O(\log\log_B n)$ I/Os). Yet this target is very difficult to achieve for general patterns. Instead, we take advantage of a new notion called *k-partitionable* and show that if P is k-partitionable, we can achieve the above target; otherwise, T contains no k-error match of P. To support this idea, we devise an I/O-efficient *screening test* that checks whether P is k-partitionable; if P is k-partitionable, the screening test would also compute some useful *LCP* values as a by-product, which can then be utilized to answer the LCP query for an arbitrary P_i in $O(\log\log_B n)$ I/Os.

Together with other cache oblivious data structures (for supporting LCA, Y-Fast Trie and WLA), we are able to construct an index to find all k-error matches using $O((|P| + occ)/B + \log^k n \log\log_B n)$ I/Os. The space of the index is $O((n \log^k n)/B)$ disk pages. To the best of our knowledge, this is the first external-memory data structure that does not require $\Omega(|P| + occ + \text{poly}(\log n))$ I/Os. Note that both Cole et al.'s index and our index can work even if the alphabet size is unbounded.

Recall that the internal-memory index by Chan et al. [11] occupies only $O(n)$ space. The reduction of space demands a more involved searching algorithm. In particular, they need the data structure of [10] to support a special query called Tree-Cross-Product. Again, we can 'externalize' this index. Here, the difficulties come in two parts: (i) computing the LCP values, and (ii) answering the Tree-Cross-Product queries. For (i), we will use the same approach as we externalize Cole et al.'s index. For (ii), there is no external memory counter-part for the data structure of [10]; instead, we reduce the Tree-Cross-Product query to a two-dimensional orthogonal range search query, the latter can be answered efficiently using an external-memory index based on the work in [1]. In this way, for any fixed $k \geq 2$, we can construct an index using $O((n \log n)/B)$ disk pages, which can find all k-error matches of P in $O((|P|+occ)/B+\log^{k(k+1)} n \log\log n)$ I/Os. Following [11], our second result assumes alphabet size is constant.

In Section 2, we give a survey of a few interesting queries that have efficient cache-oblivious solutions. In particular, the index for WLA (weighted level ancestor) is not known in the literature. Section 3 reviews Cole et al.'s internal memory index for k-error matching and discusses how to turn it into an external memory index. Section 4 defines the k-partitionable property, describes the screening test, and show how to compute LCP queries efficiently. Finally, Section 5 states our result obtained by externalizing the index of Chan et al.

2 Preliminaries

2.1 Suffix Tree, Suffix Array, and Inverse Suffix Array

Given a text $T[1..n]$, the substring $T[i..n]$ for any $i \in [1,n]$ is called a suffix of T. We assume that characters in T are drawn from an ordered alphabet which is of constant size, and $T[n] = \$$ is a distinct character that does not appear elsewhere in T. The *suffix tree* of T [19,25] is a compact trie storing all suffixes of T. Each edge corresponds to a substring of T, which is called the *edge label*. For any node u, the concatenation of edge labels along the path from root to u is called the *path label* of u. There are n leaves in the suffix tree, with each leaf corresponding to a suffix of T. Each leaf stores the starting position of its corresponding suffix, which is called the *leaf label*. The children of an internal node are ordered by the lexicographical order of their edge labels.

The *suffix array* of T [18], denoted by SA, is an array of integers such that $SA[i]$ stores the starting position of the ith smallest suffix in the lexicographical order. It is worth-mentioning that SA can also be obtained by traversing the suffix tree in a left-to-right order and recording the leaf labels. Furthermore, the descendant leaves of each internal node u in the suffix tree correspond to a contiguous range in the suffix array, and we call this the SA range of u.

The *inverse suffix array*, denoted by SA^{-1}, is defined such that $SA^{-1}[i] = j$ if and only if $i = SA[j]$. When stored in the external memory, the space of both arrays take $O(n/B)$ disk pages, and each entry can be reported in one I/O.

Suppose that we are given a pattern P, which appears at position i of T. That is, $T[i..i+|P|-1] = P$. Then, we observe that P is a prefix of the suffix $T[i..n]$. Furthermore, each other occurrence of P in T corresponds to a distinct suffix of T sharing P as a prefix. Based on this observation, the following lemma from [18] shows a nice property about the suffix array.

Lemma 1. *Suppose P is a pattern appearing in T. Then there exists $i \le j$ such that $SA[i]$, $SA[i+1], \ldots, SA[j]$ are the starting positions of all suffixes sharing P as a prefix. In other words, $SA[i..j]$ lists all occurrences of P in T.* □

2.2 Cache-Oblivious String Dictionaries

Recently, Brodal and Fagerberg proposed an external-memory index for a text T of length n that supports efficient pattern matching query [9]. Their index takes $O(n/B)$ disk pages of storage; also, it does not require the knowledge of M or B to operate and is therefore cache-oblivious. For the pattern matching query, given any input pattern P, we can find all occurrences of P in T using $O((|P| + occ)/B + \log_B n)$ I/Os.

In this paper, we are interested in answering a slightly more general query. Given a pattern P, let ℓ be the length of the longest prefix of P that appears in T. We want to find all suffixes of T that has $P[1..\ell]$ as a prefix (that is, all suffixes of T whose common prefix with P is the longest among the others). We denote Q to be the set of starting positions of all such suffixes. Note that Q occupies a contiguous region in SA, say $SA[i..j]$. Now we define the *LCP query* of P with respect to T, denoted by $LCP(P,T)$, which reports (i) the *SA range*, $[i,j]$, that corresponds to the SA region occupied by Q, and (ii) the *LCP length*, ℓ.

With very minor adaptation, the index in [9] can readily be used to support efficient LCP query, as stated in the following lemma.

Lemma 2. *We can construct a cache-oblivious index for a text T of length n, such that given a pattern P, we can find $LCP(P,T)$, its SA range, and its length in $O(|P|/B + \log_B n)$ I/Os. The space of the index is $O(n/B)$ disk pages.* □

2.3 LCA Index on Rooted Tree

For any two nodes u and v in a rooted tree, a *common ancestor* of u and v is a node that appears in both the path from u to the root and the path from v to the root; among all common ancestors of u and v, the one that is closest to u and v is called the *lowest common ancestor* of u and v, denoted by $LCA(u,v)$. The lemma below states the performance of an external-memory index for LCA queries, which follows directly from the results in [16,5].

Lemma 3. *Given a rooted tree with n nodes, we can construct a cache-oblivious index of size $O(n/B)$ disk pages such that for any nodes u and v in the tree, $LCA(u,v)$ can be reported in $O(1)$ I/Os.* □

2.4 Cache-Oblivious Y-Fast Trie

Given a set X of x integers, the predecessor of r in X, denoted by $Pred(r,X)$, is the largest integer in X which is smaller than r. If the integers in X are chosen from $[1,n]$, the *Y-fast trie* on X [26] can find the predecessor of any input r in $O(\log\log n)$ time under the word RAM model; [3] the space occupancy is $O(x)$ words. In the external-memory setting, we can store the Y-fast trie easily using the van Emde Boas layout [7,22,23,20], giving the following lemma.

Lemma 4. *Given a set X of x integers chosen from $[1,n]$, we can construct a cache-oblivious Y-fast trie such that $Pred(r,X)$ for any integer r can be answered using $O(\log\log_B n)$ I/Os. The space of the Y-fast trie is $O(x/B)$ disk pages.* □

[3] A word RAM supports standard arithmetic and bitwise boolean operations on word-sized operands in $O(1)$ time.

2.5 Cache-Oblivious WLA Index

Let R be an edge-weighted rooted tree with n nodes, where the weight on each edge is an integer in $[1, W]$. We want to construct an index on R so that given any node u and any integer w, we can find the unique node v (if exists) with the following properties: (1) v is an ancestor u, (2) sum of weights on the edge from the root of R to v is at least w, and (3) no ancestor of v satisfies the above two properties. We call v the weighted level ancestor of u at depth w, and denote it by $WLA(u, w)$.

Assume that $\log W = O(\log n)$. In the internal-memory setting, we can construct an index that requires $O(n)$ words of space and finds $WLA(u, w)$ in $O(\log \log n)$ time [4]. In the following, we describe the result of a new WLA index that works cache-obliviously, which may be of independent interest. This result is based on a recursive structure with careful space management, whose proof is deferred to the full paper.

Lemma 5. *We can construct a cache-oblivious index on R such that for any node u and any integer w, $WLA(u, w)$ can be reported in $O(\log \log_B n)$ I/Os. The total space of the index is $O(n/B)$ disk pages.* □

2.6 Cache-Oblivious Index for Join Operation

Let T be a text of length n. For any two strings Q_1 and Q_2, suppose that $LCP(Q_1, T)$ and $LCP(Q_2, T)$ are known. The *join* operation for Q_1 and Q_2 is to compute $LCP(Q_1Q_2, T)$, where Q_1Q_2 is the concatenation of Q_1 and Q_2.

Cole et al. (Section 5 of [13]) had developed an index of $O(n \log n)$ words that performs the *join* operation in $O(\log \log n)$ time in the internal-memory setting. Their index assumes the internal-memory results of LCA index, Y-fast trie, and WLA index. In the following lemma, we give an index that supports efficient *join* operations in the cache-oblivious setting.

Lemma 6. *We can construct a cache-oblivious index on T of $O((n \log n)/B)$ disk pages and supports the join operation in $O(\log \log_B n)$ I/Os.*

Proof. Using Lemmas 3, 4, and 5, we can directly extend Cole et al.'s index into a cache-oblivious index. □

3 A Review of Cole et al.'s k-Error Matching

In this section, we review the internal-memory index for k-error matching proposed by Cole et al. [13], and explain the challenge in turning it into a cache-oblivious index.

To index a text T of length n, Cole et al.'s index includes two data structures: (1) the suffix tree of T that occupies $O(n)$ words, and (2) a special tree structure, called k-error tree, that occupies a total of $O(n \log^k n)$ words in space. The k-error tree connects a number of $(k-1)$-error trees, each of which in turn connects to a number of $(k-2)$-error trees, and so on. The bottom of this recursive structure are 0-error trees.

Given a pattern P, Cole et al.'s matching algorithm considers different ways of making k edit operations on P in order to obtain an exact match in T. Intuitively, the matching algorithm first considers all possible locations of the leftmost error on P to obtain a match; then for each location i that has an error, we can focus on searching the remaining suffix, $P[i+1..|P|]$, for subsequent errors. The searches are efficiently supported by the recursive tree structure. More precisely, at the top level, the k-error tree will immediately identify all matches of P in T with no errors; in addition, for those matches of P with at least one error, the k-error tree classifies the different ways that the leftmost edit operation on P into $O(\log n)$ groups, and then each group creates a search in a dedicated $(k-1)$-error tree. Subsequently, each $(k-1)$-error tree being searched will immediately identify all matches of P with one error, while for those matches of P with at least two errors, the $(k-1)$-error tree further classifies the different ways that the second-leftmost edit operation on P into $O(\log n)$ groups, and then each group creates a search in a dedicated $(k-2)$-error tree. The process continues until we are searching a 0-error tree, in which all matches of P with exactly k errors are reported.

The classification step in each k'-error tree is cleverly done to avoid repeatedly accessing characters in P. It does so by means of a constant number of LCA, LCP, Pred, and WLA queries; then, we are able to create enough information (such as the starting position of the remaining suffix of P to be matched) to guide the subsequent $O(\log n)$ searches in the $(k'-1)$-error trees. Reporting matches in each error tree can also be done by a constant number of LCA, LCP, Pred, and WLA queries. In total, it can be done by $O(\log^k n)$ of these queries. See Figure 1 for the framework of Cole et al.'s algorithm.

Each LCA, Pred, or WLA query can be answered in $O(\log\log n)$ time. For the LCP queries, they are all in the form of $LCP(P_i, T)$, where P_i denotes the suffix $P[i..|P|]$. Instead of computing these values on demand, Cole et al. computes all these LCP values at the beginning of the algorithm. There are $|P|$ such LCP values, which can be computed in $O(|P|)$ time by exploiting the *suffix links* of the suffix tree of T (the algorithm is essentially McCreight's suffix tree construction algorithm [19]). Consequently, each LCP query is returned in $O(1)$ time when needed. Then, Cole et al.'s index supports k-error matching in $O(|P| + \log^k n \log\log n + occ)$ time, where occ denotes the number of occurrences.

3.1 Externalization of Cole et al.'s Index

One may now think of turning Cole et al.'s index directly into a cache-oblivious index, based on the existing techniques. While each LCA, Pred, or WLA query can be answered in $O(\log\log_B n)$ I/Os by storing suitable data structures (See Lemmas 3, 4, and 5), the bottleneck lies in answering the $O(\log^k n)$ LCP queries. In the external memory setting, though we can replace the suffix tree with Brodal and Fagerberg's cache-oblivious string dictionary (Lemma 2), we can no longer exploit the suffix links as efficiently as before. That means, if we compute $LCP(P_i, T)$ for all i in advance, we will need $\Omega(|P|^2/B)$ I/Os. Alternatively, if

```
Algorithm APPROXIMATE_MATCH($P$)
Input: A pattern $P$
Output: All occurrences of $k$-error match of $P$ in $T$
1.  $R \leftarrow k$-error tree of $T$;
2.  SEARCH_ERROR_TREE($P, R,$ nil);
3.  return;

Subroutine SEARCH_ERROR_TREE($P, R, I$)
Input: A pattern $P$, an error tree $R$, information $I$ to guide the search
    of $P$ in $R$
1.  if $R$ is a 0-error tree
2.     then Output all matches of $P$ with $k$ errors based on $R$ and $I$;
3.          return;
4.     else (* $R$ is a $k'$-error tree for some $k' > 0$ *)
5.          Output all matches of $P$ with $k - k'$ errors based on $R$
            and $I$;
6.          Classify potential error positions into $O(\log n)$ groups
            based on $P$, $R$, and $I$;
7.          for each group $i$
8.              Identify the $(k' - 1)$-error tree $R_i$ corresponding
                to group $i$;
9.              Compute information $I_i$ to guide the search of $P$
                in $R_i$;
10.             SEARCH_ERROR_TREE($P, R_i, I_i$);
11. return;
```

Fig. 1. Cole et al.'s algorithm for k-error matching

we compute each LCP query on demand without doing anything at the beginning, we will need a total of $\Omega((\log^k n)|P|/B)$ I/Os to answer all LCP queries during the search process. In summary, a direct translation of Cole et al.'s index into an external memory index will need $\Omega((\min\{|P|^2, |P|\log^k n\} + occ)/B + \log^k n \log\log_B n)$ I/Os for k-error matching.

In the next section, we propose another approach, where we compute *some* useful LCP values using $O(|P|/B + k\log_B n)$ I/Os at the beginning, and each subsequent query of $LCP(P_i, T)$ can be answered efficiently in $O(\log\log_B n)$ I/Os (see Lemma 9 in Section 4). This result leads us to the following theorem.

Theorem 1. *For a fixed integer k, we can construct a cache-oblivious index on T of size $O((n\log^k n)/B)$ disk pages such that, given any pattern P, the k-error matches of P can be found in $O((|P| + occ)/B + \log^k n \log\log_B n)$ I/Os.* □

4 Cache-Oblivious *k*-Error Matching

Let P be a pattern, and let $P_i = P[i..|P|]$ be a suffix of P. In this section, our target is to perform some preprocessing on P in $O(|P|/B + k\log_B n)$ I/Os to obtain some useful $LCP(P_i, T)$ values, such that subsequent query of $LCP(P_j, T)$ for any j can be answered in $O(\log \log_B n)$ I/Os.

We observe that for a general pattern P, the above target may be difficult to achieve. Instead, we take advantage by concerning only those P that potentially has k-error matches. We formulate a notion called *k-partitionable* and show that

– if P is k-partitionable, we can achieve the above target;
– if P is not k-partitionable, there must be no k-error match of P in T.

In Section 4.1, we first define the k-partitionable property, and describe an efficient *screening test* that checks whether P is k-partitionable; in case P is k-partitionable, the screening test would have computed $LCP(P_i, T)$ values for some i as a by-product. In Section 4.2, we show how to utilize these precomputed LCP values to answer $LCP(P_j, T)$ for any j in $O(\log \log_B n)$ I/Os.

In the following, we assume that we have maintained the suffix array and inverse suffix array of T. Each entry of these two arrays will be accessed one at a time, at the cost of one I/O per access.

4.1 k-Partitionable and Screening Test

Consider the following partitioning process on P. In Step 1, we delete the first ℓ characters of P where ℓ is the LCP length reported by $LCP(P, T)$. While P is not empty, Step 2 removes further the first character from P. Then, we repeatedly apply Step 1 and Step 2 until P is empty. In this way, P is partitioned into $\pi_1, c_1, \pi_2, c_2, \ldots, \pi_d, c_d, \pi_{d+1}$ such that π_i is a string that appears in T, and c_i is called a *cut-character* such that $\pi_i c_i$ is a string that does not appear in T. (Note that π_{d+1} is an empty string if P becomes empty after some Step 2.) Note that this partitioning is unique, and we call this the *greedy partitioning* of P.

Definition 1. *P is called k-partitionable if the greedy partitioning of P consists of at most k cut-characters.* □

The following lemma states that k-partitionable property is a necessary condition for the existence of k-error match.

Lemma 7. *If P has a k-error match, P is k-partitionable.* □

The *screening test* on P performs the greedy partitioning of P to check if P is k-partitionable. If not, we can immediately conclude that P does not have a k-error match in T. One way to perform the screening test is to apply Lemma 2 repeatedly, so that we discover π_1 and c_1 in $O(|P|/B + \log_B n)$ I/Os, then discover π_2 and c_2 in $O((|P| - |\pi_1| - 1)/B + \log_B n)$ I/Os, and so on. However, in the worst case, this procedure will require $O(k(|P|/B + \log_B n))$ I/Os. In the following lemma, we make a better use of Lemma 2 with the standard doubling technique and show how to use $O(|P|/B + k \log_B n)$ I/Os to determine whether P passes the screening test or not.

Lemma 8. *The screening test on P can be done cache-obliviously in $O(|P|/B + k \log_B n)$ I/Os.*

Proof. Let $r = \lceil |P|/k \rceil$. In Round 1, we perform the following steps.

- We apply Lemma 2 on $P[1..r]$ to see if it appears in T. If so, we double the value of r and check if $P[1..r]$ appears in T. The doubling continues until we obtain some $P[1..r]$ which does not appear in T, and in which case, we have also obtained π_1 and $LCP(\pi_1, T)$.
- Next, we remove the prefix π_1 from P. The first character of P will then become the cut-character c_1, and we apply Lemma 2 to get $LCP(c_1, T)$. After that, remove c_1 from P.

In each subsequent round, say Round i, we reset the value of r to be $\lceil |P|/k \rceil$, and apply the same steps to find π_i and c_i (as well as $LCP(\pi_i, T)$ and $LCP(c_i, T)$). The algorithm stops when P is empty, or when we get c_{k+1}.

It is easy to check that the above process correctly outputs the greedy partitioning of P (or, up to the cut-character c_{k+1} if P does not become empty) and thus checks if P is k-partitionable. The number of I/Os of the above process can be bounded as follows. Let a_i denote the number of times we apply Lemma 2 in Round i, and b_i denote the total number of characters compared in Round i. Then, the total I/O cost is at most $O((\sum_i b_i)/B + (\sum_i a_i)\log_B n)$ by Lemma 2. The term $\sum_i b_i$ is bounded by $O(|P| + k)$ because Round i compares $O(|\pi_i| + \lceil |P|/k \rceil)$ characters, and there are only $O(k)$ rounds. For a_i, it is bounded by $O(\log(k|\pi_i|/|P|) + 1)$, so that by Jensen's inequality, the term $\sum_i a_i$ is bounded by $O(k)$. □

4.2 Computing LCP for *k*-Partitionable Pattern

In case P is k-partitionable, the screening test in Section 4.1 would also have computed the answers for $LCP(\pi_i, T)$ and $LCP(c_i, T)$. To answer $LCP(P_j, T)$, we will make use of the *join* operation (Lemma 6) as follows. Firstly, we determine which π_i or c_i covers the jth position of P.[4] Then, there are two cases:

- **Case 1:** If the jth position of P is covered by π_i, we notice that the LCP length of $LCP(P_j, T)$ cannot be too long since $\pi_{i+1}c_{i+1}$ does not appear in T. Denote $\pi_i(j)$ to be the suffix of π_i that overlaps with P_j. Indeed, we have:

  Fact 1. $LCP(P_j, T) = LCP(\pi_i(j)c_i\pi_{i+1}, T)$.

  This shows that $LCP(P_j, T)$ can be found by the *join* operations in Lemma 6 repeatedly on $\pi_i(j)$, c_i and π_{i+1}. The SA range of $\pi_i(j)$ can be found easily using SA, SA^{-1} and WLA as follows. Let $[p, q]$ be the SA range of π_i. The pth smallest suffix is the string $T[SA[p]..n]$, which has π_i as a prefix. We can compute $p' = SA^{-1}[SA[p] + j]$, such that the p'th smallest suffix has $\pi_i(j)$ as a prefix. Using the WLA index, we can locate the node (or edge) in the suffix tree of T corresponding to $\pi_i(j)$. Then, we can retrieve the required SA range from this node. The LCP query on P_j can be answered in $O(\log\log_B n)$ I/Os.

[4] This is in fact a predecessor query and can be answered in $O(\log\log_B n)$ I/Os by maintaining a Y-fast trie for the starting positions of each π_i and c_i.

– **Case 2:** If c_i is the jth character of P, the LCP query on P_j can be answered by the *join* operation on c_i and π_{i+1} in $O(\log\log_B n)$ I/Os, using similar arguments as in Case 1.

Thus, we can conclude the section with the following lemma.

Lemma 9. *Let T be a text of length n, and k be a fixed integer. Given any pattern P, we can perform a screening test in $O(|P|/B + k\log_B n)$ I/Os such that if P does not pass the test, it implies there is no k-error match of P in T. In case P passes the test, $LCP(P[j..|P|], T)$ for any j can be returned in $O(\log\log_B n)$ I/Os.* □

5 $O(n\log n)$ Space Cache-Oblivious Index

To obtain an $O(n\log n)$-space index, we externalize Chan et al.'s internal-memory index, so that for patterns longer than $\log^{k+1} n$, they can be searched in $O((|P| + occ)/B + \log^{k(k+1)} n\log\log n)$ I/Os. Roughly speaking, this index consists of a 'simplified' version of the index in Theorem 1, together with the range-searching index by Arge et al. [1] to answer the Tree-Cross-Product queries. To handle short patterns, we find that the internal-memory index of Lam et al. [17] can be used directly without modification, so that short patterns can be searched in $O(\log^{k(k+1)} n\log\log n + occ/B)$ I/Os.

Due to space limitation, we only state our result obtained by the above schemes. Details are deferred to the full paper.

Theorem 2. *For a constant $k \geq 2$, we can construct a cache-oblivious index on T of size $O(n\log n/B)$ pages such that on given any pattern P, the k-error matches of P can be found in $O((|P| + occ)/B + \log^{k(k+1)} n\log\log n)$ I/Os. For $k = 1$, searching takes $O((|P| + occ)/B + \log^3 n\log_B n)$ I/Os.* □

Acknowledgement

The authors would like to thank Gerth Stølting Brodal and Rolf Fagerberg for discussion on their results in [9], and the anonymous referees for their comments.

References

1. Arge, L., Brodal, G.S., Fagerberg, R., Laustsen, M.: Cache-Oblivious Planar Orthogonal Range Searching and Counting. In: Proc. of Annual Symposium on Computational Geometry, pp. 160–169 (2005)
2. Aggarwal, A., Vitter, J.S.: The Input/Output Complexity of Sorting and Related Problems. Communications of the ACM 31(9), 1116–1127 (1988)
3. Amir, A., Keselman, D., Landau, G.M., Lewenstein, M., Lewenstein, N., Rodeh, M.: Indexing and Dictionary Matching with One Error. In: Proc. of Workshop on Algorithms and Data Structures, pp. 181–192 (1999)

4. Amir, A., Landau, G.M., Lewenstein, M., Sokol, D.: Dynamic Text and Static Pattern Matching. In: Proc. of Workshop on Algorithms and Data Structures, pp. 340–352 (2003)
5. Bender, M.A., Farach-Colton, M.: The LCA Problem Revisited. In: Proc. of Latin American Symposium on Theoretical Informatics, pp. 88–94 (2000)
6. Bender, M.A., Farach-Colton, M., Kuszmaul, B.C.: Cache-Oblivious String B-trees. In: Proc. of Principles of Database Systems, pp. 233–242 (2006)
7. Bender, M.A., Demaine, E.D., Farach-Colton, M.: Cache-Oblivious B-trees. In: Proc. of Foundations of Computer Science, pp. 399–409 (2000)
8. Brodal, G.S., Fagerberg, R.: Funnel Heap—A Cache Oblivious Priority Queue. In: Proc. of Int. Symposium on Algorithms and Computation, pp. 219–228 (2002)
9. Brodal, G.S., Fagerberg, R.: Cache-Oblivious String Dictionaries. In: Proc. of Symposium on Discrete Algorithms, pp. 581–590 (2006)
10. Buchsbaum, A.L., Goodrich, M.T., Westbrook, J.: Range Searching Over Tree Cross Products. In: Proc. of European Symposium on Algorithms, pp. 120–131 (2000)
11. Chan, H.L., Lam, T.W., Sung, W.K., Tam, S.L., Wong, S.S.: A Linear Size Index for Approximate Pattern Matching. In: Proc. of Symposium on Combinatorial Pattern Matching, pp. 49–59 (2006)
12. Cobbs, A.: Fast Approximate Matching using Suffix Trees. In: Proc. of Symposium on Combinatorial Pattern Matching, pp. 41–54 (1995)
13. Cole, R., Gottlieb, L.A., Lewenstein, M.: Dictionary Matching and Indexing with Errors and Don't Cares. In: Proc. of Symposium on Theory of Computing, pp. 91–100 (2004)
14. Ferragina, P., Grossi, R.: The String B-tree: A New Data Structure for String Searching in External Memory and Its Application. JACM 46(2), 236–280 (1999)
15. Frigo, M., Leiserson, C.E., Prokop, H., Ramachandran, S.: Cache-Oblivious Algorithms. In: Proc. of Foundations of Computer Science, pp. 285–298 (1999)
16. Harel, D., Tarjan, R.: Fast Algorithms for Finding Nearest Common Ancestor. SIAM Journal on Computing 13, 338–355 (1984)
17. Lam, T.W., Sung, W.K., Wong, S.S.: Improved Approximate String Matching Using Compressed Suffix Data Structures. In: Proc. of International Symposium on Algorithms and Computation, pp. 339–348 (2005)
18. Manber, U., Myers, G.: Suffix Arrays: A New Method for On-Line String Searches. SIAM Journal on Computing 22(5), 935–948 (1993)
19. McCreight, E.M.: A Space-economical Suffix Tree Construction Algorithm. JACM 23(2), 262–272 (1976)
20. Prokop, H.: Cache-Oblivious Algorithms, Master's thesis, MIT (1999)
21. Ukkonen, E.: Approximate Matching Over Suffix Trees. In: Proc. of Symposium on Combinatorial Pattern Matching, pp. 228–242 (1993)
22. van Emde Boas, P.: Preserving Order in a Forest in Less Than Logarithmic Time and Linear Space. Information Processing Letters 6(3), 80–82 (1977)
23. van Emde Boas, P., Kaas, R., Zijlstra, E.: Design and Implementation of an Efficient Priority Queue. Mathematical Systems Theory 10, 99–127 (1977)
24. Vitter, J.S.: External Memory Algorithms and Data Structures: Dealing with Massive Data, 2007. Revision to the article that appeared in ACM Computing Surveys 33(2), 209–271 (2001)
25. Weiner, P.: Linear Pattern Matching Algorithms. In: Proc. of Symposium on Switching and Automata Theory, pp. 1–11 (1973)
26. Willard, D.E.: Log-Logarithmic Worst-Case Range Queries are Possible in Space$\Theta(N)$. Information Processing Letters 17(2), 81–84 (1983)

Improved Approximate String Matching and Regular Expression Matching on Ziv-Lempel Compressed Texts

Philip Bille[1], Rolf Fagerberg[2], and Inge Li Gørtz[3,⋆]

[1] IT University of Copenhagen. Rued Langgaards Vej 7, 2300 Copenhagen S, Denmark
beetle@itu.dk

[2] University of Southern Denmark. Campusvej 55, 5230 Odense M, Denmark
rolf@imada.sdu.dk

[3] Technical University of Denmark. Informatics and Mathematical Modelling, Building 322, 2800 Kgs. Lyngby, Denmark
ilg@imm.dtu.dk

Abstract. We study the approximate string matching and regular expression matching problem for the case when the text to be searched is compressed with the Ziv-Lempel adaptive dictionary compression schemes. We present a time-space trade-off that leads to algorithms improving the previously known complexities for both problems. In particular, we significantly improve the space bounds. In practical applications the space is likely to be a bottleneck and therefore this is of crucial importance.

1 Introduction

Modern text databases, e.g., for biological and World Wide Web data, are huge. To save time and space the data must be kept in compressed form and allow efficient searching. Motivated by this Amir and Benson [1,2] initiated the study of *compressed pattern matching problems*, that is, given a text string Q in compressed form Z and a specified (uncompressed) pattern P, find all occurrences of P in Q without decompressing Z. The goal is to search more efficiently than the naïve approach of decompressing Z into Q and then searching for P in Q. Various compressed pattern matching algorithms have been proposed depending on the type of pattern and compression method, see e.g., [2,9,11,10,17,13]. For instance, given a string Q of length u compressed with the Ziv-Lempel-Welch scheme [22] into a string of length n, Amir et al. [3] gave an algorithm for finding all exact occurrences of a pattern string of length m in $O(n + m^2)$ time and space.

In this paper we study the classical approximate string matching and regular expression matching problems on compressed texts. As in previous work on these problems [10,17] we focus on the popular ZL78 and ZLW adaptive dictionary compression schemes [24,22]. We present a new technique that gives

⋆ Supported by a grant from the Carlsberg Foundation.

a general time-space trade-off. The resulting algorithms improve all previously known complexities for both problems. In particular, we significantly improve the space bounds. When searching large text databases space is likely to be a bottleneck and therefore this is of crucial importance.

In the following sections we give the details of our contribution.

1.1 Approximate String Matching

Given strings P and Q and an *error threshold* k, the *approximate string matching problem* is to find all ending positions of substrings of Q whose *edit distance* to P is at most k. The edit distance between two strings is the minimum number of insertions, deletions, and substitutions needed to convert one string to the other. The classical dynamic programming solution due to Sellers [20] solves the problem in $O(um)$ time and $O(m)$ space, where u and m are the length of Q and P, respectively. Several improvements of this result are known, see e.g., the survey by Navarro [16]. For this paper we are particularly interested in the fast solution for small values of k, namely, the $O(uk)$ time algorithm by Landau and Vishkin [12] and the more recent $O(uk^4/m + u)$ time algorithm due to Cole and Hariharan [7] (we assume w.l.o.g. that $k < m$). Both of these can be implemented in $O(m)$ space.

Recently, Kärkkäinen et al. [10] studied the problem for text compressed with the ZL78/ZLW compression schemes. If n is the length of the compressed text their algorithm achieves $O(nmk + occ)$ time and $O(nmk)$ space, where occ is the number of occurrences of the pattern. Currently, this is the only non-trivial worst-case bound for the problem. For special cases and restricted versions of the problem other algorithms have been proposed [14,19]. An experimental study of the problem and an optimized practical implementation can be found in [18].

In this paper, we show that the problem is closely connected to the uncompressed problem and we achieve a simple time-space trade-off. More precisely, let $t(m, u, k)$ and $s(m, u, k)$ denote the time and space, respectively, needed by any algorithm to solve the (uncompressed) approximate string matching problem with error threshold k for pattern and text of length m and u, respectively. We show the following result.

Theorem 1. *Let Q be a string compressed using* ZL78 *into a string Z of length n and let P be a pattern of length m. Given Z, P, and a parameter $\tau \geq 1$, we can find all approximate occurrences of P in Q with at most k errors in $O(n(\tau + m + t(m, 2m + 2k, k)) + occ)$ expected time and $O(n/\tau + m + s(m, 2m + 2k, k) + occ)$ space.*

The expectation is due to hashing and can be removed at an additional $O(n)$ space cost. In this case the bound also hold for ZLW compressed strings. We assume that the algorithm for the uncompressed problem produces the matches in sorted order (as is the case for all algorithms that we are aware of). Otherwise, additional time for sorting must be included in the bounds. To compare Theorem 1 with the result of Karkkainen et al. [10] plug in the Landau-Vishkin algorithm and set $\tau = mk$. This gives an algorithm using $O(nmk + occ)$ time and

$O(n/mk+m+occ)$ space. This matches the best known time bound while improving the space by a factor $\Theta(m^2k^2)$. Alternatively, if we plugin the Cole-Hariharan algorithm and set $\tau = k^4+m$ we get an algorithm using $O(nk^4+nm+occ)$ time and $O(n/(k^4+m)+m+occ)$ space. Whenever $k = O(m^{1/4})$ this is $O(nm+occ)$ time and $O(n/m+m+occ)$ space.

To the best of our knowledge, all previous non-trivial compressed pattern matching algorithms for ZL78/ZLW compressed text, with the exception of a very slow algorithm for exact string matching by Amir et al. [3], use $\Omega(n)$ space. This is because the algorithms explicitly construct the dictionary trie of the compressed texts. Surprisingly, our results show that for the ZL78 compression schemes this is not needed to get an efficient algorithm. Conversely, if very little space is available our trade-off shows that it is still possible to solve the problem without decompressing the text.

1.2 Regular Expression Matching

Given a regular expression R and a string, Q the *regular expression matching problem* is to find all ending position of substrings in Q that matches a string in the language denoted by R. The classic textbook solution to this problem due to Thompson [21] solves the problem in $O(um)$ time and $O(m)$ space, where u and m are the length of Q and R, respectively. Improvements based on the Four Russian Technique or word-level parallelism are given in [15,6,4].

The only solution to the compressed problem is due to Navarro [17]. His solution depends on word RAM techniques to encode small sets into memory words, thereby allowing constant time set operations. On a unit-cost RAM with w-bit words this technique can be used to improve an algorithm by at most a factor $O(w)$. For $w = O(\log u)$ a similar improvement is straightforward to obtain for our algorithm and we will therefore, for the sake of exposition, ignore this factor in the bounds presented below. With this simplification Navarro's algorithm uses $O(nm^2 + occ \cdot m \log m)$ time and $O(nm^2)$ space, where n is the length of the compressed string. In this paper we show the following time-space trade-off:

Theorem 2. *Let Q be a string compressed using* ZL78 *or* ZLW *into a string Z of length n and let R be a regular expression of length m. Given Z, R, and a parameter $\tau \geq 1$, we can find all occurrences of substrings matching R in Q in $O(nm(m+\tau) + occ \cdot m \log m)$ time and $O(nm^2/\tau + nm)$ space.*

If we choose $\tau = m$ we obtain an algorithm using $O(nm^2 + occ \cdot m \log m)$ time and $O(nm)$ space. This matches the best known time bound while improving the space by a factor $\Theta(m)$. With word-parallel techniques these bounds can be improved slightly.

1.3 Techniques

If pattern matching algorithms for ZL78 or ZLW compressed texts use $\Omega(n)$ space they can explicitly store the dictionary trie for the compressed text and apply any linear space data structure to it. This has proven to be very useful for compressed pattern matching. However, as noted by Amir et al. [3], $\Omega(n)$ may not be feasible for large texts and therefore more space-efficient algorithms are needed. Our main technical contribution is a simple $o(n)$ data structure for `ZL78` compressed texts. The data structure gives a way to compactly represent a subset of the trie which combined with the compressed text enables algorithms to quickly access relevant parts of the trie. This provides a general approach to solve compressed pattern matching problems in $o(n)$ space, which combined with several other techniques leads to the above results. Due to lack of space we have left out the details for regular expression matching. They can be found in the full version of the paper [5].

2 The Ziv-Lempel Compression Schemes

Let Σ be an *alphabet* containing $\sigma = |\Sigma|$ *characters.* A *string* Q is a sequence of characters from Σ. The *length* of Q is $u = |Q|$ and the unique string of length 0 is denoted ϵ. The ith character of Q is denoted $Q[i]$ and the substring beginning at position i of length $j - i + 1$ is denoted $Q[i, j]$. The Ziv-Lempel algorithm from 1978 [24] provides a simple and natural way to represent strings, which we describe below. Define a `ZL78` compressed string (abbreviated *compressed string* in the remainder of the paper) to be a string of the form

$$Z = z_1 \cdots z_n = (r_1, \alpha_1)(r_2, \alpha_2) \ldots (r_n, \alpha_n),$$

where $r_i \in \{0, \ldots, i-1\}$ and $\alpha_i \in \Sigma$. Each pair $z_i = (r_i, \alpha_i)$ is a *compression element*, and r_i and α_i are the *reference* and *label* of z_i, denoted by $\text{reference}(z_i)$ and $\text{label}(z_i)$, respectively. Each compression element *represents* a string, called a *phrase.* The phrase for z_i, denoted $\text{phrase}(z_i)$, is given by the following recursion.

$$\text{phrase}(z_i) = \begin{cases} \text{label}(z_i) & \text{if reference}(z_i) = 0, \\ \text{phrase}(\text{reference}(z_i)) \cdot \text{label}(z_i) & \text{otherwise.} \end{cases}$$

The $\cdot$ denotes concatenation of strings. The compressed string Z *represents* the concatenation of the phrases, i.e., the string $\text{phrase}(z_1) \cdots \text{phrase}(z_n)$.

Let Q be a string of length u. In `ZL78`, the compressed string representing Q is obtained by greedily parsing Q from left-to-right with the help of a dictionary D. For simplicity in the presentation we assume the existence of an initial compression element $z_0 = (0, \epsilon)$ where $\text{phrase}(z_0) = \epsilon$. Initially, let $z_0 = (0, \epsilon)$ and let $D = \{\epsilon\}$. After step i we have computed a compressed string $z_0 z_1 \cdots z_i$ representing $Q[1, j]$ and $D = \{\text{phrase}(z_0), \ldots, \text{phrase}(z_i)\}$. We then find the longest prefix of $Q[j+1, u-1]$ that matches a string in D, say $\text{phrase}(z_k)$, and let $\text{phrase}(z_{i+1}) = \text{phrase}(z_k) \cdot Q[j + 1 + |\text{phrase}(z_k)|]$. Set $D = D \cup \{\text{phrase}(z_{i+1})\}$ and let $z_{i+1} = (k, Q[j + 1 + |\text{phrase}(z_{i+1})|])$. The compressed string $z_0 z_1 \ldots z_{i+1}$ now represents the string $Q[1, j + |\text{phrase}(z_{i+1})|])$ and $D = \{\text{phrase}(z_0), \ldots, \text{phrase}(z_{i+1})\}$. We repeat this process until all of Q has been read.

$Q = \text{ananas}$

$Z = (0,\text{a})(0,\text{n})(1,\text{n})(1,\text{s})$

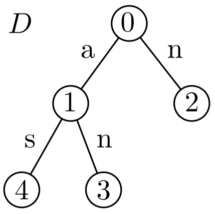


Fig. 1. The compressed string Z representing Q and the corresponding dictionary trie D. Taken from [17].

Since each phrase is the concatenation of a previous phrase and a single character, the dictionary D is prefix-closed, i.e., any prefix of a phrase is a also a phrase. Hence, we can represent it compactly as a trie where each node i corresponds to a compression element z_i and phrase(z_i) is the concatenation of the labels on the path from z_i to node i. Due to greediness, the phrases are unique and therefore the number of nodes in D for a compressed string Z of length n is $n + 1$. An example of a string and the corresponding compressed string is given in Fig. 1.

Throughout the paper we will identify compression elements with nodes in the trie D, and therefore we use standard tree terminology, briefly summed up here: The *distance* between two elements is the number of edges on the unique simple path between them. The *depth* of element z is the distance from z to z_0 (the root of the trie). An element x is an *ancestor* of an element z if phrase(x) is a prefix of phrase(z). If also $|\text{phrase}(x)| = |\text{phrase}(z)| - 1$ then x is the *parent* of z. If x is ancestor of z then z is a *descendant* of x and if x is the parent of z then z is the *child* of x.The *length* of a path p is the number of edges on the path, and is denoted $|p|$. The *label* of a path is the concatenation of the labels on these edges.

Note that for a compression element z, reference(z) is a pointer to the parent of z and label(z) is the label of the edge to the parent of z. Thus, given z we can use the compressed text Z directly to decode the label of the path from z towards the root in constant time per element. We will use this important property in many of our results.

If the dictionary D is implemented as a trie it is straightforward to compress Q or decompress Z in $O(u)$ time. Furthermore, if we do not want to explicitly decompress Z we can compute the trie in $O(n)$ time, and as mentioned above, this is done in almost all previous compressed pattern matching algorithm on Ziv-Lempel compression schemes. However, this requires at least $\Omega(n)$ space which is insufficient to achieve our bounds. In the next section we show how to partially represent the trie in less space.

2.1 Selecting Compression Elements

Let $Z = z_0 \dots z_n$ be a compressed string. For our results we need an algorithm to select a compact subset of the compression elements such that the distance from any element to an element in the subset is no larger than a given threshold. More precisely, we show the following lemma.

Lemma 1. *Let Z be a compressed string of length n and let $1 \leq \tau \leq n$ be parameter. There is a set of compression elements C of Z, computable in $O(n\tau)$ expected time and $O(n/\tau)$ space with the following properties:*

(i) $|C| = O(n/\tau)$.
(ii) For any compression element z_i in Z, the minimum distance to any compression element in C is at most 2τ.

Proof. Let $1 \leq \tau \leq n$ be a given parameter. We build C incrementally in a left-to-right scan of Z. The set is maintained as a dynamic dictionary using dynamic perfect hashing [8], i.e., constant time worst-case access and constant time amortized expected update. Initially, we set $C = \{z_0\}$. Suppose that we have read $z_0, \ldots, z_i$. To process z_{i+1} we follow the path p of references until we encounter an element y such that $y \in C$. We call y the *nearest special element* of z_{i+1}. Let l be the number of elements in p including z_{i+1} and y. Since each lookup in C takes constant time the time to find the nearest special element is $O(l)$. If $l < 2 \cdot \tau$ we are done. Otherwise, if $l = 2 \cdot \tau$, we find the τth element y' in the reference path and set $C := C \cup \{y'\}$. As the trie grows under addition of leaves condition (ii) follows. Moreover, any element chosen to be in C has at least τ descendants of distance at most τ that are not in C and therefore condition (i) follows. The time for each step is $O(\tau)$ amortized expected and therefore the total time is $O(n\tau)$ expected. The space is proportional to the size of C hence the result follows. $\square$

2.2 Other Ziv-Lempel Compression Schemes

A popular variant of ZL78 is the ZLW compression scheme [22]. Here, the label of compression elements are not explicitly encoded, but are defined to be the first character of the next phrase. Hence, ZLW does not offer an asymptotically better compression ratio over ZL78 but gives a better practical performance. The ZLW scheme is implemented in the UNIX program `compress`. From an algorithmic viewpoint ZLW is more difficult to handle in a space-efficient manner since labels are not explicitly stored with the compression elements as in ZL78. However, if $\Omega(n)$ space is available then we can simply construct the dictionary trie. This gives constant time access to the label of a compression elements and therefore ZL78 and ZLW become "equivalent". This is the reason why Theorem 1 holds only for ZL78 when space is $o(n)$ but for both when the space is $\Omega(n)$.

Another well-known variant is the ZL77 compression scheme [23]. Unlike ZL78 and ZLW phrases in the ZL77 scheme can be any substring of text that has already been processed. This makes searching much more difficult and none of the known techniques for ZL78 and ZLW seems to be applicable. The only known algorithm for pattern matching on ZL77 compressed text is due to Farach and Thorup [9] who gave an algorithm for the exact string matching problem.

3 Approximate String Matching

In this section we consider the compressed approximate string matching problem. Before presenting our algorithm we need a few definitions and properties of approximate string matching.

Let A and B be strings. Define the *edit distance* between A and B, $\gamma(A, B)$, to be the minimum number of insertions, deletions, and substitutions needed to transform A to B. We say that $j \in [1, |S|]$ is a *match with error at most* k of A in a string S if there is an $i \in [1, j]$ such that $\gamma(A, S[i, j]) \leq k$. Whenever k is clear from the context we simply call j a *match*. All positions i satisfying the above property are called a *start* of the match j. The set of all matches of A in S is denoted $\Gamma(A, S)$. We need the following well-known property of approximate matches.

Proposition 1. *Any match j of A in S with at most k errors must start in the interval* $[\max(1, j - |A| + 1 - k), \min(|S|, j - |A| + 1 + k)]$.

Proof. Let l be the length of a substring B matching A and ending at j. If the match starts outside the interval then either $l < |A| - k$ or $l > |A| + k$. In these cases, more than k deletions or k insertions, respectively, are needed to transform B to A. □

3.1 Searching for Matches

Let P be a string of length m and let k be an error threshold. To avoid trivial cases we assume that $k < m$. Given a compressed string $Z = z_0 z_1 \ldots z_n$ representing a string Q of length u we show how to find $\Gamma(P, Q)$ efficiently.

Let $l_i = |\text{phrase}(z_i)|$, let $u_0 = 1$, and let $u_i = u_{i-1} + l_{i-1}$, for $1 \leq i \leq n$, i.e., l_i is the length of the ith phrase and u_i is the starting position in Q of the ith phrase. We process Z from left-to-right and at the ith step we find all matches in $[u_i, u_i + l_i - 1]$. Matches in this interval can be either *internal* or *overlapping* (or both). A match j in $[u_i, u_i + l_i - 1]$ is internal if it has a starting point in $[u_i, u_i + l_i - 1]$ and overlapping if it has a starting point in $[1, u_i - 1]$. To find all matches we will compute the following information for z_i.

– The start position, u_i, and length, l_i, of phrase(z_i).
– The *relevant prefix*, rpre(z_i), and the *relevant suffix*, rsuf(z_i), where

$$\begin{aligned}\text{rpre}(z_i) &= Q[u_i, \min(u_i + m + k - 1, u_i + l_i - 1)] \ ,\\ \text{rsuf}(z_i) &= Q[\max(1, u_i + l_i - m - k), u_i + l_i - 1] \ .\end{aligned}$$

  In other words, rpre(z_i) is the largest prefix of length at most $m + k$ of phrase(z_i) and rsuf(z_i) is the substring of length $m + k$ ending at $u_i + l_i - 1$. For an example see Fig. 2.
– The *match sets* $M_I(z_i)$ and $M_O(z_i)$, where

$$\begin{aligned}M_I(z_i) &= \Gamma(P, \text{phrase}(z_i)) \ ,\\ M_O(z_i) &= \Gamma(P, \text{rsuf}(z_{i-1}) \cdot \text{rpre}(z_i)) \ .\end{aligned}$$

  We assume that both sets are represented as sorted lists in increasing order.

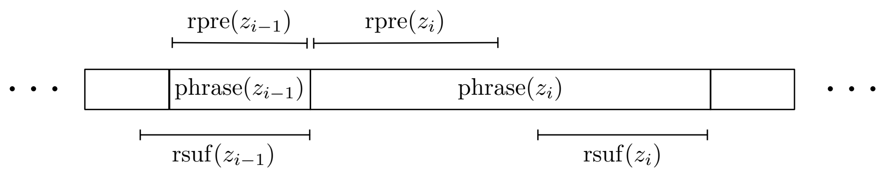


Fig. 2. The relevant prefix and the relevant suffix of two phrases in Q. Here, $|\text{phrase}(z_{i-1})| < m + k$ and therefore $\text{rsuf}(z_{i-1})$ overlaps with previous phrases.

We call the above information the *description* of z_i. In the next section we show how to efficiently compute descriptions. For now, assume that we are given the description of z_i. Then, the set of matches in $[u_i, u_i + l_i - 1]$ is reported as the set

$$\begin{aligned}M(z_i) = {} & \{j + u_i - 1 \mid j \in M_I(z_i)\} \cup \\ & \{j + u_i - 1 - |\text{rsuf}(z_{i-1})| \mid j \in M_O(z_i) \cap [u_i, u_i + l_i - 1]\} .\end{aligned}$$

We argue that this is the correct set. Since $\text{phrase}(z_i) = Q[u_i, u_i + l_i - 1]$ we have that

$$j \in M_I(z_i) \Leftrightarrow j + u_i - 1 \in \Gamma(P, Q[u_i, u_i + l_i - 1] .$$

Hence, the set $\{j+u_i-1 \mid j \in M_I(z_i)\}$ is the set of all internal matches. Similarly, $\text{rsuf}(z_{i-1}) \cdot \text{rpre}(z_i) = Q[u_i - |\text{rsuf}(z_{i-1})|, u_i + |\text{rpre}(z_i)| - 1]$ and therefore

$$j \in M_O(z_i) \Leftrightarrow j+u_i-1-|\text{rsuf}(z_{i-1})| \in \Gamma(P, Q[u_i-|\text{rsuf}(z_{i-1})|, u_i+1+|\text{rpre}(z_i)|]).$$

By Proposition 1 any overlapping match must start at a position within the interval $[\max(1, u_i-m+1-k), u_i]$. Hence, $\{j+u_i-1-|\text{rsuf}(z_{i-1})| \mid j \in M_O(z_i)\}$ includes all overlapping matches in $[u_i, u_i + l_i - 1]$. Taking the intersection with $[u_i, u_i + l_i - 1]$ and the union with the internal matches it follows that the set $M(z_i)$ is precisely the set of matches in $[u_i, u_i+l_i-1]$. For an example see Fig. 3.

Next we consider the complexity of computing the matches. To do this we first bound the size of the M_I and M_O sets. Since the length of any relevant suffix and relevant prefix is at most $m + k$, we have that $|M_O(z_i)| \leq 2(m + k) < 4m$, and therefore the total size of the M_O sets is at most $O(nm)$. Each element in the sets $M_I(z_0), \dots, M_I(z_n)$ corresponds to a unique match. Thus, the total size of the M_I sets is at most *occ*, where *occ* is the total number of matches. Since both sets are represented as sorted lists the total time to compute the matches for all compression elements is $O(nm + occ)$.

3.2 Computing Descriptions

Next we show how to efficiently compute the descriptions. Let $1 \leq \tau \leq n$ be a parameter. Initially, we compute a subset C of the elements in Z according to Lemma 1 with parameter τ. For each element $z_j \in C$ we store l_j, that is, the length of $\text{phrase}(z_j)$. If $l_j > m + k$ we also store the index of the ancestor x of

Q = ananasbananer, P = base, Z = (0,a)(0,n)(1,n)(1,s)(0,b)(3,a)(2,e)(0,r)

Descriptions

	z_0	z_1	z_2	z_3	z_4	z_5	z_6	z_7
u_i	1	2	3	5	7	8	11	13
l_i	1	1	2	2	1	3	2	1
rpre(z_i)	a	n	an	as	b	ana	ne	r
rsuf(z_i)	a	an	anas	ananas	nanasb	asbana	banane	ananer
$M_I(z_i)$	$\emptyset$	$\emptyset$	$\emptyset$	$\{2\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$M_O(z_i)$	$\emptyset$	$\emptyset$	$\emptyset$	$\{6\}$	$\{6,7\}$	$\{5,6,7,8\}$	$\{2,3,4,5,6\}$	$\{2,3,4,6\}$
$M(z_i)$	$\emptyset$	$\emptyset$	$\emptyset$	$\{6\}$	$\{7\}$	$\{8,9,10\}$	$\{12\}$	$\emptyset$

Fig. 3. Example of descriptions. Z is the compressed string representing Q. We are looking for all matches of the pattern P with error threshold $k = 2$ in Z. The set of matches is $\{6, 7, 8, 9, 10, 12\}$.

z_j of depth $m + k$. This information can easily be computed while constructing C within the same time and space bounds, i.e., using $O(n\tau)$ time and $O(n/\tau)$ space.

Descriptions are computed from left-to-right as follows. Initially, set $l_0 = 0$, $u_0 = 0$, rpre$(z_0) = \epsilon$, rsuf$(z_0) = \epsilon$, $M_I(z_0) = \emptyset$, and $M_O(z_0) = \emptyset$. To compute the description of z_i, $1 \le i \le n$, first follow the path p of references until we encounter an element $z_j \in C$. Using the information stored at z_j we set $l_i := |p| + l_j$ and $u_i = u_{i-1} + l_{i-1}$. By Lemma 1(ii) the distance to z_j is at most 2τ and therefore l_i and u_i can be computed in $O(\tau)$ time given the description of z_{i-1}.

To compute rpre(z_i) we compute the label of the path from z_0 towards z_i of length $\min(m + k, l_i)$. There are two cases to consider: If $l_i \le m + k$ we simply compute the label of the path from z_i to z_0 and let rpre(z_i) be the reverse of this string. Otherwise ($l_i > m + k$), we use the "shortcut" stored at z_j to find the ancestor z_h of distance $m + k$ to z_0. The reverse of the label of the path from z_h to z_0 is then rpre(z_i). Hence, rpre(z_i) is computed in $O(m + k + \tau) = O(m + \tau)$ time.

The string rsuf(z_i) may be the divided over several phrases and we therefore recursively follow paths towards the root until we have computed the entire string. It is easy to see that the following algorithm correctly decodes the desired substring of length $\min(m + k, u_i)$ ending at position $u_i + l_i - 1$.

1. Initially, set $l := \min(m + k, u_i + l_i - 1)$, $t := i$, and $s := \epsilon$.
2. Compute the path p of references from z_t of length $r = \min(l, \text{depth}(z_t))$ and set $s := s \cdot \text{label}(p)$.
3. If $r < l$ set $l := l - r$, $t := t - 1$, and repeat step 2.
4. Return rsuf(z_i) as the reverse of s.

Since the length of rsuf(z_i) is at most $m+k$, the algorithm finds it in $O(m+k) = O(m)$ time.

The match sets M_I and M_O are computed as follows. Let $t(m, u, k)$ and $s(m, u, k)$ denote the time and space to compute $\Gamma(A, B)$ with error threshold k

for strings A and B of lengths m and u, respectively. Since $|\mathrm{rsuf}(z_{i-1})\cdot\mathrm{rpre}(z_i)| \leq 2m + 2k$ it follows that $M_O(z_i)$ can be computed in $t(m, 2m + 2k, k)$ time and $s(m, 2m+2k, k)$ space. Since $M_I(z_i) = \Gamma(P, \mathrm{phrase}(z_i))$ we have that $j \in M_I(z_i)$ if and only if $j \in M_I(\mathrm{reference}(z_i))$ or $j = l_i$. By Proposition 1 any match ending in l_i must start within $[\max(1, l_i - m + 1 - k), \min(l_i, l_i - m + 1 + k)]$. Hence, there is a match ending in l_i if and only if $l_i \in \Gamma(P, \mathrm{rsuf}'(z_i))$ where $\mathrm{rsuf}'(z_i)$ is the suffix of $\mathrm{phrase}(z_i)$ of length $\min(m + k, l_i)$. Note that $\mathrm{rsuf}'(z_i)$ is a suffix of $\mathrm{rsuf}(z_i)$ and we can therefore trivially compute it in $O(m + k)$ time. Thus,

$$M_I(z_i) = M_I(\mathrm{reference}(z_i)) \cup \{l_i \mid l_i \in \Gamma(P, \mathrm{rsuf}'(z_i))\} \ .$$

Computing $\Gamma(P, \mathrm{rsuf}'(z_i))$ uses $t(m, m + k, k)$ time and $s(m, m + k, k)$ space. Subsequently, constructing $M_I(z_i)$ takes $O(|M_I(z_i)|)$ time and space. Recall that the elements in the M_I sets correspond uniquely to matches in Q and therefore the total size of the sets is occ. Therefore, using dynamic perfect hashing [8] on pointers to non-empty M_I sets we can store these using in total $O(occ)$ space.

3.3 Analysis

Finally, we can put the pieces together to obtain the final algorithm. The preprocessing uses $O(n\tau)$ expected time and $O(n/\tau)$ space. The total time to compute all descriptions and report occurrences is expected $O(n(\tau + m + t(m, 2m + 2k, k)) + occ)$. The description for z_i, except for $M_I(z_i)$, depends solely on the description of z_{i-1}. Hence, we can discard the description of z_{i-1}, except for $M_I(z_{i-1})$, after processing z_i and reuse the space. It follows that the total space used is $O(n/\tau + m + s(m, 2m + 2k, k) + occ)$. This completes the proof of Theorem 1. Note that if we use $\Omega(n)$ space we can explicitly construct the dictionary. In this case hashing is not needed and the bounds also hold for the `ZLW` compression scheme.

References

1. Amir, A., Benson, G.: Efficient two-dimensional compressed matching. In: Proceedings of the 2nd Data Compression Conference, pp. 279–288 (1992)
2. Amir, A., Benson, G.: Two-dimensional periodicity and its applications. In: Proceedings of the 3rd Symposium on Discrete algorithms, pp. 440–452 (1992)
3. Amir, A., Benson, G., Farach, M.: Let sleeping files lie: pattern matching in Z-compressed files. J. Comput. Syst. Sci. 52(2), 299–307 (1996)
4. Bille, P.: New algorithms for regular expression matching. In: Proceedings of the 33rd International Colloquium on Automata, Languages and Programming, pp. 643–654 (2006)
5. Bille, P., Fagerberg, R., Gørtz, I.L.: Improved approximate string matching and regular expression matching on ziv-lempel compressed texts (2007) Draft of full version available at `arxiv.org/cs/DS/0609085`
6. Bille, P., Farach-Colton, M.: Fast and compact regular expression matching, Submitted to a journal (2005) Preprint availiable at `arxiv.org/cs/0509069`

7. Cole, R., Hariharan, R.: Approximate string matching: A simpler faster algorithm. SIAM J. Comput. 31(6), 1761–1782 (2002)
8. Dietzfelbinger, M., Karlin, A., Mehlhorn, K., auf der Heide, F.M., Rohnert, H., Tarjan, R.: Dynamic perfect hashing: Upper and lower bounds. SIAM J. Comput. 23(4), 738–761 (1994)
9. Farach, M., Thorup, M.: String matching in Lempel-Ziv compressed strings. Algorithmica 20(4), 388–404 (1998)
10. Kärkkäinen, J., Navarro, G., Ukkonen, E.: Approximate string matching on Ziv-Lempel compressed text. J. Discrete Algorithms 1(3-4), 313–338 (2003)
11. Kida, T., Takeda, M., Shinohara, A., Miyazaki, M., Arikawa, S.: Multiple pattern matching in LZW compressed text. In: Proceedings of the 8th Data Compression Conference, pp. 103–112 (1998)
12. Landau, G.M., Vishkin, U.: Fast parallel and serial approximate string matching. J. Algorithms 10(2), 157–169 (1989)
13. Mäkinen, V., Ukkonen, E., Navarro, G.: Approximate matching of run-length compressed strings. Algorithmica 35(4), 347–369 (2003)
14. Matsumoto, T., Kida, T., Takeda, M., Shinohara, A., Arikawa, S.: Bit-parallel approach to approximate string matching in compressed texts. In: Proceedings of the 7th International Symposium on String Processing and Information Retrieval, pp. 221–228 (2000)
15. Myers, E.W.: A four-russian algorithm for regular expression pattern matching. J. ACM 39(2), 430–448 (1992)
16. Navarro, G.: A guided tour to approximate string matching. ACM Comput. Surv. 33(1), 31–88 (2001)
17. Navarro, G.: Regular expression searching on compressed text. J. Discrete Algorithms 1(5-6), 423–443 (2003)
18. Navarro, G., Kida, T., Takeda, M., Shinohara, A., Arikawa, S.: Faster approximate string matching over compressed text. In: Proceedings of the Data Compression Conference (DCC '01), p. 459. IEEE Computer Society, Washington, DC, USA (2001)
19. Navarro, G., Raffinot, M.: A general practical approach to pattern matching over Ziv-Lempel compressed text. Technical Report TR/DCC-98-12, Dept. of Computer Science, Univ. of Chile (1998)
20. Sellers, P.: The theory and computation of evolutionary distances: Pattern recognition. J. Algorithms 1, 359–373 (1980)
21. Thompson, K.: Programming techniques: Regular expression search algorithm. Commun. ACM 11, 419–422 (1968)
22. Welch, T.A.: A technique for high-performance data compression. IEEE Computer 17(6), 8–19 (1984)
23. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Trans. Inform. Theory 23(3), 337–343 (1977)
24. Ziv, J., Lempel, A.: Compression of individual sequences via variable-rate coding. IEEE Trans. Inform. Theory 24(5), 530–536 (1978)

Self-normalised Distance with Don't Cares

Peter Clifford[1] and Raphaël Clifford[2]

[1] Dept. of Statistics, University of Oxford, UK
clifford@stats.ox.ac.uk

[2] Dept. of Computer Science, University of Bristol, Bristol, BS8 1UB, UK
clifford@cs.bris.ac.uk

Abstract. We present $O(n \log m)$ algorithms for a new class of problems termed *self-normalised distance with don't cares*. The input is a pattern p of length m and text t of length $n > m$. The elements of these strings are either integers or wild card symbols. In the shift version, the problem is to compute $\min_\alpha \sum_{j=0}^{m-1}(\alpha + p_j - t_{i+j})^2$ for all i, where wild cards do not contribute to the sum. In the shift-scale version, the objective is to compute $\min_{\alpha,\beta} \sum_{j=0}^{m-1}(\alpha + \beta p_j - t_{i+j})^2$ for all i, similarly. We show that the algorithms have the additional benefit of providing simple $O(n \log m)$ solutions for the problems of exact matching with don't cares, exact shift matching with don't cares and exact shift-scale matching with don't cares.

Classification: Algorithms and data structures; pattern matching; string algorithms.

1 Introduction

A fundamental problem in image processing is to measure the similarity between a small image segment or template and regions of comparable size within a larger scene. It is well known that the cross-correlation between the two can be computed efficiently at every position in the larger image using the fast Fourier transform (FFT). In practice, images may differ in a number of ways including being rotated, scaled or affected by noise. We consider here the case where the intensity or brightness of an image occurrence is unknown and where parts of either image contain *don't care* or *wild card* pixels, i.e. pixels that are considered to be irrelevant as far as image similarity is concerned. As an example, a rectangular image segment may contain a facial image and the objective is to identify the face in a larger scene. However, some faces in the larger scene are in shadow and others in light. Furthermore, background pixels around the faces may be considered to be irrelevant for facial recognition and these should not affect the search algorithm.

In order to overcome the first difficulty of varying intensity within an image, a standard approach is to compute the *normalised cross-correlation* when comparing a template to part of a larger image (see e.g. [12]). Thus both template and image are rescaled in order to make any matches found more meaningful and to allow comparisons between matches at different positions.

For simplicity we will describe algorithms in a one-dimensional context involving one-dimensional FFTs. The extension to two and higher dimensions is straightforward.

Consider two strings with elements that are either integers in the range $1, \ldots, N$ or don't care symbols. Let the text $t = t_0, \ldots, t_{n-1}$ and pattern $p = p_0, \ldots, p_{m-1}$. In the absence of don't care symbols, the squared Euclidean distance between the pattern and the text at position i is $\sum_{j=0}^{m-1}(p_j - t_{i+j})^2$. In this case, for each i, the pattern can be *normalised*, or fitted as closely as possible to the text, by adding an equal quantity to its values to minimise the distance. The fit can then be assessed by computing

$$A_i = \min_\alpha \sum_{j=0}^{m-1}(\alpha + p_j - t_{i+j})^2$$

for each i.

When either p_j or t_{i+j} is a don't care symbol the contribution of the pair to the sum is taken to be zero, i.e. they make no contribution to the distance and the minimisation is carried out using the sum of the remaining terms. The objective is to find the resulting values of A_i for each i. We call this the shift version of the *self-normalised distance problem with don't cares.*

More generally we can consider the problem of computing

$$B_i = \min_{\alpha,\beta} \sum_{j=0}^{m-1}(\alpha + \beta p_j - t_{i+j})^2$$

for each i with a similar convention for don't care symbols. We call this the shift-scale version of the self-normalised distance problem with don't cares. We show that in both cases the self-normalised distance problem can be solved in $O(n \log m)$ by the use of fast Fourier transforms of integer vectors[1].

Our main results also provide a $O(n \log m)$ time solution to the classic problem of exact matching with don't cares, which we believe is conceptually simpler than existing methods since it only involves integer codes. Further, we obtain the same time complexity for the problem of exact shift matching with don't cares and the new problem of exact shift-scale matching with don't cares.

1.1 Previous Work

Combinatorial pattern matching has concerned itself mainly with strings of symbolic characters where the distance between individual characters is specified by some convention. However in recent years there has been a surge in interest in provably fast algorithms for distance calculation and approximate matching between numerical strings. Many different metrics have been considered, with for example, $O(n\sqrt{m \log m})$ time solutions found for the Hamming distance [1, 14], L_1 distance [4, 6, 3] and *less-than* matching [2] problems and an $O(\delta n \log m)$ algorithm for the δ-bounded version of the L_∞ norm first discussed in [7] and then improved in [6, 16]. A related problem to the shift version of self-normalised

[1] We assume the RAM model of computation throughout in order to be consistent with previous work on matching with wild cards.

distance is transposition invariant matching (see e.g. [17]) which has been considered under a number of different distance measures, including edit distance and longest common subsequence. In the image processing literature, a fast and practical solution to the problem of computing fast normalised cross-correlation for template matching was given in [15].

The problem of determining the time complexity of exact matching with don't cares on the other hand has been well studied over many years. Fischer and Paterson [10] presented the first solution based on fast Fourier transforms (FFT) with an $O(n \log m \log |\Sigma|)$ time algorithm in 1974. Subsequently, the major challenge has been to remove the dependency on the alphabet size. Indyk [11] gave a randomised $O(n \log n)$ time algorithm which was followed by a simpler and slightly faster $O(n \log m)$ time randomised solution by Kalai [13]. In 2002 a deterministic $O(n \log m)$ time solution was given by Cole and Hariharan [8] and further simplified in [5]. In the same paper an $O(n \log(\max(m, N)))$ time algorithm for the exact shift matching problem with strings of positive integers is presented, where N is the largest value in the input.

Notation. With two numerical strings x and y of equal length, we use the notation xy for the string with elements $x_i y_i$. Similarly, $x^2 y$ is the string with elements $x_i^2 y_i$, and x/y is the string with elements x_i/y_i, etc. Our algorithms make extensive use of the fast Fourier transform (FFT). An important property of the FFT is that all the inner-products,

$$(t \otimes p)[i] \overset{\text{def}}{=} \textstyle\sum_{j=0}^{m-1} p_j t_{i+j}, \quad 0 \le i \le n-m,$$

can be calculated accurately and efficiently in $O(n \log m)$ time (see e.g. [9], Chapter 32). In order to reduce the time complexity from $O(n \log n)$ to $O(n \log m)$ we employ a standard trick. The text is partitioned into n/m overlapping substrings of length $2m$ and the matching algorithm is performed on each one. Each iteration takes $O(m \log m)$ time giving an overall time complexity of $O((n/m)m \log m)$ $= O(n \log m)$.

Accuracy. Since we only make use of integer codes, the numerical accuracy of the algorithms presented in this paper need only be sufficient to distinguish 0 from other integer values. By way of comparison, the well known deterministic exact matching with wildcards algorithm of [8] involves coding into rational numbers that are subsequently represented with fixed accuracy in fast Fourier transform calculations. Furthermore, these calculations must be accurate enough to distinguish 0 from values as small as $1/N(N-1)$ (see [8] for further details).

2 Problems and Solutions

Let text $t = t_0, \ldots, t_{n-1}$ and pattern $p = p_0, \ldots, p_{m-1}$. We give $O(n \log m)$ solutions for shift and shift-scale versions of the self-normalised distance problem with don't cares. We show that this enables us to solve exact matching, exact shift matching and exact shift-scale matching in the same time complexity for inputs containing don't care symbols.

2.1 Self-normalised Distance with Don't Cares

In the self-normalised distance with don't cares problem, p and t are strings with elements that are either numerical or wild card values. We use $*$ for the wild card symbol and define a new string p' with $p'_j = 0$ if $p_j = *$, and $p'_j = 1$ otherwise. Similarly, define $t'_i = 0$ if $t_i = *$, and 1 otherwise.

We can now express the shift version of the self-normalised squared distance at position i as $A_i = \min_\alpha \sum (\alpha + p_j - t_{i+j})^2 p'_j t'_{i+j}$ where here and in the remainder of this paper we will use $\sum$ as an abbreviation for $\sum_{j=0}^{m-1}$.

```
**Input**: Pattern $p$ and text $t$
**Output**: $A_i = \min_\alpha \sum(\alpha + p_j - t_{i+j})^2 p'_j t'_{i+j}, \quad 0 \le i \le n-m$
Create binary string $p'_j = 0$ if $p_j = *$, 1 otherwise;
Create binary string $t'_i = 0$ if $t_i = *$, 1 otherwise;
$A \leftarrow (t^2 t') \otimes p' - 2(tt') \otimes (pp') + t' \otimes (p^2 p') - [(tt') \otimes p' - t' \otimes (pp')]^2 / (t' \otimes p')$
```

Algorithm 1. Shift version of self-normalised distance with don't cares

Theorem 1. *The shift version of the self-normalised distance with don't cares problem can be solved in $O(n \log m)$ time.*

Proof. Consider Algorithm 1. The first two steps require only single passes over the input. Similarly p^2p', pp', tt' and t^2t' can all be calculated in linear time once t' and p' are known. This leaves 6 correlation calculations to compute A. Using the FFT, each correlation calculation takes $O(n \log m)$ time which is therefore the overall time complexity of the algorithm.

To find A_i, the minimum value of $\sum(\alpha + p_j - t_{i+j})^2 p'_j t'_{i+j}$, we differentiate with respect to α and obtain the minimising value as

$$\widehat{\alpha} = \frac{\sum (t_{i+j} - p_j) p'_j t'_{i+j}}{\sum p'_j t'_{i+j}} = \frac{((tt') \otimes p' - (pp') \otimes t')\,[i]}{(p' \otimes t')[i]}. \tag{1}$$

Substituting $\alpha = \widehat{\alpha}$, expanding and collecting terms we find A_i is the ith element of

$$(t^2 t') \otimes p' - 2(tt') \otimes (pp') + t' \otimes (p^2 p') - \frac{((tt') \otimes p' - t' \otimes (pp'))^2}{(t' \otimes p')}$$

Similarly, the minimised shift-scale distance at position i can be written as $B_i = \min_{\alpha,\beta} \sum (\alpha + \beta p_j - t_{i+j})^2 p'_j t'_{i+j}$.

Theorem 2. *The shift-scale version of the self-normalised distance with don't cares problem can be solved in $O(n \log m)$ time.*

Proof. Consider Algorithm 2. Notice that the same 6 correlations have to be calculated for this problem so the running time is $O(n \log m)$.

```
**Input**: Pattern $p$ and text $t$
**Output**: $B_i = \min_{\alpha,\beta} \sum(\alpha + \beta p_j - t_{i+j})^2 p'_j t'_{i+j}, \quad 0 \le i \le n-m$
Create binary string $p'_j = 0$ if $p_j = *$, 1 otherwise;
Create binary string $t'_i = 0$ if $t_i = *$, 1 otherwise;
$T_1 \leftarrow (tt') \otimes (pp') - [(tt') \otimes p'] [t' \otimes (pp')] / (t' \otimes p')$ ;
$T_2 \leftarrow t' \otimes (p^2 p') - [t' \otimes (pp')]^2 / (t' \otimes p')$;
$B \leftarrow (t^2 t') \otimes p' - [(tt') \otimes (p')]^2 / (t' \otimes p') - T_1^2 / T_2$;
```

Algorithm 2. Shift-scale version of self-normalised distance with don't cares

To find B_i, the minimum value of $\sum(\alpha + \beta p_j - t_{i+j})^2 p'_j t'_{i+j}$, we differentiate with respect to both α and β to obtain the minimising values $\widehat{\alpha}$ and $\widehat{\beta}$. Substituting these values, expanding and collecting terms we find

$$B_i = \left((t^2 t') \otimes p'\right)[i] - [(tt') \otimes (p')]^2 [i] / (t' \otimes p')[i] - T_i^2 / U_i,$$

where

$$T_i = ((tt') \otimes (pp'))[i] - ((tt') \otimes p')[i] (t' \otimes (pp'))[i] / (t' \otimes p')[i]$$

and

$$U_i = \left(t' \otimes (p^2 p')\right)[i] - (t' \otimes (pp'))^2 [i] / (t' \otimes p')[i].$$

Note that the shift approximation $\widehat{\alpha}$ at each location is given by (1) and involves just three cross-correlations. Similar expressions are available for the minimising values of α and β in the shift-scale distance problem.

3 Exact Matching with Don't Cares

In their seminal paper Cole and Hariharan [8] show how to use non-boolean coding to solve the problems of exact matching and exact shift matching with wildcards (don't cares). They show that their algorithms having running times of $O(n \log(m))$ and $O(n \log(\max(m, N))$ respectively.

In the exact matching problem, the pattern p is said to *occur* at location i in t if, for every position j in the pattern, either $p_j = t_{i+j}$ or at least one of p_j and t_{i+j} is the wild card symbol. The exact matching problem is to find all locations where the pattern occurs.

Cole and Hariharan use four codings for the exact matching problem. In the first two codings, wildcards in p and t are replaced by 0 and non-wildcards by 1. This gives p' and t'. The second coding replaces wildcards in t by 0 and any number a by $1/a$. The resulting string is t''. Finally, the same coding is used to generate p''. Cross-correlations are then calculated using the FFT, giving $p' \otimes t', p' \otimes t''$ and $t' \otimes p''$ and an exact match at location i is then declared if

$$((pp') \otimes t'' + (tt') \otimes p'' - 2p' \otimes t')[i] = 0.$$

Coleman and Hariharan show that this quantity can be as small as $1/(N(N-1))$ when it is non-zero, so that calculations must be carried out with sufficient precision to make this distinction. In simple terms the algorithm can be thought of as testing whether

$$\sum \frac{p'_j t'_{i+j}(p_j - t_{i+j})^2}{p_j t_{i+j}} = \sum p'_j p_j t''_{i+j} - 2p'_j t'_{i+j} + p''_j t'_{i+j} t_{i+j}) = 0,$$

A simple alternative is to test whether

$$\sum p'_j t'_{i+j}(p_j - t_{i+j})^2 = \sum (p'_j p_j^2 t'_{i+j} - 2p'_j p_j t'_{i+j} t_{i+j} + p'_j t'_{i+j} t^2_{i+j})$$

equals 0. This is similar to the approach taken in [5]. The advantage now is that all calculations are in integers.

Exact Shift Matching with Don't Cares

For the exact shift matching problems with wildcards, a match is said to occur at location i if, for some shift α and for every position j in the pattern, either $\alpha + p_j = t_{i+j}$ or at least one of p_j and t_{i+j} is the wild card symbol. Coleman and Hariharan [8] introduce a new coding for this problem that maps the string elements into 0 for wildcards and complex numbers of modulus 1 otherwise. The FFT is then used to find the (complex) cross-correlation between these coded strings and finally a shift match is declared at location i if the ith element of the modulus of the cross-correlation is equal to $(p' \otimes t')[i]$.

Our Algorithm 1 provides a straightforward alternative method for shift matching with wildcards. It has the advantage of only using simple integer codings. Since Algorithm 1 finds the minimum L_2 distance at location i, over all possible shifts, it is only necessary to test whether this distance is zero. The running time for the test is then $O(n \log(m)$ since it is determined by the running time of Algorithm 1.

```
Input: Pattern $p$ and text $t$
Output: $A_i = 0$ iff $p$ has a shift occurrence at location $i$ in $t$
$A \leftarrow$ Algorithm 1 $(p, t)$
```

Algorithm 3. Algorithm for exact shift matching with don't cares

Theorem 3. *The problem of exact shift matching with don't cares can be solved in $O(n \log m)$ time.*

Exact Shift-Scale Matching with Don't Cares

The pattern p is said to *occur* at location i in t with shift α and scale β if, for every position j in the pattern, either $\alpha + \beta p_j = t_{i+j}$ or at least one of p_j and t_{i+j} is the wild card symbol.

```
Input: Pattern $p$ and text $t$
Output: $B_i = 0$ iff $p$ has a shift occurrence at location $i$ in $t$
$B \leftarrow$ Algorithm 2 $(p, t)$
```

Algorithm 4. Algorithm for exact shift-scale matching with don't cares

Theorem 4. *The problem of exact shift-scale matching with don't cares can be solved in $O(n \log m)$ time.*

Proof. Consider Algorithm 4 and note that Algorithm 2 runs in $O(n \log m)$ time and provides the minimised distance at each location under shift-scale normalisation. This is zero if and only if there is an exact match with a suitable shift and scaling.

4 Discussion

We have shown how to use the FFT to obtain $O(n \log m)$ time solutions to a number of problems involving strings with numerical and wild card/don't care symbols. We have focussed on one-dimensional self-normalisation problems with don't cares in which linear distortions of the pattern are compared with the text in terms of the L_2 distance. The technology can be readily extended to more general polynomial distortions and higher dimensional problems can be tackled in the same manner at the cost of notational complexity. It is an open question whether self-normalisation with don't cares can be applied to other distance measures without increasing the time complexity of their respective search algorithms.

Acknowledgements

The authors would like to thank an anonymous referee in a previous conference for helpful comments on the use of cross-correlations for matching in different metrics.

References

[1] Abrahamson, K.: Generalized string matching. SIAM journal on Computing 16(6), 1039–1051 (1987)
[2] Amir, A., Farach, M.: Efficient 2-dimensional approximate matching of half-rectangular figures. Information and Computation 118(1), 1–11 (1995)
[3] Amir, A., Lipsky, O., Porat, E., Umanski, J.: Approximate matching in the L_1 metric. In: Apostolico, A., Crochemore, M., Park, K. (eds.) CPM 2005. LNCS, vol. 3537, pp. 91–103. Springer, Heidelberg (2005)
[4] Atallah, M.J.: Faster image template matching in the sum of the absolute value of differences measure. IEEE Transactions on Image Processing 10(4), 659–663 (2001)

[5] Clifford, P., Clifford, R.: Simple deterministic wildcard matching. Information Processing Letters 101(2), 53–54 (2007)

[6] Clifford, P., Clifford, R., Iliopoulos, C.S.: Faster algorithms for δ,γ-matching and related problems. In: Apostolico, A., Crochemore, M., Park, K. (eds.) CPM 2005. LNCS, vol. 3537, pp. 68–78. Springer, Heidelberg (2005)

[7] Clifford, R., Iliopoulos, C.: String algorithms in music analysis. Soft Computing 8(9), 597–603 (2004)

[8] Cole, R., Hariharan, R.: Verifying candidate matches in sparse and wildcard matching. In: Proceedings of the Annual ACM Symposium on Theory of Computing, pp. 592–601 (2002)

[9] Cormen, T.H., Leiserson, C.E., Rivest, R.L.: Introduction to Algorithms. MIT Press, Cambridge (1990)

[10] Fischer, M., Paterson, M.: String matching and other products. In: Karp, R. (ed.) Proceedings of the 7th SIAM-AMS Complexity of Computation, pp. 113–125 (1974)

[11] Indyk, P.: Faster algorithms for string matching problems: Matching the convolution bound. In: Proceedings of the 38th Annual Symposium on Foundations of Computer Science, pp. 166–173 (1998)

[12] Jain, R., Kasturi, R., Schunck, B.G.: Machine Vision. McGraw-Hill, New York (1995)

[13] Kalai, A.: Efficient pattern-matching with don't cares. In: Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 655–656, Philadelphia, PA, USA (2002)

[14] Kosaraju, S.R.: Efficient string matching. Manuscript (1987)

[15] Lewis, J.P.: Fast template matching. In: Vision Interface, pp. 120–123 (1995)

[16] Lipsky, O., Porat, E.: Approximate matching in the l_∞ metric. In: Consens, M.P., Navarro, G. (eds.) SPIRE 2005. LNCS, vol. 3772, pp. 91–103. Springer, Heidelberg (2005)

[17] Mäkinen, V., Navarro, G., Ukkonen, E.: Transposition invariant string matching. Journal of Algorithms 56(2), 124–153 (2005)

Move-to-Front, Distance Coding, and Inversion Frequencies Revisited*

Travis Gagie and Giovanni Manzini

Dipartimento di Informatica, Università del Piemonte Orientale,
I-15100 Alessandria, Italy
{travis,manzini}@mfn.unipmn.it

Abstract. Move-to-Front, Distance Coding and Inversion Frequencies are three somewhat related techniques used to process the output of the Burrows-Wheeler Transform. In this paper we analyze these techniques from the point of view of how effective they are in the task of compressing low-entropy strings, that is, strings which have many regularities and are therefore highly compressible. This is a non-trivial task since many compressors have non-constant overheads that become non-negligible when the input string is highly compressible.

Because of the properties of the Burrows-Wheeler transform, being *locally optimal* ensures an algorithm compresses low-entropy strings effectively. Informally, local optimality implies that an algorithm is able to effectively compress *an arbitrary partition* of the input string. We show that in their original formulation neither Move-to-Front, nor Distance Coding, nor Inversion Frequencies is locally optimal. Then, we describe simple variants of the above algorithms which are locally optimal. To achieve local optimality with Move-to-Front it suffices to combine it with Run Length Encoding. To achieve local optimality with Distance Coding and Inversion Frequencies we use a novel "escape and re-enter" strategy. Since we build on previous results, our analyses are simple and shed new light on the inner workings of the three techniques considered in this paper.

1 Introduction

Burrows-Wheeler compression is important in itself and as a key component of compressed full-text indices [15]. It is therefore not surprising that the theory and practice of Burrows-Wheeler compression has recently received much attention [5,6,7,9,10,11,12,14,13].

In the original Burrows-Wheeler compression algorithm [4] the output of the Burrows-Wheeler Transform (bwt from now on) is processed by Move-to-Front encoding followed by an order-0 encoder. Recently, [12] has provided a simple and elegant analysis of this algorithm and of the variant in which Move-to-Front encoding is replaced by Distance Coding. This analysis improves the previous

* Partially supported by Italian MIUR Italy-Israel FIRB Project "Pattern Discovery Algorithms in Discrete Structures, with Applications to Bioinformatics".

one in [14] and provides new insight on the Move-to-Front and Distance Coding procedures. In [12] the output size of Burrows-Wheeler algorithms on input s is bounded by $\mu|s|\,H_k(s) + \Theta(|s|)$ for any $\mu > 1$, where H_k is the k-th order empirical entropy (more details in Sect. 2). We point out that this is a significant bound only as long as the string s is not too compressible. For highly compressible strings, for example $s = ab^n$, we have $|s|\,H_k(s) = O(\log|s|)$ so the $\Theta(|s|)$ term in the above bound becomes the dominant term. In this case, the bound tells one nothing about how close the compression ratio is to the entropy of the input string.

The above considerations suggest that further work is required on the problem of designing algorithms with an output size bounded by $\lambda|s|\,H_k(s)+\Theta(1)$ where $\lambda > 1$ is a constant independent of k, $|s|$, and of the alphabet size. We call this an "entropy-only" bound. An algorithm achieving an "entropy-only" bound guarantees that even for highly compressible strings the compression ratio will be proportional to the entropy of the input string. Note that the capability of achieving "entropy-only" bounds is one of the features that differentiate Burrows-Wheeler compression algorithms from the family of Lempel-Ziv compressors [14].

As observed in [14], the key property for achieving "entropy-only" bounds is that the compression algorithm used to process the bwt be *locally optimal.* Informally, a locally optimal algorithm has the property of being able to compress efficiently *an arbitrary partition* of the input string (see Definition 2). Starting from these premises, in this paper we prove the following results:

1. We analyze Move-to-Front (Mtf), Distance Coding (Dc), and Inversion Frequencies Coding (If)—which is another popular variant of Move-to-Front encoding. We prove that in their original formulation none of these algorithms is locally optimal.
2. We describe simple variants of the above algorithms which are locally optimal. Therefore, when used together with the bwt these variants achieve "entropy-only" bounds. To achieve local optimality with Mtf it suffices to combine it with Run Length Encoding (Rle from now on). To achieve local optimality with If and Dc we use an "escape and re-enter" technique.
3. The procedures Mtf, If, and Dc all output sequences of positive integers. One can encode these sequences either using a prefix-free encoding [6] or feed the whole sequence to an Order-0 encoder [4]. Taking advantage of previous results of Burrows-Wheeler compression [12,13], we are able to provide a simple analysis for both options.

In terms of "entropy-only" bounds our best result is the bound $(2.707 + \eta_0)|s|\,H_k^*(s)+\Theta(1)$ bits where η_0 is a constant characteristic of the Order-0 final encoder (for example $\eta_0 \approx .01$ for arithmetic coding). This bound is achieved by our variant of Distance Coding and it improves the previously known "entropy-only" bound of $(5+3\eta_0)|s|\,H_k^*(s)+\Theta(1)$ established in [14] for Mtf + Rle.

2 Notation and Background

Let s be a string drawn from the alphabet $\Sigma = \{\sigma_1,\ldots,\sigma_h\}$. For $i = 1,\ldots,|s|$ we write $s[i]$ to denote the i-th character of s. For each $\sigma_i \in \Sigma$, let n_i be the number of occurrences of σ_i in s. The 0-*th order empirical entropy* of the string s is defined as[1] $H_0(s) = -\sum_{i=1}^{h}(n_i/|s|)\log(n_i/|s|)$. It is well known that H_0 is the maximum compression we can achieve using a fixed codeword for each alphabet symbol.

Definition 1. *An algorithm* A *is an Order-0 algorithm if for any input string* s *we have*

$$|\mathsf{A}(s)| \le |s|H_0(s) + \eta_0|s|.$$

Examples of Order-0 algorithms are Huffman coding, for which $\eta_0 = 1$, and Arithmetic coding, for which $\eta_0 \approx .01$. It is well known that we can often achieve a compression ratio better than $|s|H_0(s)$ if the codeword we use for each symbol depends on the k symbols preceding it. In this case, the maximum compression is bounded by the k-th order entropy $H_k(s)$ (see [14] for the formal definition).

In [12] the authors analyze the original Burrows-Wheeler compressor in which the output of the bwt is processed by Mtf followed by an Order-0 compressor and they prove a bound of the form

$$\mu|s|\,H_k(s) + (\log(\zeta(\mu)) + \eta_0)|s| + O(1) \tag{1}$$

where ζ is the Riemann zeta function and η_0 is the constant associated with the Order-0 algorithm (see Definition 1). The above bound holds simultaneously for any $\mu > 1$ and $k \ge 0$. This means we can get arbitrarily close to the k-th order entropy for any $k \ge 0$. Unfortunately, in (1) there is also a $\Theta(|s|)$ term which becomes dominant when s is highly compressible. For example, for $s = ab^n$ we have $|s|H_0(s) = \log|s|$. In this case, the bound (1) does not guarantee that the compression ratio is close to the entropy.

We are interested, therefore, in proving "entropy-only" bounds of the form $\lambda|s|\,H_k(s) + \Theta(1)$. Unfortunately, such bounds cannot be established. To see this, consider the family of strings $s = a^n$; we have $|s|H_0(s) = 0$ and we cannot hope to compress all strings in this family in $\Theta(1)$ space. For that reason, [14] introduced the notion of 0-*th order modified empirical* entropy:

$$H_0^*(s) = \begin{cases} 0 & \text{if } |s| = 0 \\ (1 + \lfloor \log|s| \rfloor)/|s| & \text{if } |s| \ne 0 \text{ and } H_0(s) = 0 \\ H_0(s) & \text{otherwise.} \end{cases} \tag{2}$$

Note that if $|s| > 0$, $|s|\,H_0^*(s)$ is at least equal to the number of bits needed to write down the length of s in binary. The *k-th order modified empirical entropy* H_k^* is then defined in terms of H_0^* as the maximum compression we can achieve by looking at *no more than* k symbols preceding the one to be compressed. With a rather complex analysis [14] proves the bound $(5 + 3\eta_0)|s|\,H_k^*(s) + \Theta(1)$ for the output size of an algorithm described in [4].

This paper is the ideal continuation of [12] and [14]. We analyze the classical Move-to-Front encoding [2], and two popular (and effective) alternatives: Distance Coding [3,5], and Inversion Frequencies Coding [1]. We provide simple

[1] We assume that all logarithms are taken to the base 2 and $0 \log 0 = 0$.

proofs that simple variants of these algorithms achieve "entropy-only" bounds of the form $\lambda|s|\, H_k^*(s) + \Theta(1)$. The key tool for establishing these bounds is the notion of local optimality introduced in [14].

Definition 2. *A compression algorithm* A *is* locally λ-optimal *if there exists a constant* c_h *such that for any string* s *and for any partition* $s_1 s_2 \cdots s_t$ *of* s *we have*

$$\mathsf{A}(s) \leq \lambda \left[\sum_{i=1}^{t} |s_i|\, H_0^*(s_i) \right] + \ c_h t,$$

where the constant c_h *depends only on the alphabet size* h*.*

The importance of the notion of local optimality stems from the following lemma which establishes that processing the output of the bwt with a locally optimal algorithm yields an algorithm achieving an "entropy-only" bound.

Lemma 1 ([14]). *If* A *is locally* λ*-optimal then the bound*

$$\mathsf{A}(\mathsf{bwt}(s)) \leq \lambda|s|\, H_k^*(s) + c_h h^k$$

holds simultaneously for any $k \geq 0$*.* □

Useful lemmas. Move-to-Front, Distance Coding, and Inversion Frequencies all output sequences of positive integers. In the following we assume that Pfx is a prefix-free encoder of the integers such that for any integer i we have $|\mathsf{Pfx}(i)| \leq a \log i + b$ where a and b are positive constants. Note that since $|\mathsf{Pfx}(1)| \leq b$ we must have $b \geq 1$. Also note that for γ codes we have $a = 2$ and $b = 1$. This means that it is only worth investigating prefix codes with $a \leq 2$. Indeed, a code with $a > 2$ (and necessarily $b \geq 1$) would be worse than γ codes and therefore not interesting. Hence, in the following we assume $a \leq 2$, $b \geq 1$ and thus $a \leq 2b$ and $a \leq b+1$. Under these assumptions on a and b we have the following lemma whose simple proof will be given in the full paper.

Lemma 2 (Subadditivity). *Let* $x_1, x_2, \ldots, x_k$ *be positive integers. We have*

$$|\mathsf{Pfx}(x_1 + x_2)| \leq |\mathsf{Pfx}(x_1)| + |\mathsf{Pfx}(x_2)| + 1,$$

and

$$\left|\mathsf{Pfx}\Big(\sum\nolimits_{i=1}^{k} x_k\Big)\right| \ \leq \ \Big(\sum\nolimits_{i=1}^{k} |\mathsf{Pfx}(x_i)|\Big) + 2.$$

□

The next lemma, which follows from the analysis in [12], establishes that if we feed a sequence of integers to an Order-0 encoder (see Definition 1) the output is no larger than the output produced by a prefix-free encoder with parameters $a = \mu$ and $b = \log(\zeta(\mu)) + \eta_0$ for any $\mu > 1$.

Lemma 3 ([12]). *Let* A *be an order zero encoder with parameter* η_0*. For any sequence of positive integers* $i_1 i_2 \cdots i_n$ *and for any* $\mu > 1$ *we have*

$$|\mathsf{A}(i_1 i_2 \cdots i_n)| \leq \sum\nolimits_{i=1}^{n} \big(\mu \log(i) + \log \zeta(\mu) + \eta_0\big).$$ □

Given a string $s = s_1 s_2 \cdots s_n$, a run is a substring $s_i s_{i+1} \cdots s_{i+k}$ of identical symbols, and a maximal run is a run which cannot be extended; that is, it is not a proper substring of a larger run.

Lemma 4 ([13, Sect. 3]). *The number of maximal runs in a string s is bounded by $1 + |s| H_0(s)$.* □

3 Local Optimality with Move-to-Front Encoding

Move-to-Front (Mtf) is a well known technique proposed independently in [2,16]. Mtf encodes a string by replacing each symbol with the number of distinct symbols seen since its last occurrence. To this end, Mtf 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. If the input string is defined over the alphabet Σ, in the following we assume that ranks are in the range $[1, h]$, where $h = |\Sigma|$. For our analysis we initially assume that the output of Mtf is encoded by the prefix-free encoder Pfx, such that $|\mathsf{Pfx}(i)| \leq a \log i + b$. Note that to completely determine the encoding procedure we must specify the initial status of the recency list. However, changing the initial status increases the output size by at most $O(h \log h)$ so we will add this overhead and ignore the issue. The following simple example shows that $\mathsf{Mtf} + \mathsf{Pfx}$ is not locally optimal.

Example 1. Consider the string $s = \sigma^n$ and the partition consisting of the single element s. We have $\mathsf{Mtf}(s) = 1^n$ and $|\mathsf{Pfx}(\mathsf{Mtf}(s))| = |s|b$. Since $|s|\ H_0^*(s) = 1 + \lfloor \log |s| \rfloor$ it follows that $\mathsf{Mtf} + \mathsf{Pfx}$ is not locally optimal. □

Note that a similar result holds even if we replace Pfx with an Order-0 encoder (see Definition 1). In that case the output size is at least $\eta_0 |s|$. These examples clearly show that if we feed to the final encoder $\Theta(|s|)$ "items" there is no hope of achieving "entropy-only" bounds.

The above considerations suggest the algorithm Mtf_rl, which combines Mtf with Run Length Encoding (Rle). Assume that $\sigma = s[i+1]$ is the next symbol to be encoded. Instead of simply encoding the Mtf rank r of σ, Mtf_rl finds the maximal run $s[i+1] \cdots s[i+k]$ of consecutive occurrences of σ, and encodes the pair (r, k). We define the algorithm $\mathsf{Mtf\_rl} + \mathsf{Pfx}$ as the algorithm which encodes each such pair with the codeword $\mathsf{Pfx}(r-1)$ followed by $\mathsf{Pfx}(k)$ (this is possible since we cannot have $r = 1$).

Lemma 5. *Let* $\mathsf{A}_1 = \mathsf{Mtf\_rl} + \mathsf{Pfx}$. *For any string s we have*

$$|\mathsf{A}_1(s)| \leq (a + 2b)|s|\ H_0^*(s) + O(h \log h) .$$

Proof. Assume $H_0(s) \neq 0$ (otherwise we have $s = \sigma^n$ and the proof follows by an easy computation). Let $C(i) = a \log i + b$. For each $\sigma \in \Sigma$ let P_σ denote the set of pairs (r_i, k_i) generated by the encoding of runs of σ. We have

$$|\mathsf{A}_1(s)| \leq \sum_{\sigma \in \Sigma} \left[\sum_{p_i \in P_\sigma} \big(C(r_i - 1) + C(k_i) \big) \right] + O(h \log h) .$$

Note that $\sum_\sigma |P_\sigma|$ is equal to the number of maximal runs in s. By Lemma 4 we have

$$|\mathsf{A}_1(s)| \le \sum_{\sigma \in \Sigma} a \left[\sum_{p_i \in P_\sigma} \log(r_i - 1) + \log(k_i) \right] + 2b|s|H_0(s) + O(h \log h) \,.$$

For any given σ the total number of non-zero terms in the summations within square brackets is bounded by the number n_σ of occurrences of σ in s. Since the sum of the log's argument is bounded by $|s|$, by Jensen's inequality the content of the square brackets is bounded by $n_\sigma \log(n/n_\sigma)$. Hence, the outer summation is bounded by $a|s|H_0(s)$. Since $H_0(s) \le H_0^*(s)$ the lemma is proven. □

Theorem 1. *The algorithm* $\mathsf{A}_1 = \mathsf{Mtf\_rl} + \mathsf{Pfx}$ *is locally* $(a + 2b)$*-optimal.*

Proof. By Lemma 5 it suffices to prove that

$$|\mathsf{A}_1(s_1 s_2)| \le |\mathsf{A}_1(s_1)| + |\mathsf{A}_1(s_2)| + O(h \log h).$$

To prove this inequality observe that compressing s_2 independently of s_1 changes the encoding of only the first occurrence of each symbol in s_2. This gives a $O(h \log h)$ overhead. In addition, if a run crosses the boundary between s_1 and s_2 the first part of the run will be encoded in s_1 and the second part in s_2. By Lemma 4 this produces an $O(1)$ overhead and the theorem follows. □

Theorem 2. *The algorithm* $\mathsf{Mtf\_rl} + \mathsf{Order0}$ *is locally* $(\mu + 2\log(\eta(\mu)) + \eta_0)$*-optimal for any* $\mu > 1$. *By taking* $\mu = 2.35$ *we get that* $\mathsf{Mtf\_rl} + \mathsf{Order0}$ *is locally* $(3.335 + \eta_0)$*-optimal. By Lemma 1 we conclude that for any string* s *and* $k \ge 0$

$$|\mathsf{Order0}(\mathsf{Mtf\_rl}(\mathsf{bwt}(s)))| \le (3.335 + \eta_0)|s|\, H_k^*(s) + O\big(h^{k+1} \log h\big) \,. \qquad \square$$

4 Local Optimality with Distance Coding

Distance Coding (Dc) is a variant of Mtf which is relatively little-known, probably because it was originally described only on a Usenet post [3]. Recently, [12] has proven that, combining the Burrows-Wheeler transform with Dc and an Order-0 encoder (see Def. 1), we get an output bounded by $1.7286|s|\, H_k(s) + \eta_0|s| + O(\log |s|)$. Using Lemma 4, the analysis in [12] can be easily refined to get the bound $(1.7286 + \eta_0)|s|\, H_k(s) + O(\log |s|)$. Note that this is not an "entropy-only" bound because of the presence of the $O(\log |s|)$ term.

To encode a string s over the alphabet $\Sigma = \{\sigma_1, \ldots, \sigma_h\}$ using distance coding:

1. we write the first character in s;
2. for each other character $\sigma \in \Sigma$, we write the distance to the first σ in s, or a 1 if σ does not occur (notice no distance is 1, because we do not reconsider the first character in s);
3. for each maximal run of a character σ, we write the distance from the start of that run to the start of the next maximal run of σ's, or 1 if there are no more σ's (again, notice no distance is 1);

4. finally, we encode the length ℓ of the last run in s as follows: if $\ell = 1$ we write 1, otherwise we write 2 followed by $\ell - 1$.

In other words, in Dc we encode the distance between the starting points of consecutive maximal runs of any given character $\sigma \in \Sigma$. It is not hard to prove Dc correct, by induction on the number of runs in s. In the following we define $\mathsf{Dc} + \mathsf{Pfx}$ as the algorithm in which the integers produced by Dc are encoded using Pfx.

Lemma 6. *Let* $\mathsf{A}_2 = \mathsf{Dc} + \mathsf{Pfx}$. *For any string s we have*

$$|\mathsf{A}_2(s)| \leq (a+b)|s|\, H_0^*(s) + O(h).$$

Proof. (Sketch) Assume $H_0(s) \neq 0$ (otherwise we have $s = \sigma^n$ and the proof follows by an easy computation). Writing the first character in s takes $O(\log h)$ bits; we write h copies of 1 while encoding s (or $h+1$ if the first character is a 1), which takes $O(h)$ bits. We are left with the task of bounding the cost of encoding: 1) the starting position of the first run of each character, 2) distances between runs and 3) the length of the last run. Reasoning as in Lemma 5 we can prove that the cost associated with the runs of each character σ is bounded by $a\, n_\sigma \log(|s|/n_\sigma) + br_\sigma + a + b$ bits, where n_σ is the number of occurrences of σ and r_σ is the number of (maximal) runs of σ (the only non-trivial case is the symbol which has the last run in s). Summing over all σ's and using Lemma 4 yields the thesis. □

The following example shows that $\mathsf{Dc} + \mathsf{Pfx}$ is not locally optimal.

Example 2. Consider the partition $s = s_1 s_2 s_3$ where

$$s_1 = s_3 = \sigma_1 \sigma_2 \cdots \sigma_h, \qquad , s_2 = \sigma_2^n.$$

We have $\sum_{i=1}^{2h} |s_i|\, H_k^*(s_i) = \log n + O(h \log h)$, whereas $\mathsf{Dc} + \mathsf{Pfx}$ produces an output of size $\Theta(h \log n)$. To see this, observe that for each character it has to write a distance greater than n. □

The above example suggests that to achieve local optimality with Dc we should try to avoid the encoding of "long jumps". To this end, we introduce Distance Coding with escapes ($\mathsf{Dc\_esc}$). The main difference between Dc and $\mathsf{Dc\_esc}$ is that, whenever Dc would write a distance, $\mathsf{Dc\_esc}$ compares the cost of writing that distance to the cost of escaping and re-entering later, and does whichever is cheaper.

Whenever Dc would write 1, $\mathsf{Dc\_esc}$ writes $\langle 1, 1\rangle$; this lets us use $\langle 1, 2\rangle$ as a special re-entry sequence. To escape after a run of σ's, we write $\langle 1, 1\rangle$; to re-enter at the the next run of σ's, we write $\langle 1, 2, \ell, \sigma\rangle$, where ℓ is the length of the preceding run (of some other character). To see how $\mathsf{Dc\_esc}$ works suppose we are encoding the string

$$s = \cdots \sigma_1^r\, \sigma_2^s\, \sigma_3^t\, \sigma_1^u \cdots.$$

When Dc reaches the run σ_1^r it encodes the value $r+s+t$ which is the distance to the next run of σ_1's. Instead, Dc_esc compares the cost of encoding $r+s+t$ with the cost of encoding an escape (sequence $\langle 1,1 \rangle$) plus the cost of re-entering. In this case the re-entry sequence would be written immediately after the code associated with the run σ_3^t and would consist of the sequence $\langle 1,2,t,\sigma_1 \rangle$. When the decoder finds such sequence it knows that the current run (in this case of σ_3's) will only last for t characters and, after that, there is a run of σ_1's. (Recall that Dc only encodes the starting position of each run: the end of the run is usually induced by the beginning of a new run. When we re-enter an escaped character we have to provide explicitly the length of the ongoing run).

Notice we do not distinguish between instances in which $\langle 1,1 \rangle$ indicates a character does not occur, cases in which it indicates a character does not occur again, and cases in which it indicates an escape; we view the first two types of cases as escapes without matching re-entries.

Lemma 7. *Let* $\mathsf{A}_2 = \mathsf{Dc} + \mathsf{Pfx}$ *and let* $\mathsf{A}_3 = \mathsf{Dc\_esc} + \mathsf{Pfx}$*. For any string* s *and for any partition* $s = s_1, \ldots, s_t$

$$|\mathsf{A}_3(s)| \leq \sum_{i=1}^{t} |\mathsf{A}_2(s_i)| + O(ht \log h) \ .$$

Proof. (Sketch) We consider the algorithm Dc_esc* that, instead of choosing at each step whether to escape or not, uses the following strategy: use the escape sequence if and only if we are encoding the distance between two runs whose starting points belong to different partition elements (say s_i and s_j with $j > i$). Let $\mathsf{A}_3' = \mathsf{Dc\_esc^*} + \mathsf{Pfx}$. Since Dc_esc always performs the most economical choice, we have $|\mathsf{A}_3(s)| \leq |\mathsf{A}_3'(s)|$; we prove the lemma by showing that

$$|\mathsf{A}_3'(s)| \leq \sum_{i=1}^{t} |\mathsf{A}_2(s_i)| + O(ht \log h).$$

Clearly Dc_esc* escapes at most th times. The parts of an escape/re-enter sequence that cost $\Theta(1)$ (that is, the codewords for $\langle 1,1 \rangle$, $\langle 1,2 \rangle$ and the encoding of the escaped character σ) are therefore included in the $O(ht \log h)$ term. Thus, we only have to take care of the encoding of the value ℓ which encodes the length of the run immediately preceding the re-entry point. We now show that the cost of encoding the run lengths ℓs is bounded by costs paid by Dc and not paid by Dc_esc*. Let σ denote the escaped character. Let s_j denote the partition element containing the re-entry point and let m denote the position in s_j where the new run of σ's starts (that is, at position m of s_j there starts a run of σ's; the previous one was in some s_i with $i < j$ so Dc_esc* escaped σ and is now re-entering). We consider two cases:

$\ell \leq m$. In this case we observe that the cost $|\mathsf{Pfx}(\ell)|$ paid by Dc_esc* is no greater than the cost $\mathsf{Pfx}(m)$ paid by Dc for encoding the first position of σ in s_j.

$\ell > m$. Let $m' = \ell - m$ and assume $m' < |s_{j-1}|$. In this case we observe that by Lemma 4 the cost $|\mathsf{Pfx}(\ell)|$ paid by Dc_esc* is at most 1 plus the cost $|\mathsf{Pfx}(m)|$

paid by Dc for encoding the first position of σ in s_j, plus the cost $|\mathsf{Pfx}(m')|$ paid by Dc to encode the length of the last run in s_{j-1}. If $m' > |s_{j-1}|$ then there is a run spanning s_{j-1}, s_{j-2} and so on, and the thesis follows by the first part of Lemma 4. □

Combining the above lemma with Lemma 6 we immediately get

Theorem 3. *The algorithm* $\mathsf{A}_3 = \mathsf{Dc\_esc} + \mathsf{Pfx}$ *is locally* $(a+b)$*-optimal.* □

Theorem 4. *The algorithm* $\mathsf{Dc\_esc} + \mathsf{Order0}$ *is locally* $(\mu + \log(\eta(\mu)) + \eta_0)$*-optimal for any* $\mu > 1$*. By taking* $\mu = 1.88$ *we get that* $\mathsf{Dc\_esc} + \mathsf{Order0}$ *is locally* $(2.707 + \eta_0)$*-optimal. By Lemma 1 we conclude that for any string* s *and* $k \geq 0$

$$|\mathsf{Order0}(\mathsf{Dc\_esc}(\mathsf{bwt}(s)))| \leq (2.707 + \eta_0)|s|\, H_k^*(s) + O\big(h^{k+1}\log h\big)\,. \qquad \square$$

5 Local Optimality with Inversion Frequencies Coding

Inversion Frequencies (If for short) is a coding strategy proposed in [1] as an alternative to Mtf. Given a string s over an ordered alphabet $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_h\}$, in its original formulation If works in $h-1$ phases. In the i-th phase we encode all the occurrences of the symbol σ_i as follows: first If outputs the position in s of the first occurrence of σ_i, then it outputs the number of symbols *greater than* σ_i between two consecutive occurrences of σ_i. Note that there is no need to encode the occurrences of σ_h. The output of If consists of the concatenation of the output of the single phases prefixed by an appropriate encoding of the number of occurrences of each symbol σ_i (this information is needed by the decoder to figure out when a phase is complete). For example, if $s = \sigma_2\sigma_2\sigma_1\sigma_3\sigma_3\sigma_1\sigma_3\sigma_1\sigma_3\sigma_2$, the first phase encodes the occurrences of σ_1, producing the sequence $\langle 3, 3, 2\rangle$; the second phase encodes the occurrences of σ_2, producing the sequence $\langle 1, 1, 5\rangle$. The output of If is therefore an encoding of the number of occurrences of σ_1, σ_2, and σ_3 (3, 3, and 4 in our example), followed by the sequence $\langle 3, 3, 2, 1, 1, 5\rangle$.

Note that, differently from Mtf, If usually outputs integers larger than the alphabet size. However, the encoding of the symbols becomes more and more economical, up to the last symbol in the alphabet which needs no encoding at all. Recently, in [8], the authors showed that If coding is equivalent to coding with a skewed wavelet tree, and have theoretically justified leaving the most frequent symbol for last (so that it gets encoded for free). Unfortunately, the following example shows that If is not locally optimal.

Example 3. Consider the partition $s = s_1 s_2 \cdots s_{2h}$ where

$$s_1 = \sigma_1\sigma_2\cdots\sigma_h \quad s_2 = \sigma_1^n \quad s_3 = s_1 \quad s_4 = \sigma_2^n \quad s_5 = s_1 \quad s_6 = \sigma_3^n$$

and so on up to $s_{2h} = \sigma_h^n$. We have $\sum_{i=1}^{2h} |s_i|\, H_k^*(s_i) = O(h \log n)$, whereas, no matter how we order the alphabet $\mathsf{Pfx}(\mathsf{If}(s)) = O\big(h^2 \log n\big)$. □

We now describe two variants of the basic If procedure and we prove that the second variant is $(2a+b+0.5)$-locally optimal. The first variant, called Forward Inversion Frequencies ($\mathsf{If\_rl}$), will serve only as an intermediate step but we believe it is an interesting procedure in itself.

For $i = 1, \ldots, h-1$, $\mathsf{If\_rl}$ first outputs the integer n_i equal to the number of symbols greater than σ_i that can be found from the beginning of s up to the first occurrence of σ_i in s. Then, for $i = 1, \ldots, h-1$, $\mathsf{If\_rl}$ encodes the positions of the symbol σ_i "ignoring" the occurrences of symbols smaller than σ_i. However, differently from If, $\mathsf{If\_rl}$ works with all the alphabet symbols simultaneously. This is how it is done. Assume that $s[j]$ is the next symbol to be encoded and let $s[j] = \sigma_i$. Let $s[\ell]$ be the first occurrence of a symbol greater than σ_i to the right of $s[j]$, and let $s[m]$ be the first occurrence of the symbol σ_i to the right of $s[\ell]$. The procedure $\mathsf{If\_rl}$ encodes the number k of occurrences of σ_i in $s[j] \cdots s[\ell-1]$ and the number t of occurrences of symbols greater than σ_i in $s[\ell] \cdots s[m-1]$. Then, $\mathsf{If\_rl}$ moves right in s up to the first symbol different from σ_i (and σ_h).

Note that $\mathsf{If\_rl}$ is essentially encoding the following information: "starting from $s[j]$ there are k occurrences of σ_i before we reach the first symbol greater than σ_i; after that there are t symbols greater than σ_i before we find another occurrence of σ_i". If $s[\ell]$ does not exist (there are no symbols greater than σ_i to the right of $s[j]$) or $s[m]$ does not exist (there are no occurrences of σ_i to the right of $s[\ell]$), then $\mathsf{If\_rl}$ encodes the value $t = 0$, which indicates that there are no further occurrences of σ_i. Note that the above procedure encodes no information about the last symbol σ_h. However, as a final step, it is necessary to encode whether s terminates with a run of σ_h and the length of such a run.

It is not difficult to prove that from the output of $\mathsf{If\_rl}$ we can retrieve s. For each symbol σ, $\sigma \neq \sigma_h$, the decoder maintains two variables *To_be_written* and *To_be_skipped*. The former indicates how many σ's have to be written before we find a character greater than σ; the latter indicates how many characters greater than σ there are between the current run of σ's and the next one. The decoder works in a loop. At each iteration it outputs the smallest symbol σ_i such that its associated variable *To_be_written* is nonzero; if all *To_be_written* variables are zero the decoder outputs σ_h. After outputting σ_i, the decoder decreases by one the variable *To_be_skipped* associated with all symbols smaller than σ_i. If one or more variables *To_be_skipped* reaches zero, the decoder reads a pair $\langle k, t \rangle$ from the compressed file and sets *To_be_written* $\leftarrow k$ and *To_be_skipped* $\leftarrow t$ for the pair of variables associated with the smallest symbol with *To_be_skipped* $= 0$.

We define $\mathsf{If\_rl} + \mathsf{Pfx}$ as the algorithm which encodes the pairs $\langle k, t \rangle$ as follows (for the $h-1$ initial values we use the same encoding as for the parameter t):

1. the value k is encoded with $\mathsf{Pfx}(k)$;
2. a value $t = 0$ is encoded with with a single 0 bit, a value $t > 0$ is encoded with a bit 1 followed by $\mathsf{Pfx}(t)$.

As a final step, $\mathsf{If\_rl} + \mathsf{Pfx}$ outputs a single 0 bit if s does not end with σ_h, and outputs a bit 1 followed by $\mathsf{Pfx}(\ell)$ if s ends with a run of σ_h of length $\ell \geq 1$.

Lemma 8. *Let* $\mathsf{A}_4 = \mathsf{If\_rl} + \mathsf{Pfx}$. *For any string* s *we have*

$$|\mathsf{A}_4(s)| \leq (2a + b + 0.5)|s|\, H_0^*(s) + O(1).$$

Proof. (Sketch) We reason as in the proof of Lemma 5. Each pair $\langle k, t\rangle$ has a cost $a(\log k + \log t) + 2b + 1$ if $t > 0$ or $a \log k + 2b + 1$ if $t = 0$. Refining the proof of Lemma 4, we can show that the total number of pairs is bounded by $h + |s|H_0(s)/2$. Hence, the overall contribution of the $2b + 1$ term is $(b + 0.5)|s|H_0(s) + O(h)$. To bound the contribution of the logs we use again Jensen's inequality and prove that their overall contribution is bounded by $a|s|H_0(s)$. Finally, the cost of encoding the length of the last run of σ_h is bounded by $a \log |s| + b$ which is less than $a|s|\, H_0^*(s) + b$ and the lemma follows. □

Unfortunately, we can repeat verbatim Example 3 to show that If_rl is not locally optimal. We observe that an obvious inefficiency of If and If_rl is that sometimes they are forced to pay the cost of a "long jump" many times. Consider the following example:

$$s = \sigma_1\sigma_2\sigma_3^n\sigma_2\sigma_1.$$

Assuming $\sigma_1 < \sigma_2 < \sigma_3$ we see that If_rl and If pay a $O(\log n)$ cost in the encoding of both σ_1 and σ_2 because of the presence of the σ_3^n substring. We now propose an *escape and re-enter* mechanism that essentially guarantees that in the above situation we pay the $O(\log n)$ cost at most once.

The new algorithm, called Inversion Frequencies with Escapes (If_esc), works as follows. Assume that $s[j] = \sigma_i$ is the next character to be encoded, and let $s[\ell]$, $s[m]$, k, and t be defined as for the algorithm If_rl. Moreover, let p denote the largest index such that $\ell < p \leq m$ and $s[p-1] > s[p]$ (p does not necessarily exist). If p does not exist, If_esc behaves as If_rl and outputs the pair $\langle k, t\rangle$. If p exists, If_esc chooses the more economical option between 1) encoding $\langle k, t\rangle$ and 2) escaping σ_i (which means encoding the pair $\langle k, 1\rangle$) and re-entering it at the position p (it is possible to re-enter at p since the condition $s[p-1] > s[p]$ implies that when the decoder reaches the position p it will read a new pair from the compressed file). The code for re-entering will contain the character σ_i and the number t' of characters greater than σ_i in $s[p] \cdots s[m-1]$. The crucial points of this escape/re-enter strategy are the following:

1. if $s[m-1] < \sigma_i$ then $t' = 0$ (for the choice of p there cannot be a character greater than σ_i in $s[p] \cdots s[m-1]$) so encoding t' takes a constant number of bits;
2. if $s[m-1] > \sigma_i$, then t' can be as large as $m - p$. However, If_esc will not be forced to pay the cost of "jumping" the substring $s[p] \cdots s[m-1]$ for encoding another character σ, since it will have the option of escaping σ before the position p and re-entering it at the position m.

If p does not exist the situation is analogous to case 2: If_esc will have to pay the cost of encoding t, but for any character σ, $\sigma \neq \sigma_i$, it will have the option of re-entering at m, so it will not be forced to pay that cost twice.

In the full paper we will prove the following theorem.

Theorem 5. *The algorithm* If_esc + Pfx *is locally* $(2a + b + 0.5)$*-optimal. In addition, the algorithm* If_esc + Order0 *is locally* $(4.883 + \eta_0)$*-optimal, and for any string* s *and any* $k \geq 0$ *we have*

$$|\mathsf{Order0}(\mathsf{If\_esc}(\mathsf{bwt}(s)))| \leq (4.883+\eta_0)|s|H_k^*(s)+O\big(h^{k+1}\log h\big)\,. \qquad \square$$

References

1. Arnavut, Z., Magliveras, S.: Block sorting and compression. In: Procs of IEEE Data Compression Conference (DCC), pp. 181–190 (1997)
2. Bentley, J., Sleator, D., Tarjan, R., Wei, V.: A locally adaptive data compression scheme. Communications of the ACM 29(4), 320–330 (1986)
3. Binder, E.: Distance coder, Usenet group (2000) comp.compression
4. Burrows, M., Wheeler, D.: A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation (1994)
5. Deorowicz, S.: Second step algorithms in the Burrows-Wheeler compression algorithm. Software: Practice and Experience 32(2), 99–111 (2002)
6. Fenwick, P.: Burrows-Wheeler compression with variable length integer codes. Software: Practice and Experience 32, 1307–1316 (2002)
7. Ferragina, P., Giancarlo, R., Manzini, G.: The engineering of a compression boosting library: Theory vs practice in bwt compression. In: Azar, Y., Erlebach, T. (eds.) ESA 2006. LNCS, vol. 4168, pp. 756–767. Springer, Heidelberg (2006)
8. Ferragina, P., Giancarlo, R., Manzini, G.: The myriad virtues of wavelet trees. In: Bugliesi, M., Preneel, B., Sassone, V., Wegener, I. (eds.) ICALP 2006. LNCS, vol. 4051, pp. 561–572. Springer, Heidelberg (2006)
9. Ferragina, P., Giancarlo, R., Manzini, G., Sciortino, M.: Boosting textual compression in optimal linear time. Journal of the ACM 52, 688–713 (2005)
10. Foschini, L., Grossi, R., Gupta, A., Vitter, J.: Fast compression with a static model in high-order entropy. In: Procs of IEEE Data Compression Conference (DCC), pp. 62–71 (2004)
11. Giancarlo, R., Sciortino, M.: Optimal partitions of strings: A new class of Burrows-Wheeler compression algorithms. In: Baeza-Yates, R.A., Chávez, E., Crochemore, M. (eds.) CPM 2003. LNCS, vol. 2676, pp. 129–143. Springer, Heidelberg (2003)
12. Kaplan, H., Landau, S., Verbin, E.: A simpler analysis of Burrows-Wheeler based compression. In: Lewenstein, M., Valiente, G. (eds.) CPM 2006. LNCS, vol. 4009, Springer, Heidelberg (2006)
13. Mäkinen, V., Navarro, G.: Succinct suffix arrays based on run-length encoding. Nordic Journal of Computing 12(1), 40–66 (2005)
14. Manzini, G.: An analysis of the Burrows-Wheeler transform. Journal of the ACM 48(3), 407–430 (2001)
15. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Computing Surveys. (To Appear)
16. Ryabko, B.Y.: Data compression by means of a 'book stack'. Prob.Inf.Transm, 16(4) (1980)

A Lempel-Ziv Text Index on Secondary Storage*

Diego Arroyuelo and Gonzalo Navarro

Dept. of Computer Science, Universidad de Chile,
Blanco Encalada 2120, Santiago, Chile
{darroyue,gnavarro}@dcc.uchile.cl

Abstract. *Full-text* searching consists in locating the occurrences of a given pattern $P[1..m]$ in a text $T[1..u]$, both sequences over an alphabet of size σ. In this paper we define a new index for full-text searching on *secondary storage*, based on the Lempel-Ziv compression algorithm and requiring $8uH_k + o(u \log \sigma)$ bits of space, where H_k denotes the k-th order empirical entropy of T, for any $k = o(\log_\sigma u)$. Our experimental results show that our index is significantly smaller than any other practical secondary-memory data structure: 1.4–2.3 times the text size *including the text*, which means 39%–65% the size of traditional indexes like *String B-trees* [Ferragina and Grossi, *JACM* 1999]. In exchange, our index requires more disk access to locate the pattern occurrences. Our index is able to report up to 600 occurrences per disk access, for a disk page of 32 kilobytes. If we only need to *count* pattern occurrences, the space can be reduced to about 1.04–1.68 times the text size, requiring about 20–60 disk accesses, depending on the pattern length.

1 Introduction and Previous Work

Many applications require to store huge amounts of text, which need to be searched to find patterns of interest. *Full-text searching* is the problem of locating the *occ* occurrences of a pattern $P[1..m]$ in a text $T[1..u]$, both modeled as sequences of symbols over an alphabet Σ of size σ. Unlike word-based text searching, we wish to find any *text substring*, not only whole *words* or *phrases*. This has applications in texts where the concept of word does not exist or is not well defined, such as in DNA or protein sequences, Oriental languages texts, MIDI pitch sequences, program code, etc. There exist two classical kind of queries, namely: (1) `count`(T, P): counts the number of occurrences of P in T; (2) `locate`(T, P): reports the starting positions of the *occ* occurrences of P in T.

Usually in practice the text is a very long sequence (of several of gigabytes, or even terabytes) which is known beforehand, and we want to locate (or count) the pattern occurrences as fast as possible. Thus, we preprocess T to build a data structure (or *index*), which is used to speed up the search, avoiding a sequential scan. However, by using an index we increase the space requirement. This is unfortunate when the text is very large. Traditional indexes like *suffix trees* [1]

* Supported in part by CONICYT PhD Fellowship Program (first author) and Fondecyt Grant 1-050493 (second author).

require $O(u \log u)$ bits to operate; in practice this space can be 10 times the text size [2], and so the index does not fit entirely in main memory even for moderate-size texts. In these cases the index must be stored on secondary memory and the search proceeds by loading the relevant parts into main memory.

Text compression is a technique to represent a text using less space. We denote by H_k the k-th *order empirical entropy* of a sequence of symbols T over an alphabet of size σ [3]. The value uH_k provides a lower bound to the 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 that $0 \leqslant H_k \leqslant H_{k-1} \leqslant \cdots \leqslant H_0 \leqslant \log \sigma$ (log means $\log_2$ in this paper).

To provide fast access to the text using little space, the current trend is to use *compressed full-text self-indexes*, which allows one to search and retrieve any part of the text without storing the text itself, while requiring space proportional to the compressed text size (e.g., $O(uH_k)$ bits) [4,5]. Therefore we *replace* the text with a more space-efficient representation of it, which at the same time provides indexed access to the text. This has applications in cases where we want to reduce the space requirement by not storing the text, or when accessing the text is so expensive that the index must search without having the text at hand, as occurs with most Web search engines. As compressed self-indexes replace the text, we are also interested in operations: (3) $\texttt{display}(T, P, \ell)$, which displays a context of ℓ symbols surrounding the pattern occurrences; and (4) $\texttt{extract}(T, i, j)$, which decompresses the substring $T[i..j]$, for any text positions $i \leqslant j$.

The use of a compressed full-text self-index may totally remove the need to use the disk. However, some texts are so large that their corresponding indexes do not fit entirely in main memory, even compressed. Unlike what happens with sequential text searching, which speeds up with compression because the compressed text is transferred faster to main memory [6], working on secondary storage with a compressed index usually requires *more* disk accesses in order to find the pattern occurrences. Yet, these indexes require less space, which in addition can reduce the seek time incurred by a larger index because seek time is roughly proportional to the size of the data.

We assume a model of computation where a *disk page* of size B (able to store $b = \omega(1)$ integers of $\log u$ bits, i.e. $B = b \log u$ bits) can be transferred to main memory in a single disk access. Because of their high cost, the performance of our algorithms is measured as the number of disk accesses performed to solve a query. We count *every* disk access, which is an upper bound to the real number of accesses, as we disregard the disk caching due to the operating system. We can hold a constant number of disk pages in main memory. We assume that our text T is static, i.e., there are no insertions nor deletions of text symbols.

There are not many works on full-text indexes on secondary storage, which definitely is an important issue. One of the best known indexes for secondary memory is the *String B-tree* [7], although this is not a compressed data structure. It requires (optimal) $O(\log_b u + \frac{m+occ}{b})$ disk accesses in searches and (worst-case optimal) $O(u/b)$ disk pages of space. This value is, in practice, about 12.5 times the text size (not including the text) [8], which is prohibitive for very large texts.

Clark and Munro [9] present a representation of suffix trees on secondary storage (the *Compact Pat Trees*, or *CPT* for short). This is not a compressed index, and also needs the text to operate. Although not providing worst-case guarantees, the representation is organized in such a way that the number of disk accesses is reduced to 3–4 per query. The authors claim that the space requirement of their index is comparable to that of suffix arrays, needing about 4–5 times the text size (not including the text).

Mäkinen et al. [10] propose a technique to store a *Compressed Suffix Array* on secondary storage, based on *backward searching* [11]. This is the only proposal to store a (*zero*-th order) compressed full-text self-index on secondary memory, requiring $u(H_0 + O(\log\log\sigma))$ bits of storage and a counting cost of at most $2(1 + m\lceil\log_B u\rceil)$ disk accesses. Locating the occurrences of the pattern would need $O(\log u)$ extra accesses per occurrence.

In this paper we propose a version of Navarro's LZ-index [12] that can be efficiently handled on secondary storage. Our index requires $8uH_k + o(u\log\sigma)$ bits of space for any $k = o(\log_\sigma u)$. In practice the space requirement is about 1.4–2.3 times the text size including the text, which is significantly smaller than any other practical secondary-memory data structure. Although we cannot provide worst-case guarantees at search time (just as in [9]), our experiments show that our index is effective in practice, yet requiring more disk accesses than larger indexes: our LZ-index is able to report up to 600 occurrences per disk access, for a disk page of 32 kilobytes. On the other hand, `count` queries can be performed requiring about 20–60 disk accesses (depending on the pattern length).

2 The LZ-Index Data Structure

Assume that the text $T[1..u]$ has been compressed using the LZ78 [13] algorithm into $n+1$ *phrases*, $T = B_0 \dots B_n$. We say that i is the *phrase identifier* of phrase B_i. The data structures that conform the LZ-index are [12]:

1. *LZTrie*: is the trie formed by all the LZ78 phrases $B_0 \dots B_n$. Given the properties of LZ78 compression, this trie has exactly $n+1$ nodes, each one corresponding to a phrase.
2. *RevTrie*: is the trie formed by all the reverse strings $B_0^r \dots B_n^r$. In this trie there could be *empty* nodes not representing any phrase.
3. *Node*: is a mapping from phrase identifiers to their node in *LZTrie*.
4. *RNode*: is a mapping from phrase identifiers to their node in *RevTrie*.

Each of these four structures requires $n\log n(1+o(1))$ bits if they are represented succinctly. As $n\log u = uH_k + O(kn\log\sigma) \leqslant u\log\sigma$ for any k [14], the final size of the LZ-index is $4uH_k + o(u\log\sigma)$ bits of space for any $k = o(\log_\sigma u)$.

We distinguish three types of occurrences of P in T, depending on the phrase layout [12]. For `locate` queries, pattern occurrences are reported in the format $[\![t, offset]\!]$, where t is the phrase where the occurrence starts, and $offset$ is the distance between the beginning of the occurrence and the end of the phrase. However, occurrences can be shown as text positions with little extra effort [15].

Occurrences of Type 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_i in the LZ78 referencing chain for B_k that contains the occurrence of P. Since phrase B_i has the string P as a suffix, P^r is a prefix of B_i^r, and can be easily found by searching for P^r in *RevTrie*. 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 subtrees of the *LZTrie* nodes corresponding to each v'. Total locate time is $O(m + occ_1)$.

Occurrences of Type 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. 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 t in the subtree of v_{rev} and report occurrence $[\![t, i]\!]$ if $Node[t+1]$ descends from v_{lz}. Each such check takes constant time. Yet, if the subtree of v_{lz} has fewer elements, we do the opposite: check phrases from v_{lz} in v_{rev}, using $RNode[t-1]$. The total time is proportional to the smallest subtree size among v_{rev} and v_{lz}.

Occurrences of Type 3. The occurrence spans three or more phrases, $B_{k-1} \ldots B_{\ell+1}$, such that $P[i..j] = B_k \ldots B_\ell$, $P[1..i-1]$ matches a suffix of B_{k-1} and $P[j+1..m]$ matches a prefix of $B_{\ell+1}$ (there are occ_3 occurrences of this type). As every phrase represents a different string (because of LZ78 properties), there is at most one phrase matching $P[i..j]$ for each choice of i and j. Thus, occ_3 is limited to $O(m^2)$ occurrences. We first identify the only possible phrase matching every substring $P[i..j]$. This is done by searching for every $P[i..j]$ in *LZTrie*, recording in a matrix $C_{lz}[i,j]$ the corresponding *LZTrie* node. Then we try to find the $O(m^2)$ maximal concatenations of successive phrases that match contiguous pattern substrings. If $P[i..j] = B_k \ldots B_\ell$ is a maximal concatenation, we check whether phrase $B_{\ell+1}$ starts with $P[j+1..m]$, i.e., we check whether $Node[\ell+1]$ is a descendant of node $C_{lz}[j+1, m]$. Finally we check whether phrase B_{k-1} ends with $P[1..i-1]$, by starting from $Node[i-1]$ in *LZTrie* and successively going to the parent to check whether the last $i-1$ nodes, read backwards, equal $P^r[1..i-1]$. If all these conditions hold, we report an occurrence $[\![k-1, i-1]\!]$. Overall locate time is $O(m^2 \log m)$ worst-case and $O(m^2)$ on average.

3 LZ-Index on Secondary Storage

The LZ-index [12] was originally designed to work in main memory, and hence it has a non-regular pattern of access to the index components. As a result, it is not suitable to work on secondary storage. In this section we show how to achieve locality in the access to the LZ-index components, so as to have good secondary storage performance. In this process we introduce some redundancy over main-memory proposals [12,15].

3.1 Solving the Basic Trie Operations

To represent the tries of the index we use a space-efficient representation similar to the hierarchical representation of [16], which now we make searchable. We cut the trie into disjoint *blocks* of size B such that every block stores a subtree of the whole trie. We arrange these blocks in a tree by adding some *inter-block* pointers, and thus the trie is represented by a tree of subtrees.

We cut the trie in a *bottom-up* fashion, trying to maximize the number of nodes in each block. This is the same partition used by Clark and Munro [9], and so we also suffer of very small blocks. To achieve a better fill ratio and reduce the space requirement, we store several trie blocks into each disk page.

Every trie node x in this representation is either a leaf of the whole trie, or it is an internal node. For internal nodes there are two cases: the node x is internal to a block p or x is a leaf of block p (but not a leaf of the whole trie). In the latter case, x stores a pointer to the representation q of its subtree. The leaf is also stored as a fictitious root of q, so that every block is a subtree. Therefore, every such node x has two representations: (1) as a leaf in block p; (2) as the root node of the child block q.

Each block p of N nodes and root node x consists basically of:

- the *balanced parentheses* (BP) representation [17] of the subtree, requiring $2N + o(N)$ bits;
- a bit-vector $F_p[1..N]$ (the *flags*) such that $F_p[j] = 1$ iff the j-th node of the block (in preorder) is a leaf of p, but not a leaf of the whole trie. In other words, the j-th node has a pointer to the representation of its subtree. We represent F_p using the data structure of [18] to allow *rank* and *select* queries in constant time and requiring $N + o(N)$ bits;
- the sequence $lets_p[1..N]$ of symbols labeling the arcs of the subtree, in preorder. The space requirement is $N\lceil \log \sigma \rceil$ bits;
- only in the case of *LZTrie*, the sequence $ids_p[1..N]$ of phrase identifiers in preorder. The space requirement is $N \log n$ bits;
- a pointer to the leaf representation of x in the parent block;
- the depth and preorder of x within the whole trie;
- a variable number of pointers to child blocks. The number of child blocks of a given block can be known from the number of 1s in F_p.
- an array $Size_p$ such that each pointer to child block stores the size of the corresponding subtree.

Using this information, given node x we are able to compute operations: $parent(x)$ (which gets the parent of x), $child(x, \alpha)$ (which yields the child of x by label α), $depth(x)$ (which gets the depth of x in the trie), $subtreesize(x)$ (which gets the size of the subtree of x, including x itself), $preorder(x)$ (which gets the preorder number of x in the trie), and $ancestor(x, y)$ (which tells us whether x is ancestor of node y). Operations *subtreesize*, *depth*, *preorder*, and *ancestor* can be computed without extra disk accesses, while operations *parent* and *child* require one disk access in the worst case. In [19] we explain how to compute them.

Analysis of Space Complexity. In the case of *LZTrie*, as the number of nodes is n, the space requirement is $2n + n + n \log \sigma + n \log n + o(n)$ bits, for the BP representation, the flags, the symbols, and phrase identifiers respectively. To this we must add the space required for the inter-block pointers and the extra information added to each block, such as the depth of the root, etc. If the trie is represented by a total of K blocks, these data add up to $O(K \log n)$ bits. The bottom-up partition of the trie ensures $K = O(n/b)$, so the extra information requires $O(\frac{n}{b} \log n)$ bits. As $b = \omega(1)$, this space is $o(n \log n) = o(u \log \sigma)$ bits.

In the case of *RevTrie*, as there can be empty nodes, we represent the trie using a *Patricia tree* [20], compressing empty unary paths so that there are $n \leqslant n' \leqslant 2n$ nodes. In the worst case the space requirement is $4n + 2n + 2n \log \sigma + o(n)$ bits, plus the extra information as before.

As we pack several trie blocks in a disk page, we ensure a utilization ratio of 50% at least. Hence the space of the tries can be at most doubled on disk.

3.2 Reducing the Navigation Between Structures

We add the following data structures with the aim of reducing the number of disk accesses required by the LZ-index at search time:

- $Pre_{lz}[1..n]$: a mapping from phrase identifiers to the corresponding *LZTrie* preorder, requiring $n \log n$ bits of space.
- $Rev[1..n]$: a mapping from *RevTrie* preorder positions to the corresponding *LZTrie* node, requiring $n \log u + n$ bits of space. Later in this section we explain why we need this space.
- $TPos_{lz}[1..n]$: if the phrase corresponding to the node with preorder i in *LZTrie* starts at position j in the text, then $TPos_{lz}[i]$ stores the value j. This array requires $n \log u$ bits and is used for `locate` queries.
- $LR[1..n]$: an array requiring $n \log n$ bits. If the node with preorder i in *LZTrie* corresponds to the LZ78 phrase B_k, then $LR[i]$ stores the preorder of the *RevTrie* node for B_{k-1}.
- $S_r[1..n]$: an array requiring $n \log u$ bits, storing in $S_r[i]$ the subtree size of the *LZTrie* node corresponding to the i-th *RevTrie* node (in preorder). This array is used for counting.
- $Node[1..n]$: the mapping from phrase identifiers to the corresponding *LZTrie* node, requiring $n \log n$ bits. This is used to solve `extract` queries.

As the size of these arrays depends on the compressed text size, we do not need that much space to store them: they require $3n \log u + 3n \log n + n$ bits, which summed to the tries gives $8uH_k + o(u \log \sigma)$ bits, for any $k = o(\log_\sigma u)$.

If the index is used *only* for `count` queries, we basically need arrays Pre_{lz}, LR, S_r, and the tries, plus an array $RL[1..n]$, which is similar to LR but mapping from a *RevTrie* node for B_k to the *LZTrie* preorder for B_{k+1}. All these add up to $6uH_k + o(u \log \sigma)$ bits.

After searching for all pattern substrings $P[i..j]$ in *LZTrie* (recording in $C_{lz}[i,j]$ the phrase identifier, the preorder, and the subtree size of the corresponding *LZTrie* node, along with the node itself) and all reversed prefixes $P^r[1..i]$ in *RevTrie* (recording in array $C_r[i]$ the preorder and subtree size of the corresponding *RevTrie* node), we explain how to find the pattern occurrences.

Occurrences of Type 1. Assume that the search for P^r in *RevTrie* yields node v_r. For every node with preorder i, such that $preorder(v_r) \leqslant i \leqslant preorder(v_r) + subtreesize(v_r)$ in *RevTrie*, with $Rev[i]$ we get the node v_{lz_i} in *LZTrie* representing a phrase B_t ending with P. The length of B_t is $d = depth(v_{lz_i})$, and the occurrence starts at position $d - m$ inside B_t. Therefore, if $p = preorder(v_{lz_i})$, the exact text position can be computed as $TPos_{lz}[p] + d - m$. We then traverse all the subtree of v_{lz_i} and report, as an occurrence of type 1, each node contained in this subtree, accessing $TPos_{lz}[p..p + subtreesize(v_{lz_i})]$ to find the text positions. Note that the offset $d - m$ stays constant for all nodes in the subtree.

Note that every node in the subtree of v_r produces a random access in *LZTrie*. In the worst case, the subtree of v_{lz_i} has only one element to report (v_{lz_i} itself), and hence we have occ_1 random accesses in the worst case. To reduce the worst case to $occ_1/2$, we use the n extra bits in *Rev*: in front of the $\log u$ bits of each *Rev* element, a bit indicates whether we are pointing to a *LZTrie* leaf. In such a case we do not perform a random access to *LZTrie*, but we use the corresponding $\log u$ bits to store the exact text position of the occurrence.

To avoid accessing the same *LZTrie* page more than once, even for different trie blocks stored in that page, for each $Rev[i]$ we solve all the other $Rev[j]$ that need to access the same *LZTrie* page. As the tries are space-efficient, many random accesses could need to access the same page.

For `count` queries we traverse the S_r array instead of *Rev*, summing up the sizes of the corresponding *LZTrie* subtrees without accessing them, therefore requiring $O(occ_1/b)$ disk accesses.

Occurrences of Type 2. For occurrences of type 2 we consider every possible partition $P[1..i]$ and $P[i+1..m]$ of P. Suppose the search for $P^r[1..i]$ in *RevTrie* yields node v_r (with preorder p_r and subtree size s_r), and the search for $P[i+1..m]$ in *LZTrie* yields node v_{lz} (with preorder p_{lz} and subtree size s_{lz}). Then we traverse sequentially $LR[j]$, for $j = p_{lz}..p_{lz} + s_{lz}$, reporting an occurrence at text position $TPos_{lz}[j] - i$ iff $LR[j] \in [p_r..p_r + s_r]$. This algorithm has the nice property of traversing arrays LR and $TPos_{lz}$ sequentially, yet the number of elements traversed can be arbitrarily larger than occ_2.

For `count` queries, since we have also array RL, we choose to traverse $RL[j]$, for $j = p_r..p_r + s_r$, when the subtree of v_r is smaller than that of v_{lz}, counting an occurrence only if $RL[j] \in [p_{lz}..p_{lz} + s_{lz}]$.

To reduce the number of accesses from $2\lceil \frac{s_{lz}+1}{b} \rceil$ to $\lceil \frac{2(s_{lz}+1)}{b} \rceil$, we interleave arrays LR and $TPos_{lz}$, such that we store $LR[1]$ followed by $TPos_{lz}[1]$, then $LR[2]$ followed by $TPos_{lz}[2]$, etc.

Occurrences of Type 3. We find all the maximal concatenations of phrases using the information stored in C_{lz} and C_r. If we found that $P[i..j] = B_k \dots B_\ell$ is a maximal concatenation, we check whether phrase $B_{\ell+1}$ has $P[j+1..m]$ as a prefix, and whether phrase B_{k-1} has $P[1..i-1]$ as a suffix. Note that, according to the LZ78 properties, $B_{\ell+1}$ starting with $P[j+1..m]$ implies that there exists a previous phrase B_t, $t < \ell+1$, such that $B_t = P[j+1..m]$. In other words, $C_{lz}[j+1,m]$ must not be null (i.e., phrase B_t must exist) and the phrase identifier stored at $C_{lz}[j+1,m]$ must be less than $\ell+1$ (i.e., $t < \ell+1$). If these conditions hold, we check whether $P^r[1..i-1]$ exists in *RevTrie*, using the information stored at $C_r[i-1]$. Only if all these condition hold, we check whether $Pre_{lz}[\ell+1]$ descends from the *LZTrie* node corresponding to $P[j+1..m]$ (using the preorder and subtree size stored at $C_{lz}[j+1,m]$), and if we pass this check, we finally check whether $LR[Pre_{lz}[k]]$ (which yields the *RevTrie* preorder of the node corresponding to phrase $k-1$) descend from the *RevTrie* node for $P^r[1..i-1]$ (using the preorder and subtree size stored at $C_r[i-1]$). Fortunately, we have a high probability that $Pre_{lz}[\ell+1]$ and $Pre_{lz}[k]$ need to access the same disk page. If we find an occurrence, the corresponding position is $TPos_{lz}[Pre_{lz}[k]] - (i-1)$.

`Extract` *Queries.* In [19] we explain how to solve `extract` queries.

4 Experimental Results

For the experiments of this paper we consider two text files: the text WSJ (Wall Street Journal) from the TREC collection [21], of 128 megabytes, and the XML file provided in the *Pizza&Chili Corpus*[1], downloadable from `http://pizzachili.dcc.uchile.cl/texts/xml/dblp.xml.200MB.gz`, of 200 megabytes. We searched for 5,000 random patterns, of length from 5 to 50, generated from these files. As in [8], we assume a disk page size of 32 kilobytes. We compared our results against the following state-of-the-art indexes for secondary storage:

***Suffix Arrays* (SA):** following [22] we divide the suffix array into blocks of $h \leqslant b$ elements (pointers to text suffixes), and move to main memory the first l text symbols of the first suffix of each block, i.e. there are $\frac{u}{h}l$ extra symbols. We assume in our experiments that $l = m$ holds, which is the best situation. At search time, we carry out two binary searches [23] to delimit the interval $[i..j]$ of the pattern occurrences. Yet, the first part of the binary search is done over the samples without accessing the disk. Once the blocks where i and j lie are identified, we bring them to main memory and finish the binary search, this time accessing the text on disk at each comparison. Therefore, the total cost is $2 + 2\log h$ disk accesses. We must pay $\lceil \frac{occ}{b} \rceil$ extra accesses to report the occurrences of P within those two positions. The space requirement including the text is $(5 + \frac{m}{h})$ times the text size.

***String B-trees* [7]:** in [8] they pointed out that an implementation of *String B-trees* for static texts would require about $2 + \frac{2.125}{k}$ times the text size

[1] `http://pizzachili.dcc.uchile.cl`

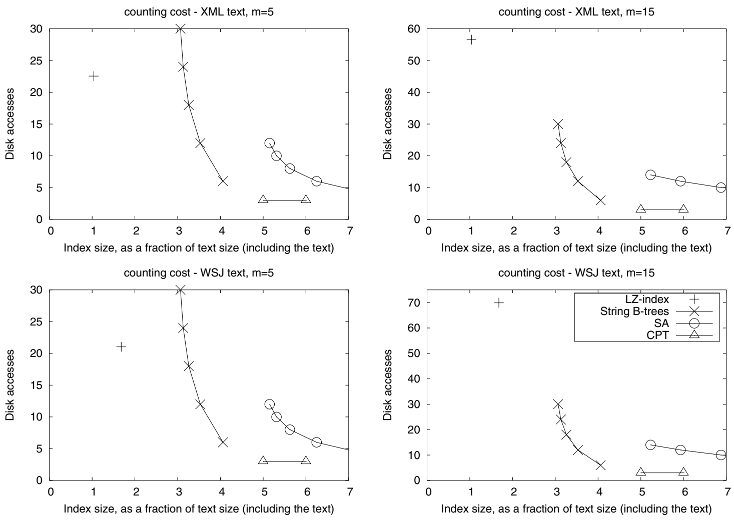


Fig. 1. `Count` cost vs. space requirement for the different indexes we tested

(where $k > 0$ is a constant) and the height h of the tree is 3 for texts of up to 2 gigabytes, since the branching factor (number of children of each tree node) is $b' \approx \frac{b}{8.25}$. The experimental number of disk accesses given by the authors is $O(\log k)(\lfloor \frac{m}{b} \rfloor + 2h) + \lceil \frac{occ}{b'} \rceil$. We assume a constant of 1 for the $O(\log k)$ factor, since this is not clear in the paper [8, Sect. 2.1] (this is optimistic). We use $k = 2$, 4, 8, 16, and 32.

***Compact Pat Trees* (CPT) [9]:** we assume that the tree has height 3, according to the experimental results of Clark and Munro. We need $1 + \lfloor \frac{occ}{b} \rfloor$ extra accesses to locate the pattern occurrences. The space is about 4–5 times the text size (plus the text).

We restrict our comparison to indexes that have been implemented, or at least simulated, in the literature. Hence we exclude the *Compressed Suffix Arrays* (CSA) [10] since we only know that it needs at most $2(1+m\lceil \log_b u \rceil)$ accesses for `count` queries. This index requires about 0.22 and 0.45 times the text size for the XML and WSJ texts respectively, which, as we shall see, is smaller than ours. However, CSA requires $O(\log u)$ accesses to report *each* pattern occurrence[2].

Fig. 1 shows the time/space trade-offs of the different indexes for `count` queries, for patterns of length 5 and 15. As it can be seen, our LZ-index requires about 1.04 times the text size for the (highly compressible) XML text, and 1.68 times the text size for the WSJ text. For $m = 5$, the counting requires about 23 disk accesses, and for $m = 15$ it needs about 69 accesses. Note that for $m = 5$, there is a difference of 10 disk accesses among the LZ-index and

[2] The work [24] extends this structure to achieve fast locate. The secondary-memory version is still a theoretical proposal and it is hard to predict how will it perform, so we cannot meaningfully compare it here.

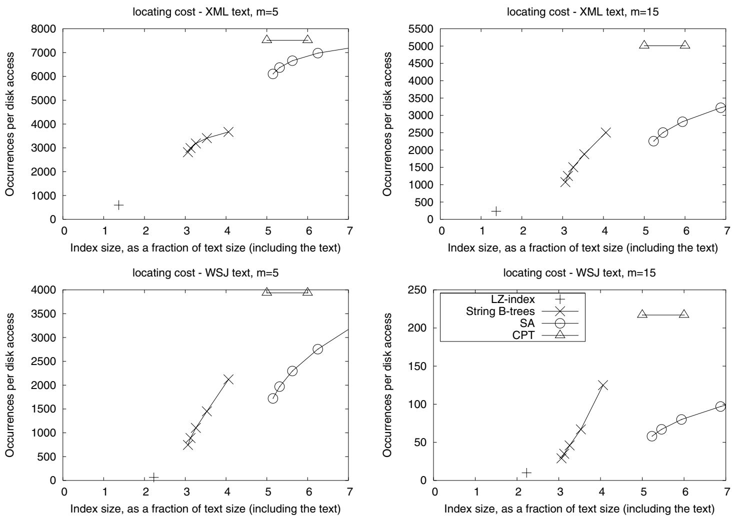


Fig. 2. `Locate` cost vs. space requirement for the different indexes we tested. Higher means better `locate` performance.

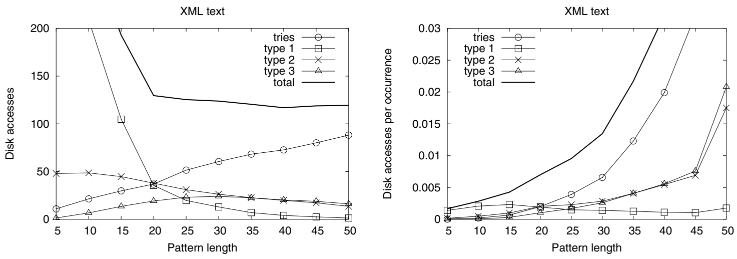


Fig. 3. Cost for the different parts of the LZ-index search algorithm

String B-trees, the latter requiring 3.39 (XML) and 2.10 (WSJ) times the space of the LZ-index. For $m = 15$ the difference is greater in favor of *String B-Trees*. The SA outperforms the LZ-index in both cases, the latter requiring about 20% the space of SA. Finally, the LZ-index needs (depending on the pattern length) about 7–23 times the number of accesses of CPTs, the latter requiring 4.9–5.8 (XML) and 3–3.6 (WSJ) times the space of LZ-index.

Fig. 2 shows the time/space trade-offs for `locate` queries, this time showing the average number of occurrences reported per disk access. The LZ-index requires about 1.37 (XML) and 2.23 (WSJ) times the text size, and is able of reporting about 597 (XML) and 63 (WSJ) occurrences per disk access for $m = 5$, and about 234 (XML) and 10 (WSJ) occurrences per disk access for $m = 15$. The average number of occurrences found for $m = 5$ is 293,038 (XML) and 27,565 (WSJ); for $m = 15$ there are 45,087 and 870 occurrences on average. *String B-trees* report 3,449 (XML) and 1,450 (WSJ) occurrences per access for $m = 5$, and for $m = 15$ the results are 1,964 (XML) and 66 (WSJ) occurrences per access, requiring 2.57 (XML) and 1.58 (WSJ) times the space of the LZ-index.

Fig. 3 shows the cost for the different parts of the LZ-index search algorithm, in the case of XML (WSJ yields similar results): the work done in the tries (labeled "`tries`"), the different types of occurrences, and the total cost ("`total`"). The total cost can be decomposed in three components: a part linear on m (trie traversal), a part linear in occ (type 1), and a constant part (type 2 and 3).

5 Conclusions and Further Work

The LZ-index [12] can be adapted to work on secondary storage, requiring up to $8uH_k + o(u \log \sigma)$ bits of space, for any $k = o(\log_\sigma u)$. In practice, this value is about 1.4–2.3 times the text size, including the text, which means 39%–65% the space of *String B-trees* [7]. Saving space in secondary storage is important not only by itself (space is very important for storage media of limited size, such as CD-ROMs), but also to reduce the high seek time incurred by a larger index, which usually is the main component in the cost of accessing secondary storage, and is roughly proportional to the size of the data.

Our index is significantly smaller than any other practical secondary-memory data structure. In exchange, it requires more disk accesses to locate the pattern occurrences. For XML text, we are able to report (depending on the pattern length) about 597 occurrences per disk access, versus 3,449 occurrences reported by *String B-trees*. For English text (WSJ file from [21]), the numbers are 63 vs. 1,450 occurrences per disk access. In many applications, it is important to find quickly a few pattern occurrences, so as to find the remaining while processing the first ones, or on user demand (think for example in Web search engines). Fig. 3 (left, see the line "`tries`") shows that for $m = 5$ we need about 11 disk accesses to report the first pattern occurrence, while *String B-trees* need about 12. If we only want to count the pattern occurrences, the space can be dropped to $6uH_k + o(u \log \sigma)$ bits; in practice 1.0–1.7 times the text size. This means 29%–48% the space of *String B-trees*, with a slowdown of 2–4 in the time.

We have considered only number of disk accesses in this paper, ignoring seek times. Random seeks cost roughly proportionally to the size of the data. If we multiply number of accesses by index size, we get a very rough idea of the overall seek times. The smaller size of our LZ-index should favor it in practice. For example, it is very close to *String B-trees* for counting on XML and $m = 5$ (Fig. 1). This product model is optimistic, but counting only accesses is pessimistic.

As future work we plan to handle dynamism and the direct construction on secondary storage, adapting the method of [16] to work on disk.

References

1. Apostolico, A.: The myriad virtues of subword trees. In: Combinatorial Algorithms on Words. NATO ISI Series, pp. 85–96. Springer, Heidelberg (1985)
2. Kurtz, S.: Reducing the space requeriments of suffix trees. Softw. Pract. Exper. 29(13), 1149–1171 (1999)

3. Manzini, G.: An analysis of the Burrows-Wheeler transform. JACM 48(3), 407–430 (2001)
4. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Computing Surveys (to appear)
5. Ferragina, P., Manzini, G.: Indexing compressed texts. JACM 54(4), 552–581 (2005)
6. Moura, E., Navarro, G., Ziviani, N., Baeza-Yates, R.: Fast and flexible word searching on compressed text. ACM TOIS 18(2), 113–139 (2000)
7. Ferragina, P., Grossi, R.: The String B-tree: a new data structure for string search in external memory and its applications. JACM 46(2), 236–280 (1999)
8. Ferragina, P., Grossi, R.: Fast string searching in secondary storage: theoretical developments and experimental results. In: Proc. SODA, pp. 373–382 (1996)
9. Clark, D., Munro, J.I.: Efficient suffix trees on secondary storage. In: Proc. SODA, 383–391 (1996)
10. Mäkinen, V., Navarro, G., Sadakane, K.: Advantages of backward searching — efficient secondary memory and distributed implementation of compressed suffix arrays. In: Proc. ISAAC, pp. 681–692 (2004)
11. Sadakane, K.: Succinct representations of lcp information and improvements in the compressed suffix arrays. In: Proc. SODA, pp. 225–232 (2002)
12. Navarro, G.: Indexing text using the Ziv-Lempel trie. J. of Discrete Algorithms 2(1), 87–114 (2004)
13. Ziv, J., Lempel, A.: Compression of individual sequences via variable-rate coding. IEEE TIT 24(5), 530–536 (1978)
14. Kosaraju, R., Manzini, G.: Compression of low entropy strings with Lempel-Ziv algorithms. SIAM J.Comp. 29(3), 893–911 (1999)
15. Arroyuelo, D., Navarro, G., Sadakane, K.: Reducing the space requirement of LZ-index. In: Proc. CPM, pp. 319–330 (2006)
16. Arroyuelo, D., Navarro, G.: Space-efficient construction of LZ-index. In: Proc. ISAAC pp. 1143–1152 (2005)
17. Munro, I., Raman, V.: Succinct representation of balanced parentheses and static trees. SIAM J.Comp. 31(3), 762–776 (2001)
18. Munro, I.: Tables. In: Chandru, V., Vinay, V. (eds.) Foundations of Software Technology and Theoretical Computer Science. LNCS, vol. 1180, pp. 37–42. Springer, Heidelberg (1996)
19. Arroyuelo, D., Navarro, G.: A Lempel-Ziv text index on secondary storage. Technical Report TR/DCC-2004, -4, Dept. of Computer Science, Universidad de Chile (2007) ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/lzidisk.ps.gz
20. Morrison, D.R.: Patricia – practical algorithm to retrieve information coded in alphanumeric. JACM 15(4), 514–534 (1968)
21. Harman, D.: Overview of the third text REtrieval conference. In: Proc. Third Text REtrieval Conference (TREC-3), NIST Special Publication, pp. 500–207 (1995)
22. Baeza-Yates, R., Barbosa, E.F., Ziviani, N.: Hierarchies of indices for text searching. Inf. Systems 21(6), 497–514 (1996)
23. Manber, U., Myers, G.: Suffix arrays: A new method for on-line string searches. SIAM J.Comp. 22(5), 935–948 (1993)
24. González, R., Navarro, G.: Compressed text indexes with fast locate. In: Proc. of CPM'07. LNCS (To appear 2007)

Dynamic Rank-Select Structures with Applications to Run-Length Encoded Texts* (Extended Abstract)

Sunho Lee and Kunsoo Park**

School of Computer Science and Engineering,
Seoul National University, Seoul, 151-742, Korea
kpark@theory.snu.ac.kr

Abstract. Given an n-length text over a σ-size alphabet, we propose a dynamic rank-select structure that supports $O((1+\frac{\log\sigma}{\log\log n})\log n)$ time operations in $n\log\sigma + o(n\log\sigma)$ bits space. If $\sigma < \log n$, then the operation time is $O(\log n)$. In addition, we consider both static and dynamic rank-select structures on the run-length encoding (RLE) of a text. For an n'-length RLE of an n-length text, we present a static structure that gives $O(1)$ time *select* and $O(\log\log\sigma)$ time *rank* using $n'\log\sigma + O(n)$ bits and a dynamic structure that provides $O((1+\frac{\log\sigma}{\log\log n})\log n)$ time operations in $n'\log\sigma + o(n'\log\sigma) + O(n)$ bits.

1 Introduction

A succinct rank-select structure is an essential ingredient of compressed full text indices such as compressed suffix array (CSA) [8,24] and FM-index [4]. Given a text of length n over a σ-size alphabet, the succinct rank-select structure occupies only the same space as the text T, $n\log\sigma$ bits, plus a small extra space, $o(n\log\sigma)$ bits. The structure can be compressed into even smaller space, $nH_k + o(n\log\sigma)$ bits, where H_k is the empirical k-th order entropy of T [16].

The static rank-select structure answers the following queries.

- $rank_T(c,i)$: gives the number of character c's up to position i in text T
- $select_T(c,k)$: gives the position of the k-th c in T.

For a dynamic structure, we consider the following insert and delete operations on T in addition to $rank_T$ and $select_T$.

- $insert_T(c,i)$: inserts character c between $T[i]$ and $T[i+1]$
- $delete_T(i)$: deletes $T[i]$ from text T

In compressed full-text indices for a text T, a rank-select structure is built on Burrows-Wheeler Transform (BWT) of T, which is a permutation of T made

* This work was supported by FPR05A2-341 of 21C Frontier Functional Proteomics Project from Korean Ministry of Science & Technology.

** Corresponding author.

Table 1. Static rank-select structures

Alphabet	Text	Time	Space	Reference
Binary	Plain	$O(\log n)$	$n + o(n)$	Jacobson [12]
($\sigma = 2$)		$O(1)$	$n + O(\frac{n \log\log n}{\log n})$ rank $n + O(\frac{n}{\log\log n})$ select	Clark [2], Munro [18]
		$O(1)$	$n + O(\frac{n \log\log n}{\log n})$ rank $n + O(\frac{n \log\log n}{\sqrt{\log n}})$ select	Kim, Na, Kim and Park [13]
		$O(1)$	$nH_0 + o(n)$	Raman, Raman and Rao [23]
polylog(n)	Plain	$O(1)$	$nH_0 + o(n)$	Ferragina, Manzini, Mäkinen and Navarro [5]
$O(n^\beta)$, $\beta < 1$		$O(\frac{\log \sigma}{\log\log n})$	$nH_0 + o(n \log \sigma)$	
General	Plain	$O(\log \sigma)$	$nH_0 + o(n \log \sigma)$	Grossi, Gupta and Vitter [7]
		$O(1)$ select $O(\log\log \sigma)$ rank	$nH_0 + O(n)$	Golynski, Munro and Rao [6]
		$O(1)$ select $O(\log\log \sigma)$ rank	$n \log \sigma + O(n)$	Hon, Sadakane and Sung [10]
	RLE	$O(\log \sigma)$ rank	$n'H'_0 + O(n)$	Mäkinen and Navarro [14]
		$O(1)$ select $O(\log\log \sigma)$ rank	$n'H'_0 + O(n)$	This paper

from lexicographically sorted suffixes of T. This rank-select structure on BWT of T directly supports the functions of compressed full-text indices. For instance, $O(1)$ *rank* queries are posted in each step of backward searching [4]. From the duality between BWT and Ψ-function of CSA [10], *select* queries are used implicitly to give Ψ-function.

Since BWT provides easy compression of a text [16], Mäkinen and Navarro employed Run-Length Encoding (RLE) of BWT to obtain Run-Length based FM-index (RLFM) [14]. They showed that the length of RLE of BWT is $nH_k + o(n)$ for $k < \alpha \log_\sigma n$, $0 < \alpha < 1$. Applying RLE to a rank structure produces RLFM of $nH_k \log \sigma + o(n \log \sigma) + O(n)$ bits space. RLFM is built upon a rank structure on a plain text as a black box.

In this paper, we propose a dynamic rank-select structure that supports $O((1 + \frac{\log \sigma}{\log\log n}) \log n)$ time operations in $n \log \sigma + o(n \log \sigma)$ bits space. For a small alphabet with $\sigma \leq \log n$, our structure gives $O(\log n)$ time operations. For a large alphabet with $\sigma > \log n$, the time complexity of our structure is an improvement upon [11]. In addition, we consider both static and dynamic rank-select structures on RLE, by following Mäkinen and Navarro's RLFM strategy. For an n'-length RLE of an n-length text, we obtain a static structure that gives $O(1)$ time *select* and $O(\log\log \sigma)$ time *rank* using $n' \log \sigma + O(n)$ bits and a dynamic structure that provides $O((1 + \frac{\log \sigma}{\log\log n}) \log n)$ time operations in $n' \log \sigma + o(n' \log \sigma) + O(n)$ bits.

The results of static rank-select structures are shown in Table 1. Jacobson first considered this problem for a binary alphabet and gave $O(\log n)$ time rank-select using $n + o(n)$ bits [12]. This was improved to $O(1)$ time rank-select in

Table 2. Dynamic rank-select structures

Alphabet	Text	Time	Space	Reference
$\sigma \le \log n$	Plain	$O(\log n)$	$n \log \sigma + o(n \log \sigma)$	This paper
	RLE	$O(\log n)$	$n' \log \sigma + o(n' \log \sigma)$ $+O(n)$	
General	Plain	$O(\log \sigma \frac{\log n}{\log \log n})$	$n \log \sigma + o(n \log \sigma)$	Raman, Raman and Rao [22]
		$O(\log \sigma \log_b n)$ rank-select $O(\log \sigma\ b)$ flip		
		$O(\log \sigma \log_b n)$ rank-select $O(\log \sigma\ b)$ insert-delete	$n \log \sigma + o(n \log \sigma)$	Hon, Sadakane and Sung [11]
		$O(\log \sigma \log n)$	$nH_0 + o(n \log \sigma)$	Mäkinen and Navarro [15]
		$O((1/\epsilon) \log \log n)$ rank-select $O((1/\epsilon) n^\epsilon)$ insert-delete	$n \log \sigma + o(n \log \sigma)$	Gupta, Hon, Shah and Vitter [9]
		$O((1 + \frac{\log \sigma}{\log \log n}) \log n)$	$n \log \sigma + o(n \log \sigma)$	This paper
	RLE	$O((1 + \frac{\log \sigma}{\log \log n}) \log n)$	$n' \log \sigma + o(n' \log \sigma)$ $+O(n)$	

$n+o(n)$ bits by Clark [2] and Munro [18], and further improved in space by Kim et al [13]. The structure by Raman et al. [23] achieves $O(1)$ time rank-select in $nH_0 + o(n)$ bits space. The trade-off between time and space was studied by Miltersen [17]. These binary structures can be the key structures of various succinct data structures [19,20,21,23].

The rank-select structure for a large alphabet was studied in the context of compressed full-text indices. Grossi et al. [7] gave a wavelet-tree structure of $nH_0 + o(n \log \sigma)$ bits size, which answers rank-select queries in $O(\log \sigma)$ time. This was improved for certain alphabet sizes by Ferragina et al [5]. The structure for a general alphabet was considered by Golynski et al. [6] and by Hon et al [10]. Using these results, the rank structure on RLE by Mäkinen and Navarro [14] immediately gives $O(\log \log \sigma)$ time rank instead of $O(\log \sigma)$ time rank.

The results of dynamic structures are shown in Table 2. For the case of a binary alphabet, it was addressed as a special case of the partial sum problem by Raman et al. [22] and Hon et al [11]. Raman et al. gave $O(\frac{\log n}{\log \log n})$ worst-case time rank-select-flip operations or $O(\log_b n)$ time rank-select with $O(b)$ amortized time flip in $n+o(n)$ bits space for $\frac{\log n}{\log \log n} \le b < n$. It was extended to support insert-delete by Hon et al [11]. Mäkinen and Navarro [15] gave an entropy-bound structure of $nH_0+o(n)$ bits space, which enables $O(\log n)$ worst-case time operations. For the case of a large alphabet, the extensions of binary structures using wavelet-trees as in [7,15] achieve $O(n \log \sigma)$ bits space with $O(\log \sigma)$ slowdown factor (Table 2). Recently, Gupta et al. [9] presented a dynamic structure of $n \log \sigma + o(n \log \sigma)$ bits space, which provides $O((1/\epsilon) \log \log n)$ time retrieving queries and $O((1/\epsilon) n^\epsilon)$ updating queries, without wavelet-trees.

Our rank-select structures can be applied to index structures based on BWT. For instance, our dynamic structure can be applied to the index of a collection of texts by Chan et al [1]. The index of a collection in [1] has two structures, *COUNT* and *PSI*, which can be replaced by *rank* and *select* on BWT of the concatenation of given texts. For the collection indexing, our structure reduces its space usage to $nH_k \log \sigma + o(nH_k \log \sigma) + O(n)$ bits, and removes its constraint that σ has to be a constant. Note that H_k is defined on the concatenation of the given texts. Our structures also support the XML indexing, which is based on the XBW transform proposed by Ferragina et al [3]. The XBW transform is an extension of BWT for XML trees, which can be handled by rank-select operations.

2 Definitions and Preliminaries

We denote by $T = T[1]T[2]\ldots T[n]$ the text of length n over a σ-size alphabet $\Sigma = \{0, 1, \ldots \sigma - 1\}$. T is assumed to be terminated by a unique endmarker, $T[n] = \$$, which is lexicographically smaller than any other character in Σ. We assume the alphabet size σ is $o(n)$, because otherwise the size of text becomes $n \log \sigma = \Omega(n \log n)$ bits; $\Omega(n \log n)$ bits space makes the problem easy. This is the only assumption on the alphabet size.

Let n-length text $T = c_1^{l_1} c_2^{l_2} \ldots c_{n'}^{l_{n'}}$ be a concatenation of n' runs, where a run is l_i consecutive occurrences of a same character c_i. We represent a run-length encoding of T by two vectors, T' and L, such that $T' = c_1 c_2 \ldots c_{n'}$ consists of the character of each run, and bit vector $L = 10^{l_1 - 1} 10^{l_2 - 1} \ldots 10^{l_{n'} - 1}$ contains the length of each run. T' and L are denoted by $RLE(T)$, our representation of RLE of T.

The compressed full-text indices usually consist of two components: one is the Burrows-Wheeler Transform (BWT) of a text T, and the other is the Ψ-function. The BWT of T, $BWT(T)$, is a permutation of T, which has the information of suffix arrays [16]. Because $BWT(T)$ takes the same size as T, $BWT(T)$ can replace the suffix array of $O(n \log n)$ bits size and it is employed by compressed full-text indices, explicitly or implicitly. Mäkinen and Navarro showed that the number of runs in $BWT(T)$ is less than $nH_k + \sigma^k$, so the size of rank-select structures on $RLE(BWT(T))$ is bounded by the following lemma.

Lemma 1. [14] *The number of runs in* $BWT(T)$ *is at most* $n \cdot \min(H_k(T), 1) + \sigma^k$*, for any* $k \geq 0$*. In particular, this is* $nH_k(T) + o(n)$ *for any* $k \leq \alpha \log_\sigma n$*, for any constant* $0 < \alpha < 1$*.*

The other component, Ψ-function, gives the lexicographic order of the next suffix. Let $SA[i]$ denote the starting position of the lexicographically i-th suffix of T, and $SA^{-1}[i]$ be the order of the suffix with starting position i, $T[i, n]$. The Ψ-function is defined as

$$\Psi[i] = \begin{cases} SA^{-1}[SA[i] + 1] & \text{if} \quad SA[i] \neq n \\ SA^{-1}[1] & \text{otherwise.} \end{cases}$$

In fact, from the duality between $BWT(T)$ and Ψ-function [10] we obtain Ψ-function by applying *select* on $BWT(T)$. Let $F[c]$ be the number of occurrences of characters less than c in the text T. We also denote by $C[i]$, the first character of the i-th suffix, $T[SA[i], n]$. Then, Ψ-function is

$$\Psi[i] = select_{BWT(T)}(C[i], i - F[C[i]]).$$

3 Dynamic Rank-Select Structures on a Plain Text

In this section, we prove the following theorem by considering two cases of the alphabet size. For a small alphabet with $\sigma \leq \log n$, we improve Mäkinen and Navarro's binary structure [15] to support all operations in $O(\log n)$ time. For a large alphabet with $\sigma > \log n$, we extend our small-alphabet structure to provide $O((1 + \frac{\log \sigma}{\log \log n}) \log n)$ time operations, by using k-ary wavelet-trees [7,5].

Theorem 1. *Given a text T over an alphabet of size σ, there is a dynamic structure that supports $O((1 + \frac{\log \sigma}{\log \log n}) \log n)$ worst-case time rank, select and access, while supporting $O((1 + \frac{\log \sigma}{\log \log n}) \log n)$ amortized time insert and delete in $n \log \sigma + o(n \log \sigma)$ bits space.*

3.1 Dynamic Rank-Select Structure for a Small Alphabet

We assume the RAM model with word size $w = \Theta(\log n)$ bits, which supports an addition, a multiplication, and bitwise operations in $O(1)$ time. For a small alphabet structure, we partition the text into blocks of size $\frac{1}{2} \log n$ to $2 \log n$ words. Because a dynamic operation handles variable sizes of blocks, we define a partition vector I that represents the boundary of a block by a 1 and the following 0s. This partition vector I gives the mapping between a position and a block. For position i, $rank_I(1, i)$ gives the number of the block that contains i, and $select_I(1, k)$ inversely returns the starting position of the k-th block. We employ Mäkinen and Navarro's binary structure for the operations on I.

Using the partition I, we divide a given operation into an over-block operation and an in-block one. For instance, given $rank_T(c, i)$, we first compute the b-th block that contains i and the offset r of position i in the b-th block by $b = rank_I(1, i)$ and $r = i - select_I(1, b) + 1$. The over-block rank gives the number of occurrences of c before the b-th block. The in-block rank returns the number of c up to position r in the b-th block. Then, $rank_T(c, i)$ is the sum of these over-block rank and in-block rank. We define the over-block operations as

– $\textit{rank-over}_T(c, b)$: gives the number of c's in blocks preceding the b-th block.
– $\textit{select-over}_T(c, k)$: gives the number of the block containing the k-th c.
– $\textit{insert-over}_T(c, b)$: increases the number of c's in the b-th block.
– $\textit{delete-over}_T(c, b)$: decreases the number of c's in the b-th block.

Let T_b be the b-th block. The in-block operation for the b-th block is defined as

– $rank_{T_b}(c, r)$: gives the number of c's up to position r in T_b.
– $select_{T_b}(c, k)$: gives the position of the k-th c in T_b.

- $insert_{T_b}(c, r)$: inserts character c between $T_b[r]$ and $T_b[r+1]$.
- $delete_{T_b}(r)$: deletes the r-th character of T_b.

Note that the over-block updates just change the structure for over-block operations, and the in-block updates actually change the text itself.

In-block Operations To process in-block operations, we employ the same hierarchy as Mäkinen and Navarro's: a word, a sub-block and a block. We first show that the characters in a word can be processed in $O(1)$ time. Instead of a word of $\Theta(\log n)$ bits for the binary structure, our word contains $\Theta(\frac{\log n}{\log \sigma})$ characters for a σ-size alphabet. We convert $\Theta(\frac{\log n}{\log \sigma})$ characters to $\Theta(\frac{\log n}{\log \sigma})$ bits by bitwise operations, and then use the rank-select tables which take $o(n)$ bits space and give binary rank-select on a word in $O(1)$ time [2,18,13,23].

Lemma 2. *We can process rank, select, insert, and delete on a text of a word size with $\Theta(\frac{\log n}{\log \sigma})$ characters, by using $O(1)$ time and $o(n)$ bits space.*

Secondly, we show a structure for $O(\log n)$ time in-block operations as in Mäkinen and Navarro's. To refer to the b-th block, we build a red-black tree over the blocks. Each block becomes a leaf node, and an internal node v maintains $n(v)$, the number of blocks in the subtree rooted at v. From the root node to the b-th block, we choose a child of v by comparing $n(left(v))$ and b, where $left(v)$ is the left child of v. If $n(left(v)) > b$, the b-th block is in the left subtree of v. Otherwise the b-th block is in the right subtree of v. The total size of the tree is $\frac{2n}{\log n\, \Theta(\frac{\log n}{\log \sigma})} \cdot O(\log n) = O(\frac{n \log \sigma}{\log n})$ bits.

Mäkinen and Navarro introduced sub-block structures that use only $o(n \log \sigma)$ bits extra space. Because we set the block size from $\frac{1}{2}\log n$ to $2\log n$ words, allocating $2\log n$ words for each block can waste $O(\log n)$ words as extra space for each block. This causes our structure to waste total $O(n \log \sigma)$ bits space. To use only $o(n \log \sigma)$ bits space, each block is divided into sub-blocks of $\sqrt{\log n}$ words and a block has only one sub-block as extra space. A block maintains $\frac{1}{2}\sqrt{\log n}$ to $2\sqrt{\log n}$ sub-blocks by using a linked list, whose last sub-block is the extra sub-block. Since the pointer size is $O(\log n)$, we use total $\frac{2n}{\sqrt{\log n}\, \Theta(\frac{\log n}{\log \sigma})} \cdot O(\log n) = O(\frac{n \log \sigma}{\sqrt{\log n}})$ bits space for the list. The space of extra sub-blocks is also $\frac{2n}{\log n\, \Theta(\frac{\log n}{\log \sigma})} \cdot \sqrt{\log n}\, \Theta(\frac{\log n}{\log \sigma}) = O(\frac{n \log \sigma}{\sqrt{\log n}})$ bits.

To give $rank_{T_b}$ and $select_{T_b}$, we find the b-th block by traversing the red-black tree in $O(\log n)$ time and scan $O(\log n)$ words in a block by Lemma 2. The in-block updates, $insert_{T_b}$ and $delete_{T_b}$ trigger a carry character to neighbor blocks, and we process the carry by using the extra sub-block of the block. If the extra sub-block is full, we create a new sub-block and add it to the list of the block. When the last two sub-blocks become empty, we remove one empty sub-block in the list.

Lemma 3. *Given an n-length text over a σ-size alphabet with $\sigma \leq \log n$ and its partition of blocks of $\log n$ words, we can support $O(\log n)$ worst-case time $rank_{T_b}$, $select_{T_b}$, $insert_{T_b}$ and $delete_{T_b}$ using $n \log \sigma + O(\frac{n \log \sigma}{\sqrt{\log n}})$ bits space.*

Note that we can process an accessing of the i-th character, $access_T(i)$ in $O(\log n)$ time using the block structures. From $b = rank_I(1, i)$ and $r = i - select_I(1, b)$, we find the b-th block that contains the i-th character and scan its sub-blocks to get the r-th character of the b-th block.

Over-block operations. In Mäkinen and Navarro's binary structure, the over-block operations are provided by storing the number of 1s in the internal nodes of the red-black tree. For a σ-size alphabet, this method cannot obtain $o(n \log \sigma)$ bits space. We propose a new structure for the over-block operations by using the idea of the static structure by Golynski et al. [6] and Hon et al [10].

We first define a character-and-block pair. Given a text T, let $(T[i], b)$ be the character-and-block pair of $T[i]$, where the block number b is given by $rank_I(1, i)$. We define $CB(T)$ as the sorted sequence of $(T[i], b)$. Let (c_j, b_j) denote the j-th pair of $CB(T)$.

We encode $CB(T)$ into a bit vector B of $O(n)$ bits size. $CB(T)$ is a non-decreasing sequence from $(0, 1)$ to $(\sigma - 1, n_b)$, where n_b is the total number of blocks. $CB(T)$ has the consecutive occurrences of a same (c_j, b_j) pair, which means the occurrences of c_j in the b_j-th block. The bit vector B represents the occurrences of (c_j, b_j) as a 1 and the following 0s. If there is no occurrence of (c_j, b_j), it is represented as a single 1. Hence, the k-th 1 of B gives a pair (c, b), where $c = \lfloor \frac{k-1}{n_b} \rfloor$ and $b = (k - 1) \bmod n_b + 1$. Since the block size varies from $\frac{1}{2} \log n$ to $2 \log n$ words, n_b is $\frac{2n}{\log n\, \Theta(\frac{\log n}{\log \sigma})} = O(\frac{n}{\log n})$. For a small alphabet $\sigma \le \log n$, $\frac{n}{\log n} \le \frac{n}{\sigma}$. Then, the size of B is $O(n)$ bits, because the total number of 1s is $\sigma n_b = O(n)$ and the number of 0s is exactly n. The binary operations on B directly supports our over-block operations.

$$
\begin{aligned}
T &= \underline{\mathtt{babc}}\ \underline{\mathtt{ababc}}\ \underline{\mathtt{abca}} \\
I &= \underline{1000}\ \underline{10000}\ \underline{1000} \\
&\quad \underline{(\mathtt{b},1)(\mathtt{a},1)(\mathtt{b},1)(\mathtt{c},1)} \quad \underline{(\mathtt{a},2)(\mathtt{b},2)(\mathtt{a},2)(\mathtt{b},2)(\mathtt{c},2)} \quad \underline{(\mathtt{a},3)(\mathtt{b},3)(\mathtt{c},3)(\mathtt{a},3)} \\
\\
CB(T) &= (\mathtt{a},1)(\mathtt{a},2)(\mathtt{a},2)(\mathtt{a},3)(\mathtt{a},3) \quad (\mathtt{b},1)(\mathtt{b},1)(\mathtt{b},2)(\mathtt{b},2)(\mathtt{b},3) \quad (\mathtt{c},1)(\mathtt{c},2)(\mathtt{c},3) \\
B &= 10100100\ 10010010\ 101010
\end{aligned}
$$

Fig. 1. Example of $CB(T)$ and B

We employ Mäkinen and Navarro's binary rank-select on B which supports all operations in $O(\log n)$ worst-case time. The rank-select on B immediately gives *rank-over* and *select-over*. To update the number of c's in the b-th block, we update 0s after the $(cn_b + b)$-th 1 in B. If the size of the b-th block is out of the range, $\frac{1}{2} \log n$ to $2 \log n$ words, then we split or merge the b-th block in B. We split or merge the b-th block for each character by σ insert-delete operations on B. Since $\sigma \le \log n$, these σ queries can be amortized over $\Theta(\frac{\log^2 n}{\log \sigma}) = \Omega(\sigma)$ in-block operations for each character.

Lemma 4. *Given an n-length text over a σ-size alphabet with $\sigma \le \log n$ and its partition of blocks of* $\log n$ *words, we can answer rank-over and select-over in*

$O(\log n)$ worst-case time and $O(n)$ bits space, while supporting insert-over and delete-over in $O(\log n)$ amortized time.

From Lemmas 3 and 4, we can process the operations for a small alphabet by in-block operations and over-block ones. Given position i, the partition vector I gives the block b and the offset r. Then, $rank_T(c,i)$ is the sum of *rank-over*(c,b) and $rank_{T_b}(c,r)$. $select_T(c,k)$ is the sum of the position of the block given by $b =$ *select-over*(c,k) and $select_{T_b}(c,k')$. For $select_{T_b}(c,k')$, we remove c preceding T_b by $k' = k -$ *rank-over*$(c,b)+1$. $insert_T(c,i)$ is supported by *insert-over*(c,b) and $insert_{T_b}(c,r)$. $delete_T(i)$ is also provided by *delete-over*(c,b) and $delete_{T_b}(r)$, where $c = access_T(i)$.

Note that we fix $\log n$ in the above descriptions. That is, the number of characters in a word is fixed as $\Theta(\frac{\log n}{\log \sigma})$. If n becomes σn or n/σ, then we need to change the block size and the tables of the word operation. In Mäkinen and Navarro's binary structure, partial structures are built for the values $\log n - 1$, $\log n$, and $\log n + 1$ to avoid amortized $O(1)$ update [15]. In this paper, we simply re-build whenever $\log n / \log \sigma$ changes. This is amortized over all update operations and makes $O(1)$ costs per operation.

Lemma 5. *Given an n-length text over a σ-size alphabet with $\sigma \le \log n$, there is a dynamic structure that gives $O(\log n)$ worst-case time access, rank, and select while supporting $O(\log n)$ amortized-time insert and delete in $n \log \sigma + O(\frac{n \log \sigma}{\sqrt{\log n}})$ bits space.*

3.2 Dynamic Rank-Select Structure for a Large Alphabet

From the above $\log n$-alphabet rank-select, we show a dynamic rank-select for a large alphabet, by using a k-ary wavelet-tree [7,5]. Given a text T over a large alphabet with $\sigma > \log n$, we regard $T[i]$ of $\log \sigma$ bits as $\frac{\log \sigma}{\log \log n}$ characters of a $\log n$-size alphabet. We define T^j as the concatenation of the j-th $\log \log n$-bits of $T[i]$ for all i. Let T^j_s denote a subsequence of T^j such that the j-th $\log \log n$ bits of $T[i]$ belongs to T^j_s iff $T[i]$ has the same prefix s of $(j-1)\log \log n$ bits.

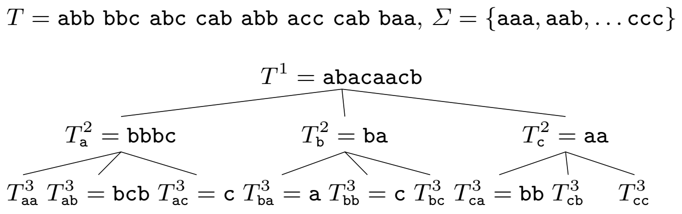


Fig. 2. Example of k-ary wavelet-tree

The k-ary wavelet-tree represents each T^j grouped by the prefix bits. The root of the tree contains T^1 and each of its children contains T^2_c for $c \in \Sigma'$. If a node of the j-th level contains T^j_s, then its children contain T^{j+1}_{s0}, T^{j+1}_{s1}, $\ldots T^{j+1}_{s(\sigma'-1)}$.

At the level $j \leq \frac{\log\sigma}{\log\log n}$, each node T_s^j represents a group of $T[i]$ by the order of prefix s. Then, a branching from T_s^j to T_{sc}^{j+1} is done by counting the number of characters less than c in T_s^j. We use total $O(n)$ bits space for counting the number of c for all T_s^j at level j. The sum of branching space is $O(\frac{n\log\sigma}{\log\log n})$. For rank-select structure, each node stores a rank-select structure on T_s^j instead of T_s^j. Then, the total size of the structure becomes $\frac{\log\sigma}{\log\log n}\cdot(n\log\log n + O(\frac{n\log\log n}{\sqrt{\log n}})) = n\log\sigma + O(\frac{n\log\sigma}{\sqrt{\log n}})$.

Let $\log\sigma$-bits character c be $l = \frac{\log\sigma}{\log\log n}$ characters of $\log\log n$ bits, $c_1c_2\ldots c_l$. Then $rank_T(c,i) = k_l$ and $select_T(c,k) = p_1$ are given by the following steps:

$$\begin{aligned}
k_1 &= rank_{T^1}(c_1, i) & p_l &= select_{T^l_{c_1c_2\cdots c_{l-1}}}(c_l, k)\\
k_2 &= rank_{T^2_{c_1}}(c_2, k_1) & p_{l-1} &= select_{T^{l-1}_{c_1c_2\cdots c_{l-2}}}(c_{l-1}, p_l)\\
&\cdots & &\cdots\\
k_l &= rank_{T^l_{c_1c_2\cdots c_{l-1}}}(c_l, k_{l-1}) & p_1 &= select_{T^1}(c_1, p_2)
\end{aligned}$$

To process $access_T$, we have to find the path of the character $T[i]$ from the root to a leaf. This path starts from $c_1 = access_{T^1}(i)$ and we can find the next node by $rank_{T^1}(c_1, i)$.

$$\begin{aligned}
c_1 &= access_{T^1}(i) & k_1 &= rank_{T^1}(c_1, i)\\
c_2 &= access_{T^2_{c_1}}(k_1) & k_2 &= rank_{T^2_{c_1}}(c_2, k_2)\\
&\cdots & &\cdots\\
c_l &= access^{l-1}_{T_{c_1c_2\cdots c_{l-1}}}(k_{l-1})
\end{aligned}$$

Processing $insert_T$ and $delete_T$ goes through the same step as $access_T$ and updates the character of each level. Finally, we obtain a rank-select structure for a large alphabet.

Lemma 6. *Given an n-length text over a σ-size alphabet with $\sigma > \log n$, there is a dynamic structure that gives $O((1+\frac{\log\sigma}{\log\log n})\log n)$ worst-case time access, rank, and select while supporting $O((1+\frac{\log\sigma}{\log\log n})\log n)$ amortized-time insert and delete in $n\log\sigma + O(\frac{n\log\sigma}{\log\log n})$ bits space.*

4 Rank-Select Structure on RLE

In this section we describe a rank-select structure on $RLE(T)$ by using the structures on (plain) text T' and on bit vectors including L. This method is an extension of RLFM by Mäkinen and Navarro's. For the static rank-select structures on T', we employ Golynski et al.'s structure or the implicit structure which is used in the construction of CSA by Hon et al.

Theorem 2. [6,10] *Given a text T over an alphabet of size σ, there is a succinct rank-select structure that supports $O(\log\log\sigma)$ time rank and $O(1)$ time select, using only $nH_0 + O(n)$ bits space. This structure can be constructed in deterministic $O(n)$ time.*

Our structure in Theorem 1 provides the dynamic rank-select on T'. For the operations on bit vectors, we use a static structure supporting $O(1)$ time rank-select in $O(n)$ bits space by Raman et al. [23] and a dynamic structure providing $O(\log n)$ time operations in $O(n)$ bits space by Mäkinen and Navarro [15]. Note that the total size of a structure on $RLE(T)$ is bounded by the size of a structure on T' plus $O(n)$ bits.

In addition to L, we use some additional bit vectors - a sorted length vector L' and a frequency table F'. A length vector L' is obtained from a sequence of runs in T sorted by the order of characters and the starting position of each run. L' represents lengths of runs in this sorted sequence. The frequency table F' gives $F'[c]$ that is the number of occurrences of characters less than c in text T'. We store F' as a bit vector that represents the number of occurrences of each character as 0s and a following 1. $F'[c]$ can be given by counting the number of 0s up to the c-th 1, i.e., $F'[c] = select_{F'}(1, c) - c$ for $1 \le c < \sigma$ and $F'[0] = 0$. From L' and F', we can also compute $F[c]$ by $select_{L'}(1, F'[c] + 1) - 1$. Then, there is a mapping between the k-th c of T and the $(F[c] + k)$-th position of L'.

$$\begin{array}{rl} T = & \underline{\mathtt{bb}}\ \underline{\mathtt{aa}}\ \underline{\mathtt{bbbb}}\ \underline{\mathtt{cc}}\ \underline{\mathtt{aaa}} \\ L = & \underline{\mathbf{1}0}\ \underline{\mathbf{1}0}\ \underline{\mathbf{1}000}\ \underline{\mathbf{1}0}\ \underline{\mathbf{1}00} \\ T' = & \mathtt{\mathbf{babca}} \end{array} \rightarrow \begin{array}{rl} & \underline{\mathtt{aa}}\ \underline{\mathtt{aaa}}\ \underline{\mathtt{bb}}\ \underline{\mathtt{bbbb}}\ \underline{\mathtt{cc}} \\ L' = & \underline{10}\ \underline{100}\ \underline{\mathbf{1}0}\ \underline{\mathbf{1}000}\ \underline{10} \\ F' = & 00100101 \end{array}$$

Fig. 3. Additional vectors for rank-select on $RLE(T)$

We start from Mäkinen and Navarro's *rank* on RLE and extend RLFM to support *select* and dynamic updates.

Lemma 7. [14] *Given $RLE(T)$, if there are $O(t_S)$ time $select_{T'}$ and $O(t_R)$ time $rank_{T'}$ using $s(T')$ bits space, then we can answer $rank_T(c, i)$ in $O(t_S + t_R) + O(1)$ time and $s(T') + O(n)$ bits.*

Lemma 8. *Given $RLE(T)$, if there is $O(t_S)$ time $select_{T'}$ using $s(T')$ bits space, then we can answer $select_T(c, k)$ in $O(t_S) + O(1)$ time and $s(T') + O(n)$ bits space.*

From Theorem 2 and Lemmas 7, 8, we obtain a static rank-select structure on $RLE(T)$. This structure can be constructed in $O(n)$ time, because binary structures are constructed in linear time by simple scanning of bit vectors. For the compressed full text, the size of RLE structure on $BWT(T)$ is $nH_k(T)H_0(T') + o(nH_0(T')) + O(n)$ by Lemma 1. Note that the structure can be compressed to achieve $nH_k(T)H_k(T')$-bound by applying Sadakane and Grossi's scheme [25].

Theorem 3. *Given $RLE(T)$ of a text T with n' runs, there is a succinct rank-select structure that supports $O(\log\log\sigma)$ time rank and $O(1)$ time select, using $n'H_0(T') + O(n)$ bits space. This structure can be constructed in deterministic $O(n)$ time.*

The dynamic rank-select on $RLE(T)$ is supported by dynamic structures on T', L, L' and F'. Since the retrieving operations, $rank_T$ and $select_T$, are provided by the same steps as in Lemmas 7 and 8, we address only the updating operations, $insert_T(c,i)$ and $delete_T(i)$. For updating $RLE(T)$, we need an additional operation $access_{T'}(i)$ that returns the i-th character, $T'[i]$.

Lemma 9. *Given $RLE(T)$, if there are $O(t_A)$ time $access_{T'}$, $O(t_R)$ time $rank_{T'}$, and $O(t_I)$ time $insert_{T'}$ using $s(T')$ bits space, then we can process $insert_T$ in $O(t_A + t_R + t_I) + O(\log n)$ time and $s(T') + O(n)$ bits space.*

Lemma 10. *Given $RLE(T)$, if there are $O(t_A)$ time $access_{T'}$, $O(t_R)$ time $rank_{T'}$, and $O(t_D)$ time $delete_{T'}$ using $s(T')$ bits space, then we can process $delete_T$ in $O(t_A + t_R + t_D) + O(\log n)$ time and $s(T') + O(n)$ bits space.*

Theorem 4. *Given $RLE(T)$ of a text T with n' runs, there is a dynamic rank-select structure that supports $O((1 + \frac{\log \sigma}{\log \log n}) \log n)$ worst-case time rank and select, while supporting $O((1 + \frac{\log \sigma}{\log \log n}) \log n)$ amortized time insert and delete in $n' \log \sigma + o(n' \log \sigma) + O(n)$ bits space.*

References

1. Chan, H.-L., Hon, W.-K., Lam, T.-W.: Compressed index for a dynamic collection of texts. In: Proceedings of the 15th Annual Symposium on Combinatorial Pattern Matching, pp. 445–456 (2004)
2. Clark, D.R.: Compact Pat Trees. PhD thesis, Univ. Waterloo (1998)
3. Ferragina, P., Luccio, F., Manzini, G., Muthukrishnan, S.: Structuring labeled trees for optimal succinctness, and beyond. In: Proceedings of the IEEE Symposium on Foundations of Computer Science, pp. 184–196 (2005)
4. Ferragina, P., Manzini, G.: Indexing compressed text. Journal of ACM 52(4), 552–581 (2005)
5. Ferragina, P., Manzini, G., Mäkinen, V., Navarro, G.: Succinct representation of sequences and full-text indexes. ACM Transactions on Algorithms (To appear)
6. Golynski, A., Munro, J.I., Rao, S.S.: Rank/select operations on large alphabets: a tool for text indexing. In: Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 368–373 (2006)
7. Grossi, R., Gupta, A., Vitter, J.S.: High-order entropy-compresssed text indexes. In: Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 841–850 (2003)
8. Grossi, R., Vitter, J.S.: Compressed suffix arrays and suffix trees with applications to text indexing and string matching. SIAM Journal on Computing 35(2), 378–407 (2005)
9. Gupta, A., Hon, W.-K., Shah, R., Vitter, J.S.: Dynamic rank/select dictionaries with applications to xml indexing. Manuscript (2006)
10. Hon, W.-K., Sadakane, K., Sung, W.-K.: Breaking a time–and–space barrier in constructing full–text indices. In: Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science, pp. 251–260 (2003)
11. Hon, W.-K., Sadakane, K., Sung, W.-K.: Succinct data structures for searchable partial sums. In: Proceedings of the 14th Annual Symposium on Algorithms and Computation, pp. 505–516 (2003)
12. Jacobson, G.: Space-efficient static trees and graphs. In: Proceedings of the 30th Annual IEEE Symposium on Foundations of Computer Science, pp. 549–554 (1989)

13. Kim, D.-K., Na, J.-C., Kim, J.-E., Park, K.: Fast computation of rank and select functions for succinct representation. Manuscript (2006)
14. Mäkinen, V., Navarro, G.: Succinct suffix arrays based on run-length encoding. In: Proceedings of the 16th Annual Symposium on Combinatorial Pattern Matching, pp. 45–56 (2005)
15. Mäkinen, V., Navarro, G.: Dynamic entropy-compressed sequences and full-text indexes. In: Proceedings of the 17th Annual Symposium on Compinatorial Pattern Matching, pp. 306–317 (2006)
16. Manzini, G.: An analysis of the burrows-wheeler transform. Journal of ACM 48(3), 407–430 (2001)
17. Miltersen, P.B.: Lower bounds on the size of selection and rank indexes. In: Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 11–12 (2005)
18. Munro, J.I.: Tables. In: Proceedings of the 16th Conference on Foundations of Software Technology and Theoretical Computer Science, pp. 37–42 (1996)
19. Munro, J.I., Raman, R., Raman, V., Rao, S.S.: Succinct representations of permutations. In: Proceedings of the 30th International Colloquium on Automata, Language, and Programming, pp. 345–356 (2003)
20. Munro, J.I., Raman, V.: Succinct representations of balanced parentheses and static trees. SIAM Journal on Computing 31(3), 762–776 (2001)
21. Munro, J.I., Rao, S.S.: Succinct representations of functions. In: Proceedings of the 31st International Colloquium on Automata, Languages, and Programming, pp. 1006–1015 (2004)
22. Raman, R., Raman, V., Rao, S.S.: Succinct dynamic data structures. In: Proceedings of the 7th International Workshop on Algorithms and Data Structures, pp. 426–437 (2001)
23. Raman, R., Raman, V., Rao, S.S.: Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In: Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 233–242 (2002)
24. Sadakane, K.: New text indexing functionalites of the compressed suffix arrays. Journal of Algorithms 48(2), 294–313 (2003)
25. Sadakane, K., Grossi, R.: Squeezing succinct data structures into entropy bounds. In: Proceedings of the 17-th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 1230–1239 (2006)

Most Burrows-Wheeler Based Compressors Are Not Optimal

Haim Kaplan and Elad Verbin

School of Computer Science, Tel Aviv University, Tel Aviv, Israel
{haimk,eladv}@post.tau.ac.il

Abstract. We present a technique for proving lower bounds on the compression ratio of algorithms which are based on the *Burrows-Wheeler Transform* (BWT). We study three well known BWT-based compressors: the original algorithm suggested by Burrows and Wheeler; BWT with distance coding; and BWT with run-length encoding. For each compressor, we show a Markov source such that for asymptotically-large text generated by the source, the compression ratio divided by the entropy of the source is a constant greater than 1. This constant is $2 - \epsilon$, 1.26, and 1.29, for each of the three compressors respectively. Our technique is robust, and can be used to prove similar claims for most BWT-based compressors (with a few notable exceptions). This stands in contrast to statistical compressors and Lempel-Ziv-style dictionary compressors, which are long known to be optimal, in the sense that for any Markov source, the compression ratio divided by the entropy of the source asymptotically tends to 1.

We experimentally corroborate our theoretical bounds. Furthermore, we compare BWT-based compressors to other compressors and show that for "realistic" Markov sources they indeed perform bad and often worse than other compressors. This is in contrast with the well known fact that on English text, BWT-based compressors are superior to many other types of compressors.

1 Introduction

In 1994, Burrows and Wheeler [5] introduced the Burrows-Wheeler Transform (BWT) together with two new lossless text-compression algorithms that are based on the BWT. Since then many other BWT-based compressors have emerged. Today, BWT-based compressors are well established alongside with dictionary-based compressors (e.g. the Lempel-Ziv family of compressors, including e.g. gzip) and statistical coders (e.g. PPMd). A widely used implementation of a BWT compressor is bzip2 [19].

BWT-based compressors are typically more effective than dictionary-based compressors, but less effective than statistical coders. For example, bzip2 shrinks the King James version of the Bible to about 21% of its original size while gzip shrinks it to about 29% of the original size, and PPMd to 19% of the original size [1]. See [1, 13, 9] for more experimental results.

In tandem with developing efficient implementations of compression algorithms, analytical upper bounds on compression ratios have been proved. The most famous result of this type is the proof that Huffman codes are the most effective prefix-free codes and, in particular, that the average length of a codeword is close to the entropy of the source (see e.g. [12]).

A compression algorithm is called *optimal* if the algorithm, when applied to a string generated by a Markov source, outputs a number of bits per character which is equal to the entropy of the source, plus a term that tends to 0 as the length of the string tends to infinity. (For a more formal definition see Section 2). It is of particular interest to prove that a compression algorithm is optimal. A famous result of this type is that the Lempel-Ziv-Welch (LZW) compression algorithm [22] is optimal [6, 21]. It is traditionally taken for granted that good compression algorithms should be optimal. In this paper we prove that, surprisingly, most BWT-based compressors are not optimal.

Several papers prove analytical upper bounds on the compression ratio of BWT-based compressors. The bounds found in the papers of Manzini [15] and of Kaplan et al. [13], analyze BWT-based algorithms of the type that is in widespread use today: such algorithms run BWT, followed by a move-to-front encoding or some variant of it, and then apply some type of order-0 encoding (see Section 2 for more information). However, no known upper bound establishes optimality of an algorithm of this type. In Section 3 we show that the reason that optimality was not proved is that these algorithms are in fact not optimal: for each algorithm of this type, there exists a Markov source where the expected number of bits per character that the compressor at stake uses is c times the entropy of the source. Here, $c > 1$ is a constant that depends on which BWT-based algorithm we are considering. For example, for the most well-known algorithm BW_{RL}, which is BWT with run-length encoding, we have $c = 1.297$. For the algorithm $BW0$ first proposed by Burrows and Wheeler, we have $c = 2 - \epsilon$ for any $\epsilon > 0$. For BW_{DC}, which is BWT with distance coding, we get $c = 1.265$. Our technique is quite robust, and can be used to prove non-optimality of most BWT-based compressors.

Two notable exceptions to the above rule are the compression booster of Ferragina et al. [10, 9], and the "block-based" BWT-based compressor of Effros et al. [8]. Both of these compressors have been proved to be optimal. However, none of these compressors is in widespread use today, and both algorithms require further study in order to determine if they are effective in practice, or if they can be made effective in practice. In this paper we do not give an in-depth discussion of these algorithms. For the compression booster, a separate study is in progress to find out whether it can indeed achieve much better compression than other BWT-based compressors, see e.g. [10, 9, 13]. Regarding the block-based compressor of Effros et al., this compressor is proved to be optimal, but only when the block-size is asymptotically large. When the block-size is asymptotically large, the compressor behaves in a way that is not typical of a "classical" BWT-based compressor, and it may behave badly in practice. For constant block-size that is independent of the length of the text, the compressor of Effros et al. is similar to more traditional BWT-based compressors, and Effros et al. state that it is not optimal.

In Section 4 we present experimental evidence that solidify the non-optimality of BWT-based compressors. We begin by applying the compressors to the theoretically bad Markov models, and comparing the compression ratio to the entropy of the source. This confirms our analytical results from Section 3. We briefly outline experiments that were omitted from this version due to lack of space, which show that on texts generated from Markov sources with realistic parameters, the compression ratio divided by the entropy of the source is significantly greater than 1. (We created such "realistic" Markov sources by setting the parameters to values that mimic actual English text).

The results presented in this paper raise an interesting question, which we have not yet fully resolved: if Lempel-Ziv compression is optimal, and BWT-based compression is not optimal, then why does gzip has worse compression ratios than bzip2? (and in general, why do practical BWT-based compressors beat dictionary-based compressors?). The textbook answer is that the proofs of optimality hold for asymptotically large file-sizes, while these compressors, in order to be practical, make some compromises, such as cutting the input into chunks and compressing each separately (in the case of bzip2), or keeping a limited-size dictionary (for gzip). These compromises imply that effectively, in practice, the compressors do not actually operate on large files, and thus the analysis that implies optimality stays largely theoretical. Granted, this is a good answer, but we would like to propose a different answer, which we think should be the subject of further research: We believe that theoretical results that are proved on Markov sources do not have straightforward theoretical implications to English text, since English text was *not* generated by a Markov source. English text has properties that Markov-generated text does not have, such as an abundance of long exact repetitions and approximate repetitions. It is an interesting open question to give a theoretical analysis of compression algorithms that takes into account such differences.

2 Preliminaries

Due to lack of space, we omit the proofs of all claims, except that of Thm. 4. All proofs can be found in the full version of this paper, which is available online.

Let s be the string which we compress, and let Σ denote the alphabet (set of symbols in s). Let $n = |s|$ be the length of 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. We denote by $[h]$ the integers $\{0, \ldots, h-1\}$.

Let $s[i]$ be the i^{th} character in s. The *context of length* k of $s[i]$ is the substring of length k that precedes $s[i]$, i.e. $s[i-k..i-1]$. For a compression algorithm A we denote by $|A(s)|$ the size *in bits* of the output of A on a string s. Note that we always measure the length of strings in characters, and the output of a compression algorithm in bits. The *zeroth order empirical entropy* of the string s is defined as $H_0(s) = \sum_{\sigma \in \Sigma} \frac{n_\sigma}{n} \log \frac{n}{n_\sigma}$. (All logarithms in the paper are to the base 2, and we define $0 \log 0 = 0$). 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. See [15] for a more detailed definition of empirical entropy.

Markov Sources. Let D be a distribution over σ. The *order-zero Markov model* M defined by D is a stochastic process that generates a string by repeatedly picking characters independently according to D.

Let $\{D_w\}$ be a family of distributions over σ, one for each string $w \in \Sigma^k$. The family D_w defines an *order-k Markov model* M over the alphabet Σ. The model M is a stochastic process that generates a string s as follows. The first k characters of s are picked arbitrarily and then each character is generated according to D_w where w is the preceding k characters generated by M.

For a Markov model M we denote by M^n the distribution over Σ^n generated by running M for n steps. In particular, for a string $s \in \Sigma^n$, we denote by $M^n(s)$ the probability that M^n produces s. The *entropy* of M is defined as $H(M) = \lim_{n \to \infty} \frac{1}{n} H(M^n)$, where $H(M^n)$ is the entropy of the distribution M^n, i.e. $H(M^n) = -\sum_{s \in \Sigma^n} M^n(s) \log M^n(s)$.

We remind the reader that Markov models are always discussed asymptotically: we fix a Markov model, and then let n tend to infinity. Thus, the model has "finite memory", while the string's length approaches infinity.

Optimality. A compression algorithm A is called *C-Markov-optimal* if for every Markov model M it holds that $E_{s \in_R M^n}[|A(s)|] \leq CnH(M) + o(n)$. Here, $o(n)$ is a function such that $\frac{o(n)}{n}$ approaches 0 as n tends to infinity. This function may depend arbitrarily on the model M (e.g. it may contain a term exponential in the order of the model). Also, $s \in_R M^n$ denotes that s is a random string chosen according to the distribution M^n. A is called *$(\geq C)$-Markov-optimal* if it is not C'-Markov-optimal for any $C' < C$.[1]

An algorithm A is called *C-w.c.-optimal* (*w.c.* stands for *worst-case*) if for all alphabets Σ, for all $k \geq 0$, for all n large enough, and for all $s \in \Sigma^n$, it holds that $|A(s)| \leq CnH_k(s) + o(n)$. The $o(n)$ term may depend arbitrarily on $|\Sigma|$ and k. An algorithm A is called *$(\geq C)$-w.c.-optimal* if it is not C'-w.c.-optimal for any $C' < C$.

The following connection holds between the notions of Markov-optimality and w.c.-optimality.

Proposition 1. *If A is C-w.c.-optimal then it is C-Markov-optimal. If A is $(\geq C)$-Markov-optimal then it is $(\geq C)$-w.c.-optimal.*

Thus, a lower bound on Markov-optimality is stronger than a lower bound on w.c.-optimality.

[1] Note that if an algorithm is $(\geq C)$-Markov-optimal, this only constitutes a lower bound, and does not imply that the algorithm is optimal in any sense. For example, the algorithm that compresses every string to itself is (≥ 2007)-Markov-optimal, but it is not C-Markov-optimal for any C.

A well-known example of an optimality result is the proof that the LZ77 compression algorithm is 1-Markov-optimal [24]. A similar analysis shows [21] that the LZW compression algorithm is 1-w.c.-optimal. Similar results have been shown for other dictionary-based compressors.

The BWT. Let us define the BWT, which is a function $bwt : \Sigma^n \rightarrow \Sigma^n$. For a string $s \in \Sigma^n$, we obtain $bwt(s)$ 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. Note 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 $bwt(s)$. See an example in Figure 1. Although it may not be obvious at first glance, this function is invertible, assuming that the location of '\$' prior to its omission is known to the inverting procedure. In fact, efficient algorithms exist for computing and inverting bwt in linear time (see for example [16]).

```
mississippi$          $ mississipp i
ississippi$m          i $mississip p
ssissippp$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
$mississippi          s sissippp$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. The string $bwt(s)$ 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 compute the original string. The first and last columns of the matrix on the right hand side are drawn slightly apart from the rest of the matrix in order to highlight them.

Compressors. Let us describe the three compression algorithms based on the BWT that we analyze in this paper: $BW0$, BW_{DC} and BW_{RL}. Many more algorithms exist, but we have chosen these three as representative examples: $BW0$ is the most basic algorithm, and the easiest to analyze. It was the algorithm originally suggested by Burrows and Wheeler in [5]. $BW0$ is conjectured to be 2-w.c.-optimal by [13]. BW_{RL} is the most famous BWT-based algorithm. It is the algorithm that the program bzip2 is based on. It was analyzed by Manzini [15], who proved that it is 5-w.c.-optimal. BW_{DC} was invented but not published by Binder (see [4, 2]), and is described in a paper of Deorowicz [7]. BW_{DC} is 1.7286-w.c.-optimal [13].

The algorithm $BW0$ compresses the input text s in three steps.

1. Compute $bwt(s)$.
2. Transform $bwt(s)$ to a string of integers $\dot{s} = mtf(bwt(s))$ by using the move to front algorithm, originally introduced in [3]. 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).[2]
3. Encode the string $\dot{s}$ by using Arithmetic Coding, to obtain the final string of bits $BW0(s) = Arith(\dot{s})$. We do not describe Arithmetic Coding here. The interested reader is referred to [23, 17]. All we use is that for any string s, $|s| H_0(s) \leq |Arith(s)| \leq |s| H_0(s) + O(\log |s|)$.[3]

We denote this algorithm by $BW0 = BWT + MTF + Arith$. Here, '+' is the *composition* operator for compression algorithms, and $Arith$ stands for Arithmetic Coding.

Intuitively, $BW0$ is a good compressor, since in $bwt(s)$ characters with the same context appear consecutively. Therefore, if s is, say, a text in English, we expect $bwt(s)$ to be a string with symbols recurring at close vicinity. As a result $\dot{s} = mtf(bwt(s))$ 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). Therefore, 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. Manzini [15] has formalized this intuition to get an upper bound on the compression ratio of $BW0$. This upper bound was improved and simplified in [13].

The algorithm $BW_{DC} = BWT + DC + Arith$ works like $BW0$, except that MTF is replaced by a transformation called *distance coding* (DC). The transformation DC encodes the character $s[i] = \sigma$ with an integer equal to the distance to the previous occurrence of σ in s, but only if this distance is not zero. When the distance is 0 (i.e. when $s[i-1] = \sigma$), we simply skip this character. Therefore, $DC(s)$ is potentially a much shorter string than s. However, $DC(s)$ might be a string over a larger alphabet, since the distances might be as big as n. (It is

[2] To completely determine the encoding we must specify the status of the recency list at the beginning of the procedure. For our purposes, it is enough to assume that in the initial recency list the characters are sorted by any arbitrary order, e.g. their ASCII order.

[3] This upper bound follows from the analysis of [23], but it is not proved anywhere in the literature, since it holds only for a costly version of Arithmetic Coding that is not used in practice. The versions of Arithmetic Coding used in practice actually have, say, $|Arith(s)| \leq |s| H_0(s) + 10^{-2} |s| + O(\log |s|)$ and are more efficient in terms of running time.

not hard to see that the number of different integers that occur in $DC(s)$ is at most $h\sqrt{2n}$).[4] To be able to decode DC, we also need to store some auxiliary information consisting of the positions of the first and last occurrence of each character. Since this additional information takes at most $2h \log n = O(\log n)$ bits we disregard this from now on. It is not hard to see that from $DC(s)$ and the auxiliary information we can recover s (see [13]). We observe that DC produces one integer per run of s. Furthermore, when s is binary, then this integer is equal to the length of the run.

The algorithm $BW_{RL} = BWT + MTF + RLE + Arith$ works like $BW0$, with an additional *run-length encoding* (RLE) step that we perform on the output of MTF before the Arithmetic Coding. The particular way RLE is implemented varies in different papers. The general idea is to encode each run as a separate character or sequence of characters so that a long run of, say, m 0's, can be represented using, say, $\log m$ characters in a new alphabet. Afterwards, we perform Arithmetic Coding on a string defined over this new alphabet. A standard version of RLE is described by Manzini [15].

3 Lower Bounds

We first state two lemmas that we use for proving our lower bounds. The first lemma roughly states that if a compression algorithm A performs badly on an order-zero Markov model then $BWT + A$ performs badly on the same model. This gives us a way to derive a lower bound on the optimality ratio of $BWT + A$ from a lower bound on the optimality ratio of A on order-0 Markov models.

Lemma 2. *Suppose that* $Pr_{s \in_R M^n}[|A(s)| > CnH(M)] > 1 - 1/n^2$*, where* M *is an order-zero Markov model. Then,* $E_{s \in_R M^n}[|A(BWT(s))|] \geq CnH(M) - O(1)$*. In particular, the compression algorithm* $BWT + A$ *is* $(\geq C)$*-Markov-optimal.*

Proof Sketch. The rough intuition behind the proof of this lemma, is that at least a $\frac{1}{n+1}$-fraction of strings are in the image of the function *bwt* (we call these strings *bwt-images*). The condition of the lemma states that all but a $\frac{1}{n^2}$-fraction of strings are hard to compress. Therefore, via a simple counting argument, we see that a vast majority of the bwt-images are hard to compress, and this gives the lemma. Special care must be taken with measuring what "a vast majority of the bwt-images" means, since some strings are bwt-images of only one string, while some may be bwt-images of $n + 1$ strings. This point accounts for most of the technical difficulty of the lemma.

The second lemma bounds the number of runs in binary strings generated by an order-zero Markov model. We use it to analyze the behavior of algorithms such as $MTF + Arith$ on order-zero Markov models.

[4] This is a simplified version of the transformation described by [4, 2, 7]. Originally, two more properties were used to discard some more characters from $DC(s)$ and make the numbers that do appear in $DC(s)$ smaller. However, we prove the lower bound using a binary alphabet, where the additional properties do not matter.

Here, by *substring* we mean a consecutive substring, and a *run* in a string s is a maximal substring consisting of identical characters. Also, denote by M_p the zero-order Markov model that outputs 'a' with probability p, and 'b' with probability $1-p$.

Lemma 3. *For any $0 < p \leq \frac{1}{2}$ and any constant $c \geq 1$, for all large enough n, a string s chosen from M_p^n has, with probability at least $1 - 1/n^2$, both of the following properties:*

1. *The number of runs in s is between $2p(1-p)(n-n^{2/3})$ and $2p(1-p)(n+n^{2/3})$.*
2. *For every $1 \leq k \leq c$, the number of runs of 'a' of length exactly k is between $(1-p)^2p^k(n-n^{2/3})$ and $(1-p)^2p^k(n+n^{2/3})$, and the number of runs of 'b' of length exactly k is between $p^2(1-p)^k(n-n^{2/3})$ and $p^2(1-p)^k(n+n^{2/3})$.*

3.1 $BW0$ is (≥ 2)-Markov-Optimal

We now prove that the algorithm $BW0 = BWT + MTF + Arith$ is (≥ 2)-Markov-optimal. Let $\epsilon > 0$ be some positive number. It is enough to prove that $BW0$ is ($\geq 2 - \epsilon$)-Markov-optimal. Let $A0 = MTF + Arith$. By Lemma 2, it is enough to show an order-zero Markov model M such that for large enough n,

$$Pr_{s \in_R M^n}[|A0(s)| > (2-\epsilon)nH(M)] > 1 - 1/n^2 \ .$$

Theorem 4. *For every $\epsilon > 0$ there exists p such that for large enough n,*

$$Pr_{s \in_R M_p^n}[|A0(s)| > (2-\epsilon)nH(M_p)] > 1 - 1/n^2 \ .$$

Therefore $BW0$ is (≥ 2)-Markov-optimal.

Proof. Let $s \in \Sigma_2^n$ be a string with r runs. Then the string $MTF(s)$ contains r '1's and $n-r$ '0's, and thus $|A0(s)| \geq r \log \frac{n}{r} + (n-r)\log\frac{n}{n-r} \geq r\log\frac{n}{r}$. We prove that $r\log\frac{n}{r} > (2-\epsilon)nH(M_p)$ for $r = 2p(1-p)(n \pm n^{2/3})$. By Lemmas 2 and 3 this gives the theorem. In fact, we prove that $r\log\frac{n}{r} > (2-\epsilon/2)nH(M_p)$ for $r = 2p(1-p)n$. The term $\pm n^{2/3}$ is "swallowed" by the slack of $\epsilon/2$ when n is large enough.
To prove that $\frac{r\log\frac{n}{r}}{nH(M_p)} > 2 - \epsilon/2$ first note that

$$\frac{r\log\frac{n}{r}}{nH(M_p)} = \frac{2p(1-p)n\log\frac{1}{2p(1-p)}}{np\log\frac{1}{p} + n(1-p)\log\frac{1}{1-p}} \ .$$

This expression approaches 2 from below when p tends to 0, since

$$\frac{2p(1-p)\log\frac{1}{2p(1-p)}}{p\log\frac{1}{p} + (1-p)\log\frac{1}{1-p}} = \frac{2p(1-p)\log\frac{1}{p} - \Theta(p)}{p\log\frac{1}{p} + \Theta(p)} \xrightarrow{p\to 0} 2 \ .$$

Thus, we can choose p to be a function of ϵ (but not of n), such that this expression is larger than $2 - \epsilon/2$. □

This gives a positive answer to Conjecture 20 in [13]. Also, it was shown there that for binary alphabet, $BW0$ is 2-w.c.-optimal. Therefore, the lower bound we prove here is tight. In [13] it was also conjectured that $BW0$ is 2-w.c.-optimal for alphabets of any size. We have not been able to prove this so far.

3.2 BW_{DC} Is ($\geq$ 1.265)-Markov-Optimal, BW_{RL} Is ($\geq$ 1.297)-Markov-Optimal

In the full version of the paper we also prove that BW_{DC} is ($\geq$ 1.265)-Markov-optimal. Due to Lemma 2, in order to prove this it suffices to prove that the algorithm $A_{DC} = DC + Arith$ performs badly on an order-zero Markov model. The Markov model that was used in the last section, which had infinitesimally small probability for producing 'a', will not work here. A_{DC} works quite well for such Markov models. Instead, we use Lemma 3 to get exact statistics (up to lower-order terms) for the character frequencies in $DC(s)$ where s is taken from the Markov model M_p, and we compute the entropy according to these character frequencies. Using a similar technique we prove that BW_{RL} is ($\geq$ 1.297)-Markov-optimal.

4 Experiments

We confirmed experimentally that our lower bounds indeed hold. For this purpose, we have coded the three algorithms $BW0$, BW_{DC}, and BW_{RL}. Unlike bzip2 which cuts the file into chunks of 900K before compressing, our programs compress the whole file in one piece.

Rather than using Arithmetic Coding on the integer sequence, as the final stage of the compression algorithm, our implementations computed the empirical entropy of the integer sequence. For any string this entropy is a lower bound on the size of the Arithmetic Encoding of that string. This made our experiment cleaner since our focus is on lower bounds. We also did not consider auxiliary data that one needs to save in order to decompress the file, such as the initial status of the *mtf* recency list, the location of the end-of-string symbol after *bwt*, etc. We refer to our implementations as "clean" versions of the respective algorithms.

To compare BWT-based compressors against dictionary-based compressors we also implemented a clean version of the LZW algorithm [22]. This implementation follows closely the theoretical version of LZW described in [22]. In particular it does not have an upper bound on the dictionary size. The compressed representation consists of a sequence of pointers (integer values) to the dictionary and a sequence of characters (the list of addition symbols). We compute the entropy of each of these sequences. The sum of both entropies is the "compressed size" of the file according to our clean LZW implementation. Note that the sequence of pointers is a string defined over a potentially very large alphabet. Therefore to arithmetically code this string one would need to store a very large table which is not realistic. We believe however that the entropy of this string does reflect in a clean way the size of the LZW encoding.

We have generated 100 files of size 10 Megabytes from the source M_p for various values of p. For each compressor and each value of p, we compute the *optimality ratio*, which is the ratio between the number of bits per character that the compressor used and the entropy of the source. These results are given in Table 1.

First notice that the performance of $BW0$, BW_{DC}, and BW_{RL} follow closely the theoretical results of Section 3. In particular for $p = 0.001$ the optimality ratio of $BW0$ is 1.823. (A calculation, omitted due to lack of space, shows that this ratio is exactly equal to the theoretical prediction). For $p = 0.051$ the optimality ratio of BW_{DC} is 1.266, and for $p = 0.06$ the optimality ratio of BW_{RL} is 1.298. Furthermore as predicted by our analysis the optimality ratio of $BW0$ increases as p goes to zero.

The optimality ratio of our "clean" implementation of LZW increases when p decreases. This effect is related to results of Kosaraju and Manzini [14]. Recall, that the optimality ratio of LZW on M_p tends to 1 as n tends to infinity. So it is surprising, to see that in our experiments the optimality ratio of LZW may get as large as 1.53, and is always at least 1.11. This indicates that the optimality ratio of LZW tends to 1 quite slowly, and is quite far from 1 for the 10MB strings that we compressed. Also, note that for some values of p, the optimality ratio of LZW is worse than that of the BWT-based compressors.

We have also checked how the optimality ratios of the algorithms change when we fix the value of p, and increase the file size. (The experimental data is not included due to lack of space. It can be found in the full version of the paper). The optimality ratio of each of the BWT-based compressors stays exactly the same when the file size grows. The optimality ratio of LZW, on the other hand, tends to 1 as n grows, but it decreases slowly. This can be explained if we recall that the optimality ratio of LZW is $1+O\left(\frac{1}{\log n}\right)$, see [18]. Note that the constant hidden by the O-notation depends on the parameters of the Markov model, see [24, 14]. (In our case, it depends on p). The low-order $O\left(\frac{1}{\log n}\right)$ term decays slowly. For example, for $n = 10^7$, $\frac{1}{\log n} \approx 0.04$. Nonetheless, for any fixed p, there exists a large enough value of n for which LZW beats the BWT-based compressors for strings of length n drawn from M_p. Thus, we can say that LZW *asymptotically beats* BWT-compressors on Markov models.

We did not consider statistical compressors in any of the experiments here. Intuitively, for an order-zero Markov model, such a compressor is simply Arithmetic Coding (or the "clean" version thereof), which has a constant optimality ratio of 1, no matter what n is. We have continued to see that statistical compressors are the best among all compressors we study, when working over Markov models.

Real Compressors. We have also performed a similar experiment using the "real" compressors *bzip2* [19], *gzip* [11], and *ppmdi* [20]. These compressors seem to work like their theoretical counterparts, with some interesting exceptions. We include these experimental results in the full version of the paper.

Experiments on Realistic High-Order Models. We have also performed experiments on high-order Markov models that were generated using realistic parameters extracted from English text. We found that BWT-based compressors behave quite badly on such Markov models. These experimental results are included in the full version of the paper.

Table 1. The optimality ratio of the clean compressors, measured on texts generated from the Markov source M_p. Each experiment was run on 100 samples, each of size 10MB. The standard deviations are not listed per-entry, but the maximal standard deviation among all those in the row is listed in the last column (the standard deviations tend to be similar for values in the same row).

p	clean BW0	clean BW_{DC}	clean BW_{RL}	clean LZW	σ
0.001	1.823	1.141	1.143	1.535	0.016
0.005	1.772	1.210	1.216	1.490	0.006
0.01	1.737	1.232	1.242	1.347	0.004
0.051	1.579	1.266	1.296	1.211	0.0012
0.06	1.552	1.265	1.298	1.202	0.0011
0.1	1.450	1.250	1.286	1.171	0.0008
0.2	1.253	1.176	1.202	1.137	0.0004
0.3	1.114	1.093	1.103	1.124	0.0002
0.4	1.029	1.027	1.028	1.113	0.00013
0.5	1.000	1.000	1.000	1.113	0.00004

5 Conclusion

In this paper we have shown lower bounds on the optimality ratio of BWT-based compressors on Markov sources. This suboptimal performance on Markov sources is corroborated by experiments with Markov sources that imitate an English text.

On the other hand it is known that on English text, BWT-based compressors work extremely well. We believe that BWT-compressors work on English text better than Dictionary-based compressors (and not much worse than statistical compressors) because of non-Markovian elements in English text.

This discrepancy between the performance of BWT-based compressors on Markov sources that resemble an English text and their performance on the text itself is yet to be explored. What kind of regularity is there in English text that compressors exploit?

References

[1] The Canterbury Corpus `http://corpus.canterbury.ac.nz`

[2] Abel, J.: Web page about Distance Coding `http://www.data-compression.info/Algorithms/DC/`

[3] Bentley, J.L., Sleator, D.D., Tarjan, R.E., Wei, V.K.: A locally adaptive data compression scheme. Communications of the ACM 29(4), 320–330 (1986)

[4] Binder, E.: Distance coder. Usenet group comp.compression (2000)

[5] Burrows, M., Wheeler, D.J.: A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, Palo Alto, California (1994)

[6] Cover, T.M., Thomas, J.A.: Elements of Information Theory. John Wiley & sons, New York (1991)

[7] Deorowicz, S.: Second step algorithms in the Burrows–Wheeler compression algorithm. Software–Practice and Experience 32(2), 99–111 (2002)
[8] Effros, M., Visweswariah, K., Kulkarni, S., Verdu, S.: Universal lossless source coding with the Burrows Wheeler transform. IEEE Transactions on Information Theory 48(5), 1061–1081 (2002)
[9] Ferragina, P., Giancarlo, R., Manzini, G.: The engineering of a compression boosting library: Theory vs practice in BWT compression. In: Azar, Y., Erlebach, T. (eds.) ESA 2006. LNCS, vol. 4168, pp. 756–767. Springer, Heidelberg (2006)
[10] Ferragina, P., Giancarlo, R., Manzini, G., Sciortino, M.: Boosting textual compression in optimal linear time. Journal of the ACM 52, 688–713 (2005)
[11] Gailly, J., Adler, M.: The gzip compressor http://www.gzip.org/
[12] Gallager, R.: Variations on a theme by Huffman. IEEE Transactions on Information Theory 24(6), 668–674 (1978)
[13] Kaplan, H., Landau, S., Verbin, E.: A simpler analysis of Burrows-Wheeler based compression. To be puiblished in Theoretical Computer Science, special issue on the Burrows-Wheeler Transform and its Applications, Preliminary version published in CPM '06 (2007)
[14] Kosaraju, S.R., Manzini, G.: Compression of low entropy strings with Lempel-Ziv algorithms. SIAM J. Comput. 29(3), 893–911 (1999)
[15] Manzini, G.: An analysis of the Burrows-Wheeler Transform. Journal of the ACM 48(3), 407–430 (2001)
[16] Manzini, G., Ferragina, P.: Engineering a lightweight suffix array construction algorithm. Algorithmica 40, 33–50 (2004)
[17] Moffat, A., Neal, R.M., Witten, I.H.: Arithmetic coding revisited. ACM Transactions on Information Systems 16(3), 256–294 (1998)
[18] Savari, S.A.: Redundancy of the Lempel-Ziv-Welch code. In: Proc. Data Compression Conference (DCC), pp. 191–200 (1997)
[19] Seward, J.: bzip2, a program and library for data compression http://www.bzip.org/
[20] Shkarin, D., Cheney, J.: ppmdi, a statistical compressor. This is Shkarin's compressor PPMII, as modified and incorporated into XMLPPM by Cheney, and then extracted from XMLPPM by Adiego. J
[21] Shor, P.: Lempel-Ziv compression (lecture notes for the course principles of applied mathematics) www-math.mit.edu/~shor/PAM/lempel_ziv_notes.pdf
[22] Welch, T.A.: A technique for high-performance data compression. Computer 17, 8–19 (1984)
[23] Witten, I.H., Neal, R.M., Cleary, J.G.: Arithmetic coding for data compression. Communications of the ACM 30(6), 520–540 (1987)
[24] Wyner, A.D., Ziv, J.: The sliding-window Lempel-Ziv algorithm is asymptotically optimal. Proc. IEEE 82(8), 872–877 (1994)

Non-breaking Similarity of Genomes with Gene Repetitions*

Zhixiang Chen[1], Bin Fu[1], Jinhui Xu[2], Boting Yang[3], Zhiyu Zhao[4], and Binhai Zhu[5]

[1] Department of Computer Science, University of Texas-American, Edinburg, TX 78739-2999, USA
chen, binfu@cs.panam.edu
[2] Department of Computer Science, SUNY-Buffalo, Buffalo, NY 14260, USA
jinhui@cse.buffalo.edu
[3] Department of Computer Science, University of Regina, Regina, Saskatchewan, S4S 0A2, Canada
boting@cs.uregina.ca
[4] Department of Computer Science, University of New Orleans, New Orleans, LA 70148, USA
zzha2@cs.uno.edu.
[5] Department of Computer Science, Montana State University, Bozeman, MT 59717-3880, USA
bhz@cs.montana.edu.

Abstract. In this paper we define a new similarity measure, the *non-breaking similarity*, which is the complement of the famous breakpoint distance between genomes (in general, between any two sequences drawn from the same alphabet). When the two input genomes $\mathcal{G}$ and $\mathcal{H}$, drawn from the same set of n gene families, contain gene repetitions, we consider the corresponding Exemplar Non-breaking Similarity problem (ENbS) in which we need to delete repeated genes in $\mathcal{G}$ and $\mathcal{H}$ such that the resulting genomes G and H have the maximum non-breaking similarity. We have the following results.

- For the Exemplar Non-breaking Similarity problem, we prove that the Independent Set problem can be linearly reduced to this problem. Hence, ENbS does not admit any factor-$n^{1-\epsilon}$ polynomial-time approximation unless P=NP. (Also, ENbS is W[1]-complete.)
- We show that for several practically interesting cases of the Exemplar Non-breaking Similarity problem, there are polynomial time algorithms.

1 Introduction

In the genome comparison and rearrangement area, the breakpoint distance is one of the most famous distance measures [15]. The implicit idea of breakpoints was initiated as early as in 1936 by Sturtevant and Dobzhansky [14].

* This research is supported by Louisiana Board of Regents under contract number LEQSF(2004-07)-RD-A-35, NSF grant CCF-0546509, NSERC grant 261290-03 and Montana EPSCOR's Visiting Scholar's Program.

Until a few years ago, in genome rearrangement research, it is always assumed that every gene appears in a genome exactly once. Under this assumption, the genome rearrangement problem is in essence the problem of comparing and sorting signed/unsigned permutations [10,11]. In the case of breakpoint distance, given two perfect genomes (in which every gene appears exactly once, i.e., there is no gene repetition) it is easy to compute their breakpoint distance in linear time.

However, perfect genomes are hard to obtain and so far they can only be obtained in several small virus genomes. For example, perfect genomes do not occur on eukaryotic genomes where paralogous genes are common [12,13]. On the one hand, it is important in practice to compute genomic distances, e.g., using Hannenhalli and Pevzner's method [10], when no gene duplication arises; on the other hand, one might have to handle this gene duplication problem as well. In 1999, Sankoff proposed a way to select, from the duplicated copies of genes, the common ancestor gene such that the distance between the reduced genomes (*exemplar genomes*) is minimized [13]. A general branch-and-bound algorithm was also implemented in [13]. Recently, Nguyen, Tay and Zhang proposed using a divide-and-conquer method to compute the exemplar breakpoint distance empirically [12].

For the theoretical part of research, it was shown that computing the exemplar signed reversal and breakpoint distances between (imperfect) genomes are both NP-complete [1]. Two years ago, Blin and Rizzi further proved that computing the exemplar conserved interval distance between genomes is NP-complete [2]; moreover, it is NP-complete to compute the minimum conserved interval matching (i.e., without deleting the duplicated copies of genes). In [6,3] it was shown that the exemplar genomic distance problem does not admit any approximation (regardless of the approximation factor) unless P=NP, as long as $G = H$ implies that $d(G, H) = 0$, for any genomic distance measure $d(\ )$. This implies that for the exemplar breakpoint distance and exemplar conserved interval distance problems, there are no polynomial-time approximations. In [6] it was also shown that even under a weaker definition of (polynomial-time) approximation, the exemplar breakpoint distance problem does not admit any weak approximation of factor $n^{1-\epsilon}$ for any $0 < \epsilon < 1$, where n is the maximum length of the input genomes. In [3,4] it was shown that under the same definition of weak approximation, the exemplar conserved interval distance problem does not admit any weak approximation of a factor which is superlinear (roughly $n^{1.5}$).

In [5] three new kinds of genomic similarities were considered. These similarity measures, which are not distance measures, do not satisfy the condition that $G = H$ implies that $d(G, H) = 0$. Among them, the exemplar common interval measure problem seems to be the most interesting one. When gene duplications are allowed, Chauve, *et al.* proved that the problem is NP-complete and left open a question whether there is any inapproximability result for it.

In this paper, we define a new similarity measure called *non-breaking similarity*. Intuitively, this is the complement of the traditional breakpoint distance measure. Compared with the problem of computing exemplar breakpoint distance, which is a minimization problem, for the exemplar non-breaking similarity problem we need to maximize the number of non-breaking points. Unfortunately we show in this paper that Independent Set can be reduced to ENbS; moreover, this reduction implies that ENbS is W[1]-complete (and ENbS does not have a factor-n^ϵ polynomial-time approximation). This reduction works even when one of the two genomes is given exemplar.

The W[1]-completeness (see [8] for details) and the recent lower bound results [7] imply that if k is the optimal solution value, unless an unlikely collapse occurs in the parameterized complexity theory, ENbS is not solvable in time $f(k)n^{o(k)}$, for any function f. However, we show that for several practically interesting cases of the problem, there are polynomial time algorithms. This is done by parameterizing some quantities in the input genomes, followed with some traditional algorithmic techniques.

This effort is not artificial: in real-life datasets, usually there are some special properties in the data. For example, as reported in [12], the repeated genes in some bacteria genome pairs are often pegged, i.e., the repeated genes are usually separated by a peg gene which occurs exactly once. Our solution can help solving cases like these, especially when the number of such repeated genes is limited.

This paper is organized as follows. In Section 2, we go over the necessary definitions. In Section 3, we reduce Independent Set to ENbS, hence showing the inapproximability result. In Section 4, we present polynomial time algorithms for several practically interesting cases. In Section 5, we conclude the paper with some discussions.

2 Preliminaries

In the genome comparison and rearrangement problem, we are given a set of genomes, each of which is a signed/unsigned sequence of genes[1]. The order of the genes corresponds to the position of them on the linear chromosome and the signs correspond to which of the two DNA strands the genes are located. While most of the past research are under the assumption that each gene occurs in a genome once, this assumption is problematic in reality for eukaryotic genomes or the likes where duplications of genes exist [13]. Sankoff proposed a method to select an *exemplar genome*, by deleting redundant copies of a gene, such that in an exemplar genome any gene appears exactly once; moreover, the resulting exemplar genomes should have a property that certain genomic distance between them is minimized [13].

The following definitions are very much following those in [1,6]. Given n *gene families* (alphabet) $\mathcal{F}$, a genome $\mathcal{G}$ is a sequence of elements of $\mathcal{F}$. (Throughout this paper, we will consider unsigned genomes, though our results can be applied to signed genomes as well.) In general, we allow the repetition of a gene family in any genome. Each occurrence of a gene family is called a *gene*, though we will not try to distinguish a gene and a gene family if the context is clear.

[1] In general a genome could contain a set of such sequences. The genomes we focus on in this paper are typically called *singletons*.

The number of a gene g appearing in a genome $\mathcal{G}$ is called the occurrence of g in $\mathcal{G}$, written as $occ(g, \mathcal{G})$. A genome $\mathcal{G}$ is called *r-repetitive*, if all the genes from the same gene family occur at most r times in $\mathcal{G}$. For example, if $\mathcal{G} = abcbaa$, $\text{occ}(b, \mathcal{G}) = 2$ and $\mathcal{G}$ is a 3-repetitive genome.

For a genome $\mathcal{G}$, alphabet($\mathcal{G}$) is the set of all the characters (genes) that appear at least once in $\mathcal{G}$. A genome G is an exemplar genome of $\mathcal{G}$ if $\text{alphabet}(G) = \text{alphabet}(\mathcal{G})$ and each gene in alphabet($\mathcal{G}$) appears exactly once in G; i.e., G is derived from $\mathcal{G}$ by deleting all the redundant genes (characters) in $\mathcal{G}$. For example, let $\mathcal{G} = bcaadage$ there are two exemplar genomes: $bcadge$ and $bcdage$.

For two exemplar genomes G and H such that $\text{alphabet}(G) = \text{alphabet}(H)$ and $|\text{alphabet}(G)| = |\text{alphabet}(H)| = n$, a breakpoint in G is a two-gene substring $g_i g_{i+1}$ such that $g_i g_{i+1}$ is not a substring in H. The number of breakpoints in G (symmetrically in H) is called the *breakpoint distance*, denoted as $\text{bd}(G, H)$. For two genomes $\mathcal{G}$ and $\mathcal{H}$, their *exemplar breakpoint distance* $\text{ebd}(\mathcal{G}, \mathcal{H})$ is the minimum $\text{bd}(G, H)$, where G and H are exemplar genomes derived from $\mathcal{G}$ and $\mathcal{H}$.

For two exemplar genomes G and H such that $\text{alphabet}(G) = \text{alphabet}(H)$ $|\text{alphabet}(G)| = |\text{alphabet}(H)| = n$, a *non-breaking point* is a common two-gene substring $g_i g_{i+1}$ that appears in both G and H. The number of non-breaking points between G and H is also called the *non-breaking similarity* between G and H, denoted as $\text{nbs}(G, H)$. Clearly, we have $\text{nbs}(G, H) = n - 1 - \text{bd}(G, H)$. For two genomes $\mathcal{G}$ and $\mathcal{H}$, their *exemplar non-breaking similarity* $\text{enbs}(\mathcal{G}, \mathcal{H})$ is the maximum $\text{nbs}(G, H)$, where G and H are exemplar genomes derived from $\mathcal{G}$ and $\mathcal{H}$. Again we have $\text{enbs}(\mathcal{G}, \mathcal{H}) = n - 1 - \text{ebd}(\mathcal{G}, \mathcal{H})$.

The Exemplar Non-breaking Similarity (ENbS) Problem is formally defined as follows:

Instance: Genomes $\mathcal{G}$ and $\mathcal{H}$, each is of length $O(m)$ and each covers n identical gene families (i.e., at least one gene from each of the n gene families appears in both $\mathcal{G}$ and $\mathcal{H}$); integer K.
Question: Are there two respective exemplar genomes of $\mathcal{G}$ and $\mathcal{H}$, G and H, such that the non-breaking similarity between them is at least K?

In the next two sections, we present several results for the optimization versions of these problems, namely, to compute or approximate the maximum value K in the above formulation. Given a maximization problem Π, let the optimal solution of Π be OPT. We say that an approximation algorithm $\mathcal{A}$ provides a *performance guarantee* of α for Π if for every instance I of Π, the solution value returned by $\mathcal{A}$ is at least OPT/α. (Usually we say that $\mathcal{A}$ is a factor-α approximation for Π.) Typically we are interested in polynomial time approximation algorithms.

3 Inapproximability Results

For the ENbS problem, let O_{ENbS} be the corresponding optimal solution value. First we have the following lemma.

Lemma 1. $0 \leq O_{ENbS} \leq n-1$.

Proof. Let the n gene families be denoted by $1, 2, ..., n$. We only consider the corresponding exemplar genomes G, H. The lower bound of O_{ENbS} is achieved by setting $G = 123 \cdots (n-1)n$ and H can be set as follows: when n is even, $H = (n-1)(n-3) \cdots 531n(n-2) \cdots 642$; when n is odd, $H = (n-1)(n-3) \cdots 642n135 \cdots (n-4)(n-2)$. It can be easily proved that between G, H there is no non-breaking point. The upper bound of O_{ENbS} is obtained by setting $G = H$ in which case any two adjacent genes form a non-breaking point. □

The above lemma also implies that different from the Exemplar Breakpoint Distance (EBD) problem, which does not admit any polynomial-time approximation at all (as deciding whether the optimal solution value is zero is NP-complete), the same cannot be said on ENbS. Given $\mathcal{G}$ and $\mathcal{H}$, it can be easily shown that deciding whether $O_{ENbS} = 0$ can be done in polynomial time (hence it is easy to decide whether there exists some approximation for ENbS—for instance, as $O_{ENbS} \leq n-1$, if we can decide that $O_{ENbS} \neq 0$ then it is easy to obtain a factor-$O(n)$ approximation for ENbS). However, the next theorem shows that even when one of $\mathcal{G}$ and $\mathcal{H}$ is given exemplar, ENbS still does not admit a factor-$n^{1-\epsilon}$ approximation.

Theorem 1. *If one of $\mathcal{G}$ and $\mathcal{H}$ is exemplar and the other is 2-repetitive, the Exemplar Non-breaking Similarity Problem does not admit any factor $n^{1-\epsilon}$ polynomial time approximation unless P=NP.*

Proof. We use a reduction from Independent Set to the Exemplar Non-breaking Similarity Problem in which each of the n genes appears in $\mathcal{G}$ exactly once and in $\mathcal{H}$ at most twice. Independent Set is a well known NP-complete problem which cannot be approximated within a factor of $n^{1-\epsilon}$ [9].

Given a graph $T = (V, E), V = \{v_1, v_2, \cdots, v_N\}, E = \{e_1, e_2, \cdots, e_M\}$, we construct $\mathcal{G}$ and $\mathcal{H}$ as follows. (We assume that the vertices and edges are sorted by their corresponding indices.) Let A_i be the sorted sequence of edges incident to v_i. For each v_i we add v_i' as an additional gene and for each e_i we add x_i, x_i' as additional genes. We have two cases: $N + M$ is even and $N + M$ is odd. We mainly focus on the case when $N + M$ is even. In this case, the reduction is as follows.

Define $Y_i = v_i A_i v_i'$, if $i \leq N$ and $Y_{N+i} = x_i x_i'$, if $i \leq M$.

$\mathcal{G} : v_1 v_1' v_2 v_2' \cdots v_N v_N' x_1 e_1 x_1' x_2 e_2 x_2' \cdots x_M e_M x_M'$.

$\mathcal{H} : Y_{N+M-1} Y_{N+M-3} \cdots Y_1 Y_{N+M} Y_{N+M-2} \cdots Y_2$.

(Construct $\mathcal{H}$ as $Y_{N+M-1} Y_{N+M-3} \cdots Y_2 Y_{N+M} Y_1 Y_3 \cdots Y_{N+M-2}$ when $N + M$ is odd. The remaining arguments will be identical.)

We claim that T has an independent set of size k iff the exemplar non-breaking similarity between $\mathcal{G}$ and $\mathcal{H}$ is k. Notice that $\mathcal{G}$ is already an exemplar genome, so $G = \mathcal{G}$.

If T has an independent set of size k, then the claim is trivial. Firstly, construct the exemplar genome H as follows. For all i, if v_i is in the independent set, then delete A_i in $Y_i = v_i A_i v_i'$ (also arbitrarily delete all redundant edges in A_s in $\mathcal{H}$

for which v_s is not in the independent set of T). There are k non-breaking points between G, H—notice that any vertex v_i which is in the independent set gives us a non-breaking point $v_i v_i'$. The final exemplar genomes obtained, G and H, obviously have k exemplar non-breaking points.

If the number of the exemplar non-breaking points between $\mathcal{G}$ and $\mathcal{H}$ is k, the first thing to notice is that $Y_i = x_i x_i'$ ($N < i \leq N + M$) cannot give us any non-breaking point. So the non-breaking points must come from $Y_i = v_i A_i v_i'$ ($i \leq N$), with some A_i properly deleted (i.e., such a Y_i becomes $v_i v_i'$ in H). Moreover, there are exactly k such A_i's deleted. We show below that any two such completely deleted A_i, A_j correspond to two independent vertices v_i, v_j in T. Assume that there is an edge e_{ij} between v_i and v_j, then as both A_i, A_j are deleted, both of the two occurrences of the gene e_{ij} will be deleted from $\mathcal{H}$. A contradiction. Therefore, if the number of the exemplar non-breaking points between $\mathcal{G}$ and $\mathcal{H}$ is k, there is an independent set of size k in T.

To conclude the proof of this theorem, notice that the reduction take polynomial time (proportional to the size of T). □

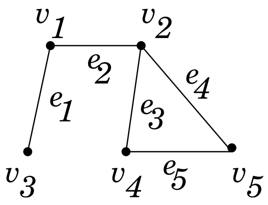


Fig. 1. Illustration of a simple graph for the reduction

In the example shown in Figure 1, we have
$\mathcal{G} : v_1 v_1' v_2 v_2' v_3 v_3' v_4 v_4' v_5 v_5' x_1 e_1 x_1' x_2 e_2 x_2' x_3 e_3 x_3' x_4 e_4 x_4' x_5 e_5 x_5'$ and
$\mathcal{H} : x_4 x_4' x_2 x_2' v_5 e_4 e_5 v_5' v_3 e_1 v_3' v_1 e_1 e_2 v_1' x_5 x_5' x_3 x_3' x_1 x_1' v_4 e_3 e_5 v_4' v_2 e_2 e_3 e_4 v_2'$.
Corresponding to the optimal independent set $\{v_3, v_4\}$, we have
$H : x_4 x_4' x_2 x_2' v_5 e_5 v_5' v_3 v_3' v_1 e_1 e_2 v_1' x_5 x_5' x_3 x_3' x_1 x_1' v_4 v_4' v_2 e_3 e_4 v_2'$. The two non-breaking points are $[v_3 v_3']$, $[v_4 v_4']$.

We comment that EBD and ENbS, even though complement to each other, are still different problems. With respect to the above theorem, when $\mathcal{G}$ is exemplar and $\mathcal{H}$ is not, there is a factor-$O(\log n)$ approximation for the EBD problem [6]. This is significantly different from ENbS, as shown in the above theorem.

4 Polynomial Time Algorithms for Some Special Cases

The proof of Theorem 1 also implies that ENbS is W[1]-complete, as Independent Set is W[1]-complete [8]. Following the recent lower bound results of Chen, *et al.*, if k is the optimal solution value for ENbS then unless an unlikely collapse occurs in the parameterized complexity theory, ENbS is not solvable in time $f(k)n^{o(k)}$, for any function f [7]. Nevertheless, we show below that for several practically interesting cases of the problem, there are polynomial time algorithms. The idea is to set a parameter in the input genomes (or sequences, as we will use interchangeably from now on) and design a polynomial time algorithm when such a parameter is $O(\log n)$.

In practical datasets, usually there are some special properties in the data. For instance, the repeated genes in the five bacteria genome pairs (Baphi-Wigg, Pmult-Hinft, Ecoli-Styphi, Xaxo-Xcamp and Ypes) are usually pegged, i.e., the repeated genes are usually separated by a peg gene which occurs exactly once [12]. When the total number of such repeated genes is a constant, our algorithm can solve this problem in polynomial time.

We first present a few extra definitions. For a genome $\mathcal{G}$ and a character g, $\text{span}(g, \mathcal{G})$ is the maximal distance between the two positions that are occupied by g in the genome $\mathcal{G}$. For example, if $\mathcal{G} = abcbaa$, $\text{span}(a, \mathcal{G}) = 5$ and $\text{span}(b, \mathcal{G}) = 2$. For a genome $\mathcal{G}$ and $c \geq 0$, we define $\text{totalocc}(c, \mathcal{G}) = \sum_{g \text{ is a character in } \mathcal{G} \text{ and } \text{span}(g,\mathcal{G}) \geq c} \text{occ}(g, \mathcal{G})$.

Assume that c and d are positive integers. A (c, d)*-even partition* for a genome $\mathcal{G}$ is $\mathcal{G} = \mathcal{G}_1\mathcal{G}_2\mathcal{G}_3$ with $|\mathcal{G}_2| = c$ and $|\mathcal{G}_1| + \lfloor |\mathcal{G}_2|/2 \rfloor = d$.

For a genome $\mathcal{G}$ and integers $c, d > 0$, a (c, d)-split G_1, G_2, G_3 for $\mathcal{G}$ is derived from a (c', d)-even partition $\mathcal{G} = \mathcal{G}_1\mathcal{G}_2\mathcal{G}_3$ for $\mathcal{G}$ for some $c \leq c' \leq 2c$ and satisfies the following conditions 1)-6):

(1) $\text{alphabet}(\mathcal{G}) = \text{alphabet}(G_1G_2G_3)$.

(2) We can further partition $\mathcal{G}_2$ into $\mathcal{G}_2 = \mathcal{G}_2^1\mathcal{G}_2^2\mathcal{G}_2^3$ such that $|\mathcal{G}_2^2| \leq c+1$, and there is at least one gene g with all its occurrences in $\mathcal{G}$ being in $\mathcal{G}_2^2$. We call such a gene g as a whole gene in $\mathcal{G}_2^2$.

(3) G_2 is obtained from $\mathcal{G}_2^2$ by deleting some genes and every gene appears at most once in G_2. And, G_2 contains one occurrence of every whole gene in $\mathcal{G}_2^2$.

(4) G_1 is obtained from $\mathcal{G}_1\mathcal{G}_2^1$ by deleting all genes in $\mathcal{G}_1\mathcal{G}_2^1$ which also appear in G_2.

(5) G_3 is obtained from $\mathcal{G}_2^3\mathcal{G}_3$ by deleting all genes in $\mathcal{G}_2^3\mathcal{G}_3$ which also appear in G_2.

(6) G_2 has no gene common with either G_1 or G_3.

Finally, for a genome $\mathcal{G}$ and integers $c, d \geq 0$, a (c, d)-decomposition is G_1x, G_2G_3, where G_1, G_2, G_3 is a (c, d)-split for $\mathcal{G}$ and x is the first character of G_2. We have the following lemma. From now on, whenever a different pair of genomes are given we assume that they are drawn from the same n gene families.

Lemma 2. *Assume that c, d are integers satisfying $c \geq 0$ and $|\mathcal{G}| - 2c \geq d \geq 2c$. and $\mathcal{G}$ is a genome with* $\text{span}(g, \mathcal{G}) \leq c$ *for every gene g in $\mathcal{G}$. Then, (1) the number of (c, d)-decompositions is at most 2^{c+1}; (2) every exemplar genome of $\mathcal{G}$ is also an exemplar genome of $G_1G_2G_3$ for some (c, d)-split G_1, G_2, G_3 of $\mathcal{G}$.*

Proof. (1). Since $\text{span}(g, \mathcal{G}) \leq c$ for every gene g in $\mathcal{G}$, it is easy to see that there is a c', $c \leq c' \leq 2c$, such that we can find (c, d)-splits G_1, G_2 and G_3 from a (c', d)-even partition $\mathcal{G} = \mathcal{G}_1\mathcal{G}_2\mathcal{G}_3$ with $\mathcal{G}_2 = \mathcal{G}_2^1\mathcal{G}_2^2\mathcal{G}_2^3$. Since $|\mathcal{G}_2^2| \leq c+1$, there are at most 2^{c+1} possible ways to obtain G_2. Therefore, the total number of decompositions is at most 2^{c+1}. (2) is easy to see. □

Lemma 3. *Let c be a positive constant and ϵ be an arbitrary small positive constant. There exists an $O(n^{c+2+\epsilon})$-time algorithm such that given an exemplar genome G, in which each genes appears exactly once, and $\mathcal{H}$, in which* $\mathrm{span}(g, \mathcal{H}) \le c$ *for every g in $\mathcal{H}$, it returns* $\mathrm{enbs}(G, \mathcal{H})$.

Proof. We use the divide-and-conquer method to compute $\mathrm{enbs}(G, \mathcal{H})$. The separator is put at the middle of $\mathcal{H}$ with width c. The genes within the region of separator are handled by a brute-force method.

```
Algorithm
$A(G, \mathcal{H})$
    Input: $G$ is a genome with no gene repetition,
        and $\mathcal{H}$ is a genome such that $\mathrm{span}(g, \mathcal{H}) \le c$ for each gene in $\mathcal{H}$.
    let $s = 0$ and $d = |\mathcal{H}|/2$.
    **for** every $(c, d)$-decomposition $H_1x, H_2H_3$ of $\mathcal{H}$)
        **begin**
            **if** the length of $H_1x$ and $H_2H_3$ is $\le \log n$
                **then** compute $A(G, H_1x)$ and $A(G, H_2H_3)$ by brute-force;
                **else** let $s' = A(G, H_1x) + A(G, H_2H_3)$;
            **if** $(s < s')$ **then** $s = s'$
        **end**
    **return** $s$;
```

The correctness of the algorithm is easy to verify. By Lemma 2 and the description of the algorithm, the computational time is based on the following recursive equation: $T(n) \le (2^{c+1}(2T(n/2 + c)) + c_0 n$, where c_0 is a constant. We show by induction that $T(n) \le c_1 n^{c+2+\epsilon}$, where c_1 is a positive constant. The basis is trivial when n is small since we can select constant c_1 large enough. Assume that $T(n) \le c_1 n^{c+2+\epsilon}$ is true all $n < m$.

$T(m) \le 2^{c+1}(2T(m/2+c)+c_0 m \le 2(2^{c+1}c_1(m/2+c)^{c+2+\epsilon})+c_0 m < c_1 m^{c+2+\epsilon}$ for all large m. □

We now have the following theorem.

Theorem 2. *Let $\mathcal{G}$ and $\mathcal{H}$ be two genomes with* $t = \mathrm{totalocc}(1, \mathcal{G}) + \mathrm{totalocc}(c, \mathcal{H})$*, for some arbitrary constant c. Then* $\mathrm{enbs}(\mathcal{G}, \mathcal{H})$ *can be computed in $O(3^{\lfloor t/3 \rfloor} n^{c+2+\epsilon})$ time.*

Proof. Algorithm:

```
    $d = 0$;
    **for** each gene $g_1$ in $\mathcal{G}$ with $\mathrm{span}(g_1, \mathcal{G}) \ge 1$
    **begin**
        **for** each position $p_1$ of $g_1$ in $\mathcal{G}$
        **begin**
            remove all $g_1$'s at all positions other than $p_1$;
        **end**
        assume that $\mathcal{G}$ has been changed to $G$;
        **for** each gene $g_2$ in $\mathcal{H}$ with $\mathrm{span}(g_2, \mathcal{H}) > c$
        **begin**
```

```
                    for each position $p_2$ of $g_2$ in $\mathcal{H}$
                    begin
                        remove all $g_2$'s at all positions other than $p_2$;
                    end
                    assume that $\mathcal{H}$ has been changed to $\mathcal{H}'$;
                    compute $d_0 = \text{enbs}(G, \mathcal{H}')$ following Lemma 3;
                    if $(d < d_0)$ then $d = d_0$;
            end
        end
        return $d$;
```

Let g_i, $1 \leq i \leq m$, be the genes in $\mathcal{G}$ and $\mathcal{H}$ with $\text{span}(g_1, \mathcal{G}) \geq 1$ in $\mathcal{G}$ or $\text{span}(g_2, \mathcal{H}) > c$ in $\mathcal{H}$. We have $t = k_1 + \cdots + k_m$. Let k_i be the number of occurrences of g_i. Notice that $k_i \geq 2$. The number of cases to select the positions of those genes in $\mathcal{G}$ and the positions of those genes in $\mathcal{H}$ is at most $k_1 \cdots k_m$, which is at most $4 \cdot 3^{\lfloor t/3 \rfloor}$ following Lemma 6. In G, every gene appears exactly once. In $\mathcal{H}'$, every gene has span bounded by c. Therefore, their distance can be computed in $O(n^{c+2+\epsilon})$ steps by Lemma 3. □

Next, we define a new parameter measure similar to the Maximum Adjacency Disruption (MAD) number in [5].

Assume that $\mathcal{G}$ and $\mathcal{H}$ are two genomes/sequences. For a gene g, define $\text{shift}(g, \mathcal{G}, \mathcal{H}) = \max_{\mathcal{G}[i]=g, \mathcal{H}[j]=g} |i - j|$, where $\mathcal{G}[i]$ is the gene/character of $\mathcal{G}$ at position i. A *space-permitted* genome $\mathcal{G}$ may have space symbols in it. For two space-permitted genomes $\mathcal{G}_1$ and $\mathcal{G}_2$, a non-breaking point $g_1 g_2$ satisfies that g_1 and g_2 appear at two positions of $\mathcal{G}$ without any other genes/characters except some spaces between them, and also at two positions of $\mathcal{H}$ without any other genes except spaces between them.

For a genome $\mathcal{G}$ and integers $c, d > 0$, an exact (c, d)-split G_1, G_2, G_3 for $\mathcal{G}$ is obtained from a (c, d)-even partition $\mathcal{G} = \mathcal{G}_1 \mathcal{G}_2 \mathcal{G}_3$ for $\mathcal{G}$ and satisfies the following conditions (1)-(5):

(1) $\text{alphabet}(\mathcal{G}) = \text{alphabet}(G_1 G_2 G_3)$.

(2) G_2 is obtained from $\mathcal{G}_2$ by replacing some characters with spaces and every non-space character appears at most once in G_2.

(3) G_1 is obtained from $\mathcal{G}_1$ by changing all $\mathcal{G}$ characters that also appear in G_2 into spaces.

(4) G_3 is obtained from $\mathcal{G}_3$ by changing all $\mathcal{G}_3$ characters that also appear in G_2 into spaces.

(5) G_2 has no common non-space character with either G_1 or G_3.

We now show the following lemmas.

Lemma 4. *Let c, k, d be positive integers. Assume that $\mathcal{G}$ is a space-permitted genome with $\text{span}(g, \mathcal{G}) \leq c$ for every character g in $\mathcal{G}$, and $\mathcal{G}$ only has spaces at the first kc positions and spaces at the last kc positions. If $|\mathcal{G}| > 2(k+4)c$ and $(k+2)c < d < |\mathcal{G}| - (k+2)c$, then $\mathcal{G}$ has at least one exact $(2c, d)$-split and for every exact $(2c, d)$-split G_1, G_2, G_3 for $\mathcal{G}$, G_2 has at least one non-space character.*

Proof. For $(k+2)c < d < |\mathcal{G}|-(k+2)c$, it is easy to see that $\mathcal{G}$ has a subsequence S of length $2c$ that starts from the d-th position in $\mathcal{G}$ and has no space character. For every subsequence S of length $2c$ of $\mathcal{G}$, if S has no space character, it has at least one character in $\mathcal{G}$ that only appears in the region of S since $\text{span}(g, \mathcal{G}) \leq c$ for every character g in $\mathcal{G}$. □

Lemma 5. *Let c be a positive constant. There exists an $O(n^{2c+1+\epsilon})$ time algorithm such that, given two space-permitted genomes/sequences $\mathcal{G}$ and $\mathcal{H}$, it returns* $\text{enbs}(\mathcal{G}, \mathcal{H})$*, if* $\text{shift}(g, \mathcal{G}, \mathcal{H}) \leq c$ *for each non-space character g, $\mathcal{G}$ and $\mathcal{H}$ only have spaces at the first and last $4c$ positions, and $|\mathcal{G}| \geq 16c$ and $|\mathcal{H}| \geq 16c$.*

Proof. Since $\text{shift}(g, \mathcal{G}, \mathcal{H}) \leq c$ for every gene/character g in $\mathcal{G}$ or $\mathcal{H}$, we have $\text{span}(g, \mathcal{G}) \leq 2c$ and $\text{span}(g, \mathcal{H}) \leq 2c$ for every character g in $\mathcal{G}$ or $\mathcal{H}$.

```
Algorithm
    $B(\mathcal{G}, \mathcal{H})$
    Input: $\mathcal{G}, \mathcal{H}$ are two space-permitted genomes.
    assume that $|\mathcal{G}| \leq |\mathcal{H}|$;
    set $s = 0$ and $d = \lfloor |\mathcal{G}|/2 \rfloor$;
    for every exact $(2c, d)$-split $G_1, G_2, G_3$ of $\mathcal{G}$
    begin
        for every exact $(2c, d)$-split $H_1, H_2, H_3$ of $\mathcal{H}$
        begin
            if the length of $\mathcal{G}$ and $\mathcal{H}$ is $\leq \log n$
                then compute $\text{enbs}(\mathcal{G}, \mathcal{H})$ by brute-force;
                else $s = B(G_1G_2, H_1H_2) + B(G_2G_3, H_2H_3) - B(G_2, H_2)$;
            if $(s < s')$ then $s = s'$;
        end
    end
    return $s$;
```

Following the divide-and-conquer method, it is easy to see that G_1G_2, H_1H_2, G_2G_3 and H_2H_3 have spaces in the first and last $2c$ positions. This is because $\text{span}(g, \mathcal{G}) \leq 2c, \text{span}(g, \mathcal{H}) \leq 2c$ for every character g. $B(G_2, H_2)$ can be determined by a linear scan, since both of them are exemplar. The computational time is determined by the recurrence relation: $T(n) = (2^{2c} + 2c)(2T(\frac{n}{2} + 2c) + O(n))$, which has solution $T(n) = O(n^{2c+1+\epsilon})$ as we show in the Lemma 3. □

Lemma 6. *Let $k \geq 3$ be a fixed integer. Assume that $k_1, k_2, \cdots, k_m$ are m integers that satisfies $k_i \geq 2$ for $i = 1, 2, \cdots, m$ and $k_1 + k_2 + \cdots + k_m = k$. Then $k_1 k_2 \cdots k_m \leq 4 \cdot 3^{\lfloor \frac{k}{3} \rfloor}$.*

Proof. We assume that for fixed k, m is the largest integer that makes the product $k_1 k_2 \cdots k_m$ maximal and $k_1 + k_2 + \cdots + k_m = k$. We claim that $k_i \leq 3$ for all $i = 1, 2, \cdots, m$. Otherwise, without loss of generality, we assume that $k_m > 3$. Clearly, $2 \cdot (k_m - 2) \geq k_m$. Replace k_m by $k'_m = 2$ and $k_{m+1'} = k_m - 2$. We still have that $k_1 + k_2 + \cdots + k_{m-1} + k'_m + k'_{m+1} = k$ and $k_1 k_2 \cdot k_{m-1} k'_m k'_{m+1} \geq k_1 k_2 \cdots k_m$. This contradicts that m is maximal. Therefore, each $k_i (i = 1, 2, \cdots, m)$ is either 2 or 3 while $k_1 + k_2 + \cdots + k_{m-1} + k_m = k$

and $k_1k_2\cdots k_m$ is still maximal. It is impossible that there are at least three 2s among $k_1, k_2, \cdots, k_m$. This is because that $2+2+2=3+3$ and $2\cdot 2\cdot 2 < 3\cdot 3$. On the other hand, the number of 3s among $k_1, k_2, \cdots, k_m$ is at most $\lfloor \frac{k}{3} \rfloor$ since $k_1+k_2+\cdots+k_{m-1}+k_m = k$. □

Finally, we have the following theorem.

Theorem 3. *Let $\mathcal{G}$ and $\mathcal{H}$ be two genomes with a total of t genes g satisfying* $\text{shift}(g,\mathcal{G},\mathcal{H}) > c$*, for some arbitrary positive constant c. Then* $\text{enbs}(\mathcal{G},\mathcal{H})$ *can be computed in $O(3^{\lfloor t/3 \rfloor} n^{2c+1+\epsilon})$ time.*

The idea to prove this theorem is as follows. We consider all possible ways to replace every gene g, $\text{shift}(g,\mathcal{G},\mathcal{H}) > c$, with space in $\mathcal{G}$ and $\mathcal{H}$, while keeping one occurrence of g in $\mathcal{G}$ and $\mathcal{H}$. For each pair of such resulting $\mathcal{G}'$ and $\mathcal{H}'$, we consider to use the algorithm in Lemma 5 to compute $\text{enbs}(\mathcal{G}',\mathcal{H}')$. Notice that we may have spaces not only in the two ends but also in the middle of $\mathcal{G}'$ or $\mathcal{H}'$. However, we can modify the method of selecting exact (c,d)-splits for the two genome. The new method is to start at the middle position of $\mathcal{G}'$ (or $\mathcal{H}'$) to find the nearest non-space gene either in the right part or the left of the middle position. Say, such a gene is u in the right part of the middle position of $\mathcal{H}'$. Then, we determine $\mathcal{H}_2$ by including c positions to the right of u and also including c or more positions to the left to make sure that the middle position is also included. The rest part in the left of $\mathcal{H}_2$ is $\mathcal{H}_1$, and the rest in the right of $\mathcal{H}_2$ is $\mathcal{H}_3$. It is easy to see that the number of genes (not spaces) in $\mathcal{H}_2$ is no more than $2c$. Similarly, we can determine an even partition for $\mathcal{G}_1$. Notice also that spaces do not contribute to constructing exact (c,d)-splits. Therefore, $\text{enbs}(\mathcal{G}',\mathcal{H}')$ can be computed, following the spirit of the algorithm in Lemma 5.

5 Concluding Remarks

We define a new measure—non-breaking similarity of genomes and prove that the exemplar version of the problem does not admit an approximation of factor $n^{1-\epsilon}$ even when one of the input genomes is given exemplar; and moreover, the problem is W[1]-complete. This differs from the corresponding result for the dual exemplar breakpoint distance problem, for which a factor-$O(\log n)$ approximation exists when one of the input genomes is exemplar (and for the general input there is no polynomial time approximation) [6]. On the other hand, we present polynomial time algorithms for several practically interesting cases under this new similarity measure. In practice, the practical datasets usually have some special properties [12], so our negative results might not hold and our positive results might be practically useful. We are currently working along this line.

References

1. Bryant, D.: The complexity of calculating exemplar distances. In: Sankoff, D., Nadeau, J. (eds.) Comparative Genomics: Empirical and Analytical Approaches to Gene Order Dynamics, Map Alignment, and the Evolution of Gene Families, pp. 207–212. Kluwer Acad. Pub. Boston, MA (2000)

2. Blin, G., Rizzi, R.: Conserved interval distance computation between non-trivial genomes. In: Wang, L. (ed.) COCOON 2005. LNCS, vol. 3595, pp. 22–31. Springer, Heidelberg (2005)
3. Chen, Z., Fu, B., Fowler, R., Zhu, B.: Lower bounds on the application of the exemplar conserved interval distance problem of genomes. In: Chen, D.Z., Lee, D.T. (eds.) COCOON 2006. LNCS, vol. 4112, pp. 245–254. Springer, Heidelberg (2006)
4. Chen, Z., Fu, B., Fowler, R., Zhu, B.: On the inapproximability of the exemplar conserved interval distance problem of genomes. J. Combinatorial Optimization (to appear)
5. Chauve, C., Fertin, G., Rizzi, R., Vialette, S.: Genomes containing duplicates are hard to compare. In: Proc. 2nd Intl. Workshop on Bioinformatics Research and Applications (IWBRA'06), LNCS 3992, pp. 783–790 (2006)
6. Chen, Z., Fu, B., Zhu, B.: The approximability of the exemplar breakpoint distance problem. In: Cheng, S.-W., Poon, C.K. (eds.) AAIM 2006. LNCS, vol. 4041, pp. 291–302. Springer, Heidelberg (2006)
7. Chen, J., Huang, X., Kanj, I., Xia, G.: Linear FPT reductions and computational lower bounds. In: Proc. 36th ACM Symp. on Theory Comput. (STOC'04), pp. 212–221 (2004)
8. Downey, R., Fellows, M.: Parameterized Complexity. Springer, Heidelberg (1999)
9. Hästad, J.: Clique is hard to approximate within $n^{1-\epsilon}$. Acta. Mathematica, 182, 105–142 (1999)
10. Hannenhalli, S., Pevzner, P.: Transforming cabbage into turnip: polynomial algorithm for sorting signed permutations by reversals. J. ACM, 46(1), 1–27 (1999)
11. Gascuel, O. (ed.): Mathematics of Evolution and Phylogeny. Oxford University Press, Oxford, UK (2004)
12. Nguyen, C.T., Tay, Y.C., Zhang, L.: Divide-and-conquer approach for the exemplar breakpoint distance. Bioinformatics 21(10), 2171–2176 (2005)
13. Sankoff, D.: Genome rearrangement with gene families. Bioinformatics 16(11), 909–917 (1999)
14. Sturtevant, A., Dobzhansky, T.: Inversions in the third chromosome of wild races of drosophila pseudoobscura, and their use in the study of the history of the species. In: Proc. Nat. Acad. Sci. USA, vol. 22 pp. 448–450 (1936)
15. Watterson, G., Ewens, W., Hall, T., Morgan, A.: The chromosome inversion problem. J. Theoretical Biology 99, 1–7 (1982)

A New and Faster Method of Sorting by Transpositions[⋆]

Maxime Benoît-Gagné and Sylvie Hamel

Département d'Informatique et de Recherche Opérationnelle,
Université de Montréal, CP.6128 Succ. Centre-Ville
Montréal, Québec Canada, H3C 3J7
phone:+1 514 343-6111(3504) fax:+1 514 343-5834
sylvie.hamel@umontreal.ca

Abstract. Some of the classical comparisons of DNA molecules consists in computing rearrangement distances between them, i.e.: a minimal number of rearrangements needed to change a molecule into another. One such rearrangement is that of *transposition*. At this time, it is not known if a polynomial time algorithm exists to compute the exact transposition distance between two permutations. In this article, we present a new and faster method of sorting by transpositions. While there does exist better algorithms with regards to distance approximation, our approach relies on a simpler structure which makes for a significantly faster computation time, while keeping an acceptable close approximation.

Keywords: Genome rearrangement, transposition distance, permutation.

1 Introduction

Among the methods used to estimate the evolutionary distance between two different organisms, some consist in finding out which evolutionary events are more probable to have modified one genome into another. One may thus search for "local" mutations – insertions, deletions or substitutions of nucleotides– or for genome rearrangements –mutations that change the gene order of a genome. These genome rearrangements are apparently rarer than local mutations so this should make them a more robust evolutionary indicator.

One natural rearrangement that is considered is a *transposition*. In this case, two DNA sequences containing the same n genes, in some different orders, are each represented by a permutation. A transposition consists in removing a *block of consecutive elements* from the sequence and placing it elsewhere. Given two species with different gene orders, we want to compute a *parsimonious transposition distance* between them. This is the minimal number of transpositions needed to change one gene order into the other one. Even if it is true that many species have different sets of genes and possibly multiple copies of these genes,

[⋆] With the support of NSERC.

we can compute a transposition distance between any two species by restraining our set of genes to those present in both species, choosing, arbitrarily, one copy for each of them. We can also choose to only work with species that have exactly the same set of genes (for examples of such species see [6,8]).

At this time, no known polynomial time algorithm exists for computing the exact transposition distance between two permutations. Several algorithms already exist to compute approximate transposition distances, with different approximation factor and time complexity. Most of these algorithms rely on the construction of a so-called *cycle graph* [1,2,3,5,10]. The proofs of soundness for the approximation factor of these algorithms rely on an analysis of these cycle graphs regarding particular properties. The resulting algorithms can be quite time-consuming even for nice theoretical time complexities. The first such algorithm, by Bafna and Pevzner [1], has an approximation factor of 1.5 and a theoretical time complexity of $\mathcal{O}(n^2)$. This algorithm relies on a complex data structure and its best known implementation, due to Walter and al. [10], has time complexity of $\mathcal{O}(n^3)$. In 2003, Hartman [5] simplified Bafna and Pevzner's algorithm without changing the time complexity or approximation factor; and in 2005, with Elias [3], he presented an 1.375 approximation algorithm, which again has quadratic complexity. To this day, this is the best approximation algorithm, with the drawback that it requires the analysis of about 80 000 different cases of cycle graphs in order to prove this 1.375 factor.

To try to simplify the implementation complexity of the problem, Walter, Dias and Meidanis [9] suggested a 2.25 approximation algorithm using a more simpler data structure known as the *breakpoint diagram*. The time complexity of their algorithm is also quadratic. Even if breakpoint diagrams are much simpler than cycle graphs, it is still a graph structure that has to be built out of the permutation to get the approximate distance. The goal of this article is to describe an $\mathcal{O}(n^2)$ algorithm that neither relies on a graph construction nor on a cycle analysis, thus resulting in a better real time complexity.

The article is organized as follows. In section 2, we recall formally the transposition distance problem and introduce the notion of *coding* for a permutation. In section 3, we present a very simple approximation algorithm for our problem and discuss its approximation factor. Section 4 presents a more complex version of time complexity $\mathcal{O}(n^4)$ but with a better approximation factor. Finally, in section 5, we establish results on both algorithms for small and large permutations and conclude.

2 Definitions

Let us briefly recall the standard definitions needed for our problem and define the manner in which permutations will be encoded.

2.1 The Transposition Distance Problem

A **permutation** π is a bijection of $[n] = \{1, 2 \ldots, n\}$ onto itself. As usual we will denote a permutation π of $[n]$ as $\pi = \pi_1\pi_2 \ldots \pi_n$.

Definition 1. *A* **transposition** *is an operation on permutations that moves a block of contiguous elements, placing it elsewhere. More formally, for* $1 \leq i \leq j \leq n+1$ *and* $j < k \leq n+1$ *(or* $1 \leq k < i$*), we define the* **transposition** $\rho(i,j,k)$ *on* π *as*

$$\rho(i,j,k) \cdot \pi = \pi_1 \ldots \pi_{i-1}\pi_j \ldots \pi_{k-1}\pi_i \ldots \pi_{j-1}\pi_k \ldots \pi_n$$

i.e.: the block going from π_i *to* π_{j-1} *(inclusively) has been removed and placed right before* π_k*.*

The **transposition distance problem** consists in finding the minimal number of transpositions needed to sequentially transform any permutation $\pi = \pi_1\pi_2 \ldots \pi_n$ into the identity permutation $\mathrm{Id} = 12 \ldots n$. This transposition distance is denoted $d(\pi)$. To simplify the presentation, sometimes a permutation $\pi = \pi_1\pi_2 \ldots \pi_n$ will be extended, as in [9], by two elements $\pi_0 = 0$ and $\pi_{n+1} = n+1$. This extended permutation is still denoted π.

Definition 2. *Given a permutation* π *of* $[n]$*, we say that we have a* **breakpoint** *at* π_i *if* $\pi_i + 1 \neq \pi_{i+1}$*, for* $0 \leq i \leq n$*. The number of breakpoints of* π *is denoted* $b(\pi)$*.*

One observes readily that at most three breakpoints can be removed by a single transposition. It is also easy to see that a permutation can be sorted by transposing one element π_i at a time to its "right position", thus removing at least one breakpoint each time. These two observations lead to the following classical lemma.

Lemma 1. *Given a permutation* π *of* $[n]$*, we have* $\frac{b(\pi)}{3} \leq d(\pi) \leq b(\pi)$ ■

2.2 Coding a Permutation

For a given permutation π, we compute two *codes*. From these codes we can derive an approximate transposition distance for the permutation. Intuitively, for a position i in a permutation π, the left (resp. right) code at this position is simply the number of elements bigger (resp. smaller) than π_i to its left (resp. its right). More formally we have

Definition 3. *Given a permutation* $\pi = \pi_1 \ldots \pi_n$*, the* **left code** *of the element* π_i *of* π*, denoted* $lc(\pi_i)$*, is*

$$lc(\pi_i) = |\{\pi_j \mid \pi_j > \pi_i \text{ and } 0 \leq j \leq i-1\}|, \text{ for } 1 \leq i \leq n,$$

Similarly, the **right code** *of the element* π_i *of* π*, denoted* $rc(\pi_i)$*, is*

$$rc(\pi_i) = |\{\pi_j \mid \pi_j < \pi_i \text{ and } i+1 \leq j \leq n+1\}|, \text{ for } 1 \leq i \leq n.$$

The left (resp. right) code of a permutation π *is then defined as the sequence of lc's (resp. rc's) of its elements.*

$$\begin{array}{rlrlrl}
\pi & = 6\ 3\ 2\ 1\ 4\ 5 & \sigma & = 3\ 5\ 2\ 1\ 6\ 4 & \gamma & = 6\ 5\ 4\ 3\ 2\ 1 \\
lc(\pi) & = 0\ 1\ 2\ 3\ 1\ 1 & lc(\sigma) & = 0\ 0\ 2\ 3\ 0\ 2 & lc(\gamma) & = 0\ 1\ 2\ 3\ 4\ 5 \\
rc(\pi) & = 5\ 2\ 1\ 0\ 0\ 0 & rc(\sigma) & = 2\ 3\ 1\ 0\ 1\ 0 & rc(\gamma) & = 5\ 4\ 3\ 2\ 1\ 0
\end{array}$$

Fig. 1. The left and right codes of some permutations

Figure 1 gives both left and right codes for three different permutations of length 6.

Observe that the identity permutation Id $= 12\ldots n$, is the only permutation for which we have $lc(\text{Id}) = 00\ldots0 = rc(\text{Id})$. In the next section we describe a method for sorting permutations using only transpositions that raise the number of zeros in either the left or right code of π.

3 Sorting a Permutation - The Easy Way

In this section, we show how to sort permutations with transpositions using only their codes. The emphasis is on the time complexity of the algorithm, but we also prove that the number of transpositions needed to sort π approximates the real distance by a factor lying between 1 and 3, depending on π.

3.1 Definitions, Algorithm and Approximation Factor

Definition 4. *Let us call* **plateau** *any maximal length sequence of contiguous elements in a number sequence that have the same nonzero value. The number of plateaux in a code c is denoted $p(c)$.*

Definition 5. *We denote by $p(\pi)$ the minimum of $p(lc(\pi))$ and $p(rc(\pi))$.*

Example 1. For the permutation π from Figure 1, we have $p(lc(\pi)) = 4$ since we have the four plateaux: 1, 2, 3 and 1 1 and $p(rc(\pi)) = 3$ since we have the three plateaux: 5, 2 and 1. Thus, $p(\pi) = 3$.

Lemma 2. *Given a permutation $\pi = \pi_1 \ldots \pi_n$, the leftmost (resp. rightmost) plateau of $lc(\pi)$ (resp. $rc(\pi)$) can be removed by a transposition to the left (resp. the right) without creating any new plateaux in the code.*

Proof. This follows directly from the definition of $lc(\pi)$ (resp. $rc(\pi)$). Suppose that the leftmost (resp. rightmost) plateau in $lc(\pi)$ (resp. $rc(\pi)$) is from position i to $j-1$. Then all entries of the left (resp. right) code are equal to 0 before position i (resp. after position $j-1$). In other terms, the entries $\pi_1, \ldots, \pi_{i-1}$ (resp. $\pi_j, \ldots, \pi_n$) are in increasing order. So, if $lc(\pi_i) = v$ (resp. $rc(\pi_i) = v$), the transposition $\rho(i, j, k)$, where $k = i - v$ (resp. $k = j + v$), removes the first plateau without creating a new one. ■

Lemma 2 gives us the following result.

Lemma 3. *We have $d(\pi) \leq p(\pi)$.*

Proof. The identity permutation Id is the only permutation for which we have $lc(\mathrm{Id}) = 00\ldots0 = rc(\mathrm{Id})$. Thus, Lemma 3 comes directly from the facts that we have $p(\pi) = 0 \iff \pi = Id$ and that we can transpose all plateaux appearing in either $lc(\pi)$ or $rc(\pi)$, one by one without creating new ones. ■

From Lemma 2 and Lemma 3 we can immediately derive the following $\mathcal{O}(n^2)$ algorithm for the computation of an approximate transposition distance.

```
Algorithm EasySorting (input: $\pi$ a permutation of $[n]$)

    $lc(\pi)$ = left code of $\pi$ (easily computed in $\mathcal{O}(n^2)$)
    $rc(\pi)$ = right code of $\pi$ (easily computed in $\mathcal{O}(n^2)$)
    $lp(\pi) = p(lc(\pi))$(easily computed in $\mathcal{O}(n)$)
    $rp(\pi) = p(rc(\pi))$(easily computed in $\mathcal{O}(n)$)
    RETURN $p(\pi) = \min\{lp(\pi), rp(\pi)\}$
```

To list the $p(\pi)$ transpositions needed to sort π, while computing $p(lc(\pi))$ and $p(rc(\pi))$, we need only to record both ends (start and finish) of the plateaux considered as well as their code values. (see Lemma 2).

Given a permutation π, Lemma 1 and Lemma 3 gives us directly the following approximation factor for $d(\pi)$.

Lemma 4. *An approximation factor of $d(\pi)$, for permutation π, with Algorithm EasySorting is*

$$\frac{c \cdot p(\pi)}{b(\pi)}, \quad where \quad c = \frac{3\left\lfloor \frac{b(\pi)}{3} \right\rfloor + b(\pi)\ mod\ 3}{\left\lceil \frac{b(\pi)}{3} \right\rceil}$$

Proof. From Lemma 1, the best possible outcome, given a permutation π, is that each transposition reduces the number of breakpoints by 3. If the number of breakpoints is a multiple of 3 then $c = 3$. Otherwise, we have to perform at least one transposition which will reduce the number of breakpoints by a factor less than 3. The constant c gives, in that case, the best average scenario. Since $b(\pi)/p(\pi)$ is the number of breakpoints removed, on average, with our algorithm, the result follows. ■

Example 2. Going back on Example 1, we get, for permutation π of Figure 1, $p(\pi) = p(rc(\pi)) = 3$. The three plateaux are 1 at position 3, 2 at position 2 and finally 5 at position 1. The next transpositions (see proof of Lemma 2) sort the permutation:

$$\begin{array}{rlclclcl} rc(\pi) = & 5\,2\,1\,0\,0\,0 & & 5\,2\,0\,0\,0\,0 & & 5\,0\,0\,0\,0\,0 & & 0\,0\,0\,0\,0\,0 \\ \pi \;\; = & 6\,3\,2\,1\,4\,5 & \xrightarrow{\rho(3,4,5)} & 6\,3\,1\,2\,4\,5 & \xrightarrow{\rho(2,3,5)} & 6\,1\,2\,3\,4\,5 & \xrightarrow{\rho(1,2,7)} & 1\,2\,3\,4\,5\,6 \end{array}$$

Since the extended permutation $\pi = \mathbf{0}\,6\,3\,2\,1\,4\,5\,\mathbf{7}$ has 6 breakpoints, our approximative theoretical factor is $\frac{3p(\pi)}{b(\pi)} = \frac{3*3}{6} = 1.5$, while the real factor is 1, since in fact $d(\pi) = 3$.

The upper bound of the theoretical approximation factor of Lemma 4 is not tight for certain permutations π of $[n]$, that we can not classify at this time. For example, it has been proved in [7] and then, later on, independently in [2] and [4] that if π is the inverse permutation $\pi = n\ n-1\ \dots\ 2\ 1$, then $d(\pi) = \lfloor n/2 \rfloor + 1$. Since, in that case $p(\pi) = n - 1$ and $b(\pi) = n + 1$, we have the theoretical approximation factor, given by Lemma 4, $\lim_{n\to\infty} \frac{c \cdot p(\pi)}{b(\pi)} = 3$, which is bigger than the real one, $p(\pi)/d(\pi)$=2.

We were able to compute the exact distance $d(\pi)$ for all permutations of n, $1 \leq n \leq 9$. We performed our algorithm on these small permutations and compared the real approximation factor of our algorithm, i.e.: $p(\pi)/d(\pi)$, and the theoretical one given by Lemma 4. The results are presented in figure 2 and 3.

	1				1,2	1,25	1,33		1,4	1,5		1,67	1,75	2	Real
n	1	1,33	1,5	1,67	2	1,67	1,33	2	2,33	1,5	2	1,67	2,33	2	Theo.
4	23	0	1	0	0	0	0	0	0	0	0	0	0	0	
5	84	0	30	0	0	0	0	1	0	5	0	0	0	0	
6	380	43	189	0	0	2	63	7	0	35	0	1	0	0	
7	1793	1096	728	0	0	408	772	28	0	140	9	64	0	2	
8	7283	11323	2142	1995	817	9153	4812	84	24	420	1501	693	23	50	

Fig. 2. Comparison between real and theoretical approximation factor for small permutations

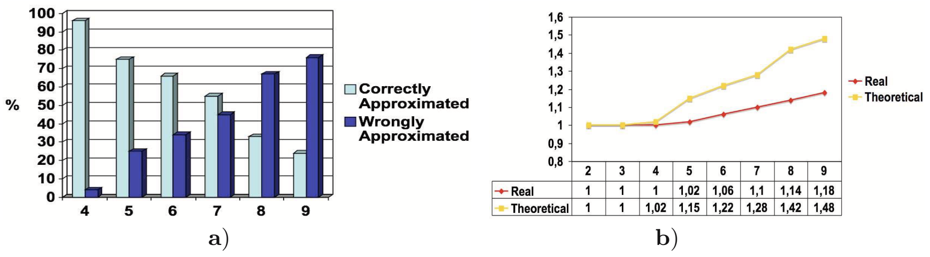


Fig. 3. Some results on the real versus theoretical approximation factor

Figure 2 shows for each n, $1 \leq n \leq 8$, the number of permutations that have a certain real versus theoretical approximation factor. It can be seen that of the 20 748 permutations of $n = 8$ that were sorted correctly (real approximation factor of 1) by our algorithm not even half of them (7283 to be exact) have a theoretical approximation factor of 1. The rest have a theoretical approximation factor of 1,33, 1,5 and even 1,67. So, the theoretical approximation factor we will get for *large permutations* is as much as 1,67 bigger than the real one. Keep that in mind when looking at the results for *large permutations* (Figure 6). In order to give a better view of what has been just discussed, Figure 3a shows, for each n, the percentage of permutations for which the real approximation factor,

$p(\pi)/d(\pi)$, is correctly approximate by Lemma 4. We see that as n grows larger, this percentage decreases rapidly.

Figure 3b shows graphically the relation between the mean of the real approximation factors and the mean of the theoretical ones obtained from the formula in Lemma 4. These data suggest that the curve of the real mean grows more slowly than the theoretical one. This may indicate that, although the theoretical approximation factor tends to 3, the real one tends to a number significantly smaller than 3.

4 Sorting a Permutation - The Other Way

The algorithm presented in the previous section naively computes an approximate transposition distance given left or right codes of permutations. Here, we summarily present ways to improve the performance of our algorithm. For lack of space, these ideas have to be developed in a future paper.

One clear way to improve the performance of our algorithm is to find an efficient way to move more than one plateau with each transposition. The following definitions introduce some properties of codes that can be used to do this. These definitions are illustrated in Figure 4. The ideas presented here have already been implemented but only for left codes. However the definitions apply for both codes and implementations for both ought to give even better performances.

Definition 6. *Given a permutation π of $[n]$, let us denote by P_1, P_2, $\cdots$ P_m, the m plateaux of $lc(\pi)$, from left to right. Let us further denote by $v(P_i)$ the value P_i i.e.: the left code of the elements of the permutation in this plateau. We have the following definitions:*

a) Two consecutive plateaux P_i and P_{i+1} are said to be a **fall** *if $v(P_i) > v(P_{i+1})$ and a* **rise** *if $v(P_i) < v(P_{i+1})$. A fall (or rise) is said to be* **good** *if there exists a position k in the permutation such that the transposition of the segment P_iP_{i+1} just before k removes at least two plateaux.*
b) Three consecutive plateaux P_i, P_{i+1} and P_{i+2} are called a **ditch** *if $v(P_{i+1}) < v(P_i), v(P_{i+2})$ and $v(P_i) = v(P_{i+2})$. They are called an* **asymmetric ditch** *if $v(P_{i+1}) < v(P_i), v(P_{i+2})$ and $v(P_i) + l = v(P_{i+2})$, where l is the length of P_{i+1}.*
c) The sequence of plateaux $P_iP_{i+1} \dots P_{i+k}$ is a **mountain** *if $v(P_i) = v(P_{i+k})$ and $v(P_{i+1}), \cdots, v(P_{i+k-1}) \geq v(P_i)$.*
d) The sequence of plateaux $P_iP_{i+1} \dots P_{i+k}$ is an **ascent** *if $k \geq 4$ and $v(P_{i+1}) = v(P_i) + l_i, \cdots, v(P_{i+k}) = v(P_{i+k-1}) + l_{i+k-1}$, where l_j is the length of P_j.*

Without going into details, let us just mention that we can use these notions to sort a permutation in more effective ways than before. Indeed, the transposition of a good fall (or rise) removes at least two plateaux (see Figure 4*a* where the good fall at position 5 and 6 of the permutation are moved before position 1). For an asymmetric ditch, it is easy to see that we can first transpose the bottom of the asymmetric ditch and then the rest, thus removing 3 plateaux with 2

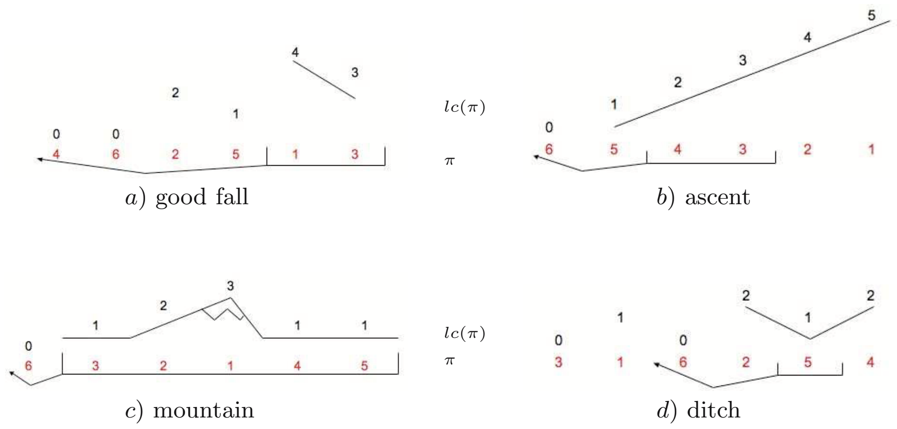


Fig. 4. Illustrations of the terms in Definition 6 and first efficient transposition

transpositions. In the case of ditches, the transposition of the bottom often creates a good fall. In this case, we can also remove 3 plateaux in 2 transpositions (see Figure 4d). Mountains can be sorted by "level". In Figure 4c, we pictured a mountain having 3 levels and 4 plateaux. By transposition of the whole mountain one position to the left, the values in the plateaux drop by 1, thus removing two plateaux. The other levels of the mountain can then be transposed one by one. Finally, as illustrated in Figure 4b, an example of an ascent is the code of the inverse permutation. This fact allows an easy generalization of the algorithm for the sorting of inverse permutations ([2,4,7]) in the ascent case. This means that an ascent containing m plateaux can be sorted with $\lfloor m/2 \rfloor + 1$ transpositions.

All these different entities can overlap in a code so that we need to transpose them in a specific order to get the best performance. We have tried different orders and the best way seems to first sort the mountains, then ascents, ditches, falls, rises and finally the remaining plateaux. This gives an $\mathcal{O}(n^5)$ algorithm. Using a heuristic testing only one transposition position while testing for a good fall or a good rise we can decrease the complexity of the algorithm to $\mathcal{O}(n^4)$. The results on small and large permutations are presented in the next section, where the algorithm is called MADFRP (for Mountain, Ascent, Ditch, Fall, Rise and Plateau). Note that we use slightly modified definitions of the entities (enabling falls and rises with more than two plateaux) in our implementation in order to use to the maximum the potential of these ideas.

5 Some Other Results and Comparisons

In [1], Bafna and Pevzner presented three algorithms to compute an approximate transposition distance of a permutation based on graph cycles properties. One is called TSort and has an approximation factor of 1,75, another is called TransSort and has an approximation factor of 1,5. Finally they mentioned a simpler version that has an approximation factor of 2. Here, we call this last algorithm BP2 and use it for comparison with our algorithm. We have implemented EasySorting, Mountain (an algorithm that uses the left code of a permutation and first sorts the mountains from left to right and then the remaining plateaux), MADFRP (briefly presented in section 4) and BP2 in JAVA and performed our computations on an AMD Athlon 64 bits, 2200 MHZ with 3.9 Gb RAM computer. The results for small and large permutations are presented, respectively, in Figure 5 and 6.

		EasySort		Mountain		MADFRP		BP2	
n	Total	$\neq d(\pi)$	Approx.	$\neq d(\pi)$	Approx.	$\neq d(\pi)$	Approx.	$\neq d(\pi)$	Approx.
2	2	0	1	0	1	0	1	0	1
3	6	0	1	0	1	0	1	0	1
4	24	0	1	0	1	0	1	1	1,5
5	120	6	1,5	6	1,5	1	1,5	7	1,5
6	720	108	1,667	103	1,5	29	1,5	86	1,5
7	5040	1423	2	1314	1,67	484	1,67	792	1,67
8	40320	17577	2	15941	2	7416	2	9162	1,75
9	362880	211863	2	190528	2	102217	2	100332	1,75

Fig. 5. Comparison of the number of wrong calculations and worst real approximation factors

		EasySort			Mountain		BP2	
n	sample size	distance	factor	time	distance	time	distance	time
10	1000	5,59	1,51	-	5,46	-	5,06	-
16	1000	10,63	1,86	-	10,11	-	8,50	-
32	1000	25,02	2,29	-	23,60	-	17,76	-
64	1000	55,45	2,55	-	52,50	-	36,09	0,00167
128	1000	117,96	2,75	0,00106	113,29	0,00181	72,57	0,00484
256	1000	244,38	2,85	0,00255	237,23	0,00529	145,18	0,02288
512	1000	498,91	2,92	0,00761	488,57	0,0234	290,56	0,1160
1000	100	985,63	2,95	0,02874	971,72	0,1152	567,53	1,299
5000	100	4982,05	2,99	0,5824	4958,75	4,448	2840,89	144,8

Fig. 6. Comparison of the mean of the transposition distances and computational time for large permutations

Figure 5 shows, for each algorithm, first the number of permutations for which the algorithm did not compute the right distance $d(\pi)$ and then, given n, the worst approximation factor for any permutation of $[n]$. The computational time is not shown here since all the algorithms were able to give their results in less than 1×10^{-3} seconds/permutation. It is interesting to see that, even with an algorithm as simple as EasySorting, a lot of permutations were sorted correctly and the worst approximation factor we got was 2. By looking at entities, like mountains, that removes more than one plateau at once, we see that the results are slightly better and we get even better than BP2 with MADFRP for n up to 8.

To compare the algorithms for large permutations, we randomly pick a sample of 1000 permutations for each n, except for $n = 1000$ and 5000, for which we pick a smaller sample of 100 permutations due to computational time. Since for these permutations π, we do not know $d(\pi)$, we computed for each sample the mean of the approximate distances given by each algorithm. The results are shown in Figure 6, where a "$-$" in the time column indicates that the calculation time is less than 1×10^{-3}. Looking at the results, we see that EasySorting and Mountain are way faster than BP2 and that they get distances close to the ones obtained by BP2, up to $n = 64$. For bigger n, BP2 is better, getting distances up to 1.7 smaller than our algorithms.

6 Conclusions and Future Work

In this article we have presented a method for the computation of an approximate transposition distance for a permutation which does not rely on graph cycles as other existing algorithms. We have divised an encoding for permutations giving a fast algorithm exhibiting good results on small permutations. On large permutations, our algorithm is way faster than any existing one with the potential small drawback that we can only prove an approximation factor bound of 3. However, this bound appeared to be far from tight as far as our experimental data suggests. We plan to investigate in more detail the aspects briefly outlined in section 4 to get more precise theoretical bounds. The practical running time of the algorithm MADFRP should also be improved by avoiding the recomputations of the codes after each transposition.

Acknowledgements

We first thank the NSERC summer intern Caroline Quinn who participated to this project in its early stages. We also thank the anonymous reviewers for their helpful and constructive comments.

References

1. Bafna, V., Pevzner, P.A.: Sorting by transpositions. SIAM J. Discrete Math. 11(2), 224–240 (1998)
2. Christie, D.A.: Genome rearrangements problems, PhD thesis, Glasgow University, Scotland (1998)
3. Elias, I., Hartman, T.: A 1.375-Approximation Algorithm for Sorting by Transpositions. In: Casadio, R., Myers, G. (eds.) WABI 2005. LNCS (LNBI), vol. 3692, pp. 204–215. Springer, Heidelberg (2005)
4. Eriksson, H., et al.: Sorting a bridge hand. Discrete Mathematics 241, 289–300 (2001)

5. Hartman, T.: A Simpler 1.5-Approximation Algorithm for Sorting by Transpositions. In: Baeza-Yates, R.A., Chávez, E., Crochemore, M. (eds.) CPM 2003. LNCS, vol. 2676, pp. 156–169. Springer, Heidelberg (2003)
6. Hoot, S.B., Palmer, J.D.: Structural rearrangements, including parallel inversions, within the chloroplast genome of Anemone and related genera. Journal of Molecular Evolution 38, 274–281 (1994)
7. Meidanis, J., Walter, M.E.M.T., Dias, Z.: Transposition distance between a permutation and its reverse. In: Proceedings of the 4th South American Workshop on String Processing (WSP'97), pp. 70–79 (1997)
8. Palmer, J.D., Herbon, L.A.: Tricircular mitochondrial genomes of Brassica and Raphanus: reversal of repeat configurations by inversion. Nucleic Acid Research 14, 9755–9764 (1986)
9. Walter, M.E.M.T., Dias, Z., Meidanis, J.: A new Approach for Approximating the Transposition Distance. In: Proceedings of SPIRE, pp. 199–208 (2000)
10. Walter, M.E.M.T., et al.: Improving the algorithm of Bafna and Pevzner for the problem of sorting by transpositions: a practical approach. Journal of Discrete Algorithms 3, 342–361 (2005)

Finding Compact Structural Motifs

Jianbo Qian[1], Shuai Cheng Li[1], Dongbo Bu[1], Ming Li[1], and Jinbo Xu[2]

[1] David R. Cheriton School of Computer Science
University of Waterloo, Waterloo
Ontario, Canada N2L 3G1
{j3qian,scli,dbu,mli}@cs.uwaterloo.ca
[2] Toyota Technological Institute at Chicago
1427 East 60th Street, Chicago, IL 60637
j3xu@tti-c.org

Abstract. Protein structure motif detection is one of the fundamental problems in Structural Bioinformatics. Compared with sequence motifs, structural motifs are more sensitive in detecting the evolutionary relationships among proteins. A variety of algorithms have been proposed to attack this problem. However, they are either heuristic without theoretical performance guarantee, or inefficient for employing an exhaustive search strategy. Here, we study a reasonably restricted version of this problem: the compact structural motif problem. In this paper, we prove that this restricted version is still NP-hard, and we present a polynomial-time approximation scheme to solve it. To the best of our knowledge, this is the first approximation algorithm with a guarantee ratio for the protein structural motif problem.

1 Introduction

Identifying structural motifs for proteins is a fundamental problem in computational biology. It has been widely accepted that during the evolution of proteins, structures are more conserved than sequences. In addition, structural motifs are thought to be tightly related to protein functions [1]. Thus, identifying the common substructures from a set of proteins can help us to know their evolutionary history and functions. With rapid growth in the number of structures in the Protein Data Bank (PDB) and protein folding methods, the need for fast and accurate structural comparison methods has become more and more crucial.

The multiple structural motif finding problem is the structural analogy of the sequence motif finding problem. For the former problem, the input consists of a set of protein structures in three-dimensional space, $\mathbf{R}^3$. The objective is to find a set of substructures, one from each protein, that exhibit the highest degree of similarity.

Roughly speaking, there are two main methods to measure the structural similarity, i.e., coordinate root mean squared deviation (cRMSD) and distance root mean squared deviation (dRMSD). The first one calculates the internal distance for each protein first, and compares this internal distance matrix for the input structures. In contrast, the second method uses the Euclidean distance between the corresponding residues from different protein structures. To do this, the optimal rigid transformation of these protein structures should be done first.

Various methods have been proposed to attack this problem under different similarity measuring strategies. Under the unit-vector RMSD (URMSD) measure, L.P Chew et al. [4] proposed an iterative algorithm to compute the consensus shape and proved the convergence of this algorithm. Applying graph-based data mining tools, D. Bandyopadhyay et al. [3] described a method to assign a protein structure to functional families using the family-specific fingerprints. Under the bottleneck metric similarity measure, M. Shatsky et al. [15] presented an algorithm for recognition of binding patterns common to a set of protein structures. This problem is also attacked in [5][6][10][13][14][18].

One of the closely related problems is the structural alignment problem, to which a lot of successful approaches have been developed. Among them, DALI [7] and CE [16] attempt to identify the alignment with minimal dRMSD, while STRUCTURAL [17] and TM-align [20] employ some heuristics to detect the alignment with minimal cRMSD.

However, the methods mentioned above are all heuristic; the solutions are not guaranteed to be optimal or near optimal. Recently, R. Kolodny et al. [9] proposed the first polynomial-time approximate algorithm for pairwise protein structure alignment based on the Lipschitz property of the scoring function. Though this method can be extended to the case of multiple protein structural alignment, the simple extension has the time complexity exponential in the number of proteins.

In this paper, we present an approximation algorithm employing the sampling technique [11] under the dRMSD measure. Adopting a reasonable assumption, we prove that our algorithm is efficient and produces a good approximation solution. In contrast to the method in [9], our algorithm is polynomial in the number of proteins. Furthermore, the sampling size is an adjustable parameter, adding more flexibility to our algorithm.

The rest of our manuscript is organized as follows. In Section 2, we introduce the notations and some background knowledge. In Section 3, we prove the NP-completeness of the compact structural motif problem. The algorithm, along with performance analysis, is given in Section 4.

2 Preliminaries

A protein consists of a sequence of amino acids (also called residues), each of which contains a few atoms including one C_α atom. In a protein structure, each atom is associated with a 3D coordinate. In this paper, we only take into consideration the C_α atom of a residue; thus a protein structure can be simplified as a sequence of 3D points. The common globular protein is generally compact, with the distance between two consecutive C_α atoms restrained by the bond length, and the volume of the bounding sphere linear in the number of residues.

A structural motif of a protein is a subset of its residues arranged by order of appearance, and its **length** is the number of elements in this subset. In this paper, we consider only the (R, C)**-compact motif**, which is bounded in the minimal ball B (refered to as **containing ball**) with radius at most R, and at most C residues in this ball do not belong to this motif.

To measure the similarity of two protein structures, a transformation, including rotation and translation, should be done first. Such a transformation is known as a rigid transformation and can be expressed with a 6D-vector $\tau = (t_x, t_y, t_z, r_1, r_2, r_3), r_1, r_2, r_3 \in [0, 2\pi], t_x, t_y, t_z \in \mathbf{R}$. Here, (t_x, t_y, t_z) denotes a translation and (r_1, r_2, r_3) denotes a rotation. Applying the transformation τ for a 3D-point u, we get a new 3D coordinate $\tau(u)$.

In this paper, we adopt $dRMSD$ to measure the similarity between two structural motifs. Formally, for two motifs $u = (u_1, u_2, \cdots, u_\ell)$ and $v = (v_1, v_2, \cdots, v_\ell)$, the $dRMSD$ distance between them is defined as $d(u, v) = \sum_{i=1}^{\ell} \|u_i - v_i\|^2$, where $\|.\|$ is the Euclidean distance.

Throughout this paper, we will study the following consensus problem:

(R, C)**-Compact Motif Problem:** Given n protein structures $\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_n$, and an integer ℓ, find a consensus with length ℓ: $q = (q_1, q_2, \ldots, q_\ell)$, where q_i is a point in $3D$, rigid transformation τ_i, and (R, C)-compact motif u_i of length ℓ for $\mathcal{P}_i$, $1 \le i \le n$, such that $\sum_{i=1}^{n} d(q, \tau_i(u_i))$ is minimized.

Before we present the approximation algorithm, we first prove that the transformation can be simplified for this special case. More specifically, it is unnecessary to consider all rigid translations; that is, we can consider only rotations without loss of generality.

The following lemma that is originally from [8] will be used in our proofs. Roughly speaking, it states that if we want to overlap two chains of points closely, we must make their centroids coincide.

Lemma 1. *Given n points $(a_1, a_2, \cdots, a_n)$ and n points $(b_1, b_2, \cdots, b_n)$ in 3D space, to minimize $\sum_i \|\rho(a_i) + T - b_i\|^2$, where ρ is a rotation matrix and T a translation vector, T must make the centroid of a_i and b_i coincide with each other.*

Lemma 2. *In the optimal solution of (R, C)-compact motif problem, the centroid of $\tau_i(u_i)$ must coincide with the centroid of q, for $1 \le i \le n$.*

This lemma is a direct corollary of Lemma 1. The basic idea is that to find the optimal rigid transformation, we can first translate the proteins so their centroids coincide with each other. Therefore, a transformation can be simplified to a vector (r_1, r_2, r_3), where $r_1,\ r_2,\ r_3 \in [0, 2\pi]$. For a real number ϵ, we can discretize the range $[0, 2\pi]$ into a series of bins with width of ϵ. We refer to this discrete transformation set $\{(i \cdot \epsilon, j \cdot \epsilon, k \cdot \epsilon) | 0 \leq i, j, k \leq \frac{2\pi}{\epsilon}\}$, where i, j, k are integers, as an **ϵ-net** of rotation space $\mathcal{T}$.

3 NP-Completeness Result

In this section, we will show that the (R, C)-compact motif problem is NP-hard. The reduction is from the sequence consensus problem, which has been proven to be NP-hard in [12].

Consensus Patterns (sum-of-pairs variant)[12]: Given a set of sequences over $\{0, 1\}$, $S = \{s_1, s_2, ..., s_n\}$, each of length m, and an integer ℓ, find a median string s of length ℓ and a substring t_i (consensus pattern) of length ℓ from each s_i, minimizing $\sum_{1 \leq i < j \leq n} d_H(s, t_i)$, where d_H is the Hamming distance.

Theorem 1. *The (R, C)-compact motif problem is NP-hard.*

Proof. For convenience, we adopt the following equivalent scoring function in our proof: $\sum_{1 \leq i < j \leq n} d(\tau_i(u_i),\ \tau_j(u_j))$. The equivalence is proved in [19].

For an instance of the Consensus Patterns problem, we will map each sequence s to a group of chains of 3D points through the following two steps:

extending step. For each of its ℓ-mer, say $M = s[i]s[i+1] \cdots s[i+\ell-1]$, we first attach its complement M' after it, then attach a tail of 2ℓ 0's and 2ℓ 1's. For example, 100 becomes 100011000000111111.

mapping step. Then we map this extended ℓ-mer to a chain of 3D points as follows: 0 is mapped to a residue at position $(0, 2i, 0)$, 1 is mapped to a residue at position $(1, 2i, 0)$. Therefore, each ℓ-mer is mapped to a chain of 6ℓ points at 2 positions. Note that the centroid of this chain is $(1/2, 2i, 0)$.

Hence, each of the n sequences, s_j, is mapped to a protein $\mathcal{P}_j$ with $6\ell(m-\ell+1)$ residues.

By our construction, each $(1, 0) - \mathit{compact}$ motif must be a chain of residues occupying 2 positions. From Lemma 2, it is easy to see that the optimal solution for this compact motif problem corresponds to the optimal solution for the original Consensus Patterns problem. Thus, the NP-hardness of the Compact Motif problem is proven. Q.E.D.

4 (R, C)-Compact Motif Finding Algorithm

The basic idea of our algorithm is as follows: first, we translate the proteins to make their centroids coincide. Then, for each discrete rigid transformation and each r-tuple of compact motifs, we calculate the median, and find from each protein the closest part to this median. The median with the minimal value of the objective function value is output.

Let $f(x) = \sum_{i=1}^{n} (x - a_i)^2$. It is easy to see that $f(x)$ is minimized when x equals the average of $\{a_i\}$. The following lemma states that if we randomly choose r number from $\{a_i\}$, and let x equals the average of them, then the expected value of $f(x)$ is $1 + 1/r$ times the minimum.

```
(R, C)-Compact Motif Finding Algorithm
Input: n protein structures $\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_n$, integer $\ell, C, r$, real number $R, \epsilon$.
Output: median consensus $u$ of length $\ell$, rigid transformation $\tau_i$, $(R,C)$-compact motif $u_i$ of length $\ell$ for $\mathcal{P}_i$, for $1 \le i \le n$.
1. Fix $\mathcal{P}_1$, translate other proteins to make their centroids coincide with that of $\mathcal{P}_1$
2. FOR every $r$ length-$\ell$ $(R,C)$-compact motif $u_1, u_2, \cdots, u_r$, where $u_i$ is a motif of some $\mathcal{P}_j$ DO
3.    FOR every $r-1$ transformations $\tau_2, \tau_3, \cdots, \tau_r$ from $\epsilon/Rn\ell$-net of rotation space $\mathcal{T}$ DO
        (a) Find the average of $u_1, \tau_2(u_2), \cdots, \tau_r(u_r)$: $u = (u_1 + \tau_2(u_2) + \cdots + \tau_r(u_r))/r$
        (b) FOR $i = 1, 2, \cdots, n$ DO
              Find the length-$\ell$ $(R,C)$-compact motif $v_i$ of $\mathcal{P}_i$ and its optimal rigid transformation $\tau'_i$ that minimize $d(u, \tau'_i(v_i))$.
        (c) Let $c(u) = \sum_{i=1}^n d(u, \tau'_i(v_i))$.
4. Output the $u$ and the corresponding $v_i$, $\tau'_i$ that minimize $c(u)$.
```

Lemma 3. *Let $a_1, a_2, ..., a_n$ be n real numbers, $1 \le r \le n$ is an integer. Then the following equation holds:*

$$\frac{1}{n^r}\sum_{1\le i_1,i_2,\ldots,i_r\le n}\sum_{i=1}^{n}\left(\frac{a_{i_1}+a_{i_2}+\ldots+a_{i_r}}{r}-a_i\right)^2=\frac{r+1}{r}\sum_{i=1}^{n}\left(\frac{a_1+a_2+\ldots+a_n}{n}-a_i\right)^2.$$

Proof. For the sake of simplicity, we use σ to denote $\sum_{i=1}^n a_i$ and σ' denote $\sum_{i=1}^n a_i^2$.

$$\begin{aligned}
&\frac{1}{n^r}\sum_{1\le i_1,i_2,\ldots,i_r\le n}\left(n\frac{(a_{i_1}+a_{i_2}+\ldots+a_{i_r})^2}{r^2}-2\frac{a_{i_1}+a_{i_2}+\ldots+a_{i_r}}{r}\sigma+\sigma'\right)\\
&=\frac{1}{n^r}\sum_{1\le i_1,i_2,\ldots,i_r\le n} n\frac{a_{i_1}^2+a_{i_2}^2+\ldots+a_{i_r}^2+2(a_{i_1}a_{i_2}+\cdots+a_{i_{r-1}}a_{i_r})}{r^2}\\
&\quad-\frac{2rn^{r-1}\sigma}{rn^r}\sigma+\sigma'\\
&=\frac{rn^{r-1}\sigma'+r(r-1)n^{r-2}\sigma^2}{r^2n^{r-1}}-\frac{2\sigma^2}{n}+\sigma'\\
&=\frac{\sigma'}{r}+\frac{r-1}{r}\frac{\sigma^2}{n}--\frac{2\sigma^2}{n}+\sigma'\\
&=\frac{r+1}{r}\left(\sigma'-\frac{\sigma^2}{n}\right)\\
&=\frac{r+1}{r}\left(\frac{\sigma^2}{n}-2\frac{\sigma^2}{n}+\sigma'\right)
\end{aligned}$$

$$= \frac{r+1}{r} \sum_{i=1}^{n} (\frac{\sigma}{n} - a_i)^2$$

Q.E.D.

The following lemma is needed for our analysis of the time complexity:

Lemma 4. *All of the (R, C)-compact motifs of length ℓ for protein $\mathcal{P}$ with m residues can be enumerated in $O(m^5 \ell^c)$ time.*

Proof. According to the definition of the (R, C)-compact motif, we know that the containing ball B of a motif must contain ℓ to $\ell + C$ residues of $\mathcal{P}$. In addition, it is easy to see that either there are 4 residues on the surface of B, or there are 3 residues on its surface and the radius of B is R, due to the minimality of B. Therefore, to enumerate the compact motifs, we can first enumerate the containing balls, which takes $O(m^5)$; then from each ball, we enumerate the motifs, which takes $O(\ell^c)$ times. In total, it takes $O(m^5 \ell^c)$ time. Q.E.D.

Theorem 2. *The (R, C)-Compact Motif Finding Algorithm outputs a solution with cost no more than*

$$(1 + \frac{1}{r})c_{opt} + O(\epsilon),$$

in time $O(n^{4r-2} m^{5r+5} R^{3r-3} \ell^{cr+c+3r-2} / \epsilon^{3r-3})$, where c_{opt} is the cost of the optimal solution.

Proof. Step 1 takes $O(nm)$ time. The enumeration of $\{u_i\}$ takes $O(n^r (m^5 \ell^c)^r)$ time, $\{\tau_i\}$ takes $O((\frac{Rn\ell}{\epsilon})^{3(r-1)})$. Step 3(a)-(c) takes $O(n \cdot m^5 \ell^c \cdot \ell)$ time (finding τ'_i takes $O(\ell)$ time according to [2]). So, the time complexity of the algorithm is $O(n^{4r-2} m^{5r+5} R^{3r-3} \ell^{cr+c+3r-2} / \epsilon^{3r-3})$.

Now, we prove the performance ratio. Given an instance of the problem, we use u^* to denote the optimal median; v_i^* and τ_i^* denote the optimal motif in $\mathcal{P}_i$ and corresponding optimal rigid transformation, respectively. Then we have $c_{opt} = \sum_{i=1}^{n} d(u^*, \tau_i^*(v_i^*))$. By the property of our cost function, it is easy to see that u^* is the average of $\tau_1^*(v_1^*), \tau_2^*(v_2^*), \cdots, \tau_n^*(v_n^*)$, i.e., $u^* = (\tau_1^*(v_1^*) + \tau_2^*(v_2^*) + \cdots + \tau_n^*(v_n^*))/n$.

First, we claim that c_{opt} can be approximated by sampling r proteins. In particular, we will show that there exist $1 \le i_1, i_2, ..., i_r \le n$ s.t.

$$\sum_{i=1}^{n} d(u^*_{i_1 i_2 \ldots i_r}, \tau_i^*(v_i^*)) \le (1 + 1/r)c_{opt}, \tag{1}$$

where $u^*_{i_1 i_2 \ldots i_r} = (\tau_{i_1}^*(v_{i_1}^*) + \tau_{i_2}^*(v_{i_2}^*) + \cdots + \tau_{i_r}^*(v_{i_r}^*))/r$. It suffices to prove that the average of such value for all $1 \le i_1, i_2, ..., i_r \le n$ is $(1 + 1/r)c_{opt}$, which can be easily deduced from Lemma 3.

For $1 \le i \le n$, let τ'_i be a rotation in $\epsilon/Rn\ell$-net (remember R is the maximum radius of the motifs) of $\mathcal{T}$ that is closest to τ_i^* in $\mathcal{T}$. Then τ_i^* can be reached from τ'_i by moving at most $\epsilon/2Rn\ell$ along each of the three dimensions. Let $u'_{i_1 i_2 \ldots i_r} = (\tau'_{i_1}(v_{i_1}^*) + \tau'_{i_2}(v_{i_2}^*) + \cdots + \tau'_{i_r}(v_{i_r}^*))/r$.

Now we will prove that $\sum_{i=1}^{n} d(u'_{i_1 i_2 \ldots i_r}, \tau'_i(v_i^*)) \le \sum_{i=1}^{n} d(u^*_{i_1 i_2 \ldots i_r}, \tau_i^*(v_i^*)) + O(\epsilon)$.

By our choice of τ'_{i_j}, it is easy to see that $\|\tau'_i(v^*_i)-\tau^*_i(v^*_i)\| \le \epsilon/Rn\ell$, $\|u'_{i_1i_2\ldots i_r}-u^*_{i_1i_2\ldots i_r}\| \le \epsilon/Rn\ell$ (more details can be found in [9]), let $u^*_{i_1i_2\ldots i_r}[j]$ be the j-th node of $u^*_{i_1i_2\ldots i_r}$, $v^*_i[j]$ be the j-th node of v^*_i, then we have

$$\begin{aligned}
&\sum_{i=1}^{n} d(u'_{i_1i_2\ldots i_r},\tau'_i(v^*_i)) = \sum_{i=1}^{n}\sum_{j=1}^{\ell} \|u'_{i_1i_2\ldots i_r}[j]-\tau'_i(v^*_i[j])\|^2\\
&= \sum_{i=1}^{n}\sum_{j=1}^{\ell} \|u'_{i_1i_2\ldots i_r}[j]-u^*_{i_1i_2\ldots i_r}[j]+u^*_{i_1i_2\ldots i_r}[j]-\tau^*_i(v^*_i[j])+\tau^*_i(v^*_i[j])\\
&\quad -\tau'_i(v^*_i[j])\|^2\\
&\le \sum_{i=1}^{n}\sum_{j=1}^{\ell} (\|u^*_{i_1i_2\ldots i_r}[j]-\tau^*_i(v^*_i[j])\| + (\|u'_{i_1i_2\ldots i_r}[j]-u^*_{i_1i_2\ldots i_r}[j]\| + \|\tau^*_i(v^*_i[j])\\
&\quad -\tau'_i(v^*_i[j])\|))^2\\
&= \sum_{i=1}^{n}\sum_{j=1}^{\ell} (\|u^*_{i_1i_2\ldots i_r}[j]-\tau^*_i(v^*_i[j])\|^2 + (\|u'_{i_1i_2\ldots i_r}[j]-u^*_{i_1i_2\ldots i_r}[j]\| + \|\tau^*_i(v^*_i[j])\\
&\quad -\tau'_i(v^*_i[j])\|)^2 + 2\|u^*_{i_1i_2\ldots i_r}[j]-\tau^*_i(v^*_i[j])\|(\|u'_{i_1i_2\ldots i_r}[j]-u^*_{i_1i_2\ldots i_r}[j]\|\\
&\quad +\|\tau^*_i(v^*_i[j])-\tau'_i(v^*_i[j])\|))\\
&\le \sum_{i=1}^{n} d(u^*_{i_1i_2\ldots i_r},\tau^*_i(v^*_i)) + O(\epsilon) + 4\epsilon^2/Rn\ell\\
&= \sum_{i=1}^{n} d(u^*_{i_1i_2\ldots i_r},\tau^*_i(v^*_i)) + O(\epsilon).
\end{aligned}$$

It is easy to see that the output of our algorithm is at least as good as $\sum_{i=1}^{n} d(u'_{i_1i_2\ldots i_r},\tau'_i(v^*_i))$. Together with (1), the performance ratio of our algorithm is proven. Q.E.D

5 Conclusion

In this paper, we present a sampling-based approximation algorithm for the problem of finding the compact consensus shape from a family of proteins. Our algorithm requires that the consensus pattern satisfies the compactness condition. To find a good algorithm in more general case is an interesting problem.

References

1. Aloy, P., Querol, E., Aviles, F.X., Sternberg, M.J.: Automated structure-based prediction of functional sites in proteins: Applications to assessing the validity of inheriting protein function from homology in genome annotation and to protein docking. Journal of Molecular Biology 311, 395–C408 (2001)

2. Arun, K.S., Huang, T.S., Blostein, S.D.: Least square fitting of two 3-d point sets. IEEE Transactions on Pattern Analysis and Machine Intelligence 9(5), 698–700 (1987)
3. Bandyopadhyay, D., Huan, J., Liu, J., Prins, J., Snoeyink, J., Wang, W., Tropsha, A.: Structure-based function inference using protein family-specific fingerprints. Journal of Protein Science 15, 1537–1543 (2006)
4. Chew, L.P., Kedem, K.: Finding the consensus shape of a protein family. In: Proc. 18th Annual ACM Symposium on Computational Geometry, pp. 64–73 (2002)
5. Gelfand, I., Kister, A., Kulikowski, C., Stoyanov, O.: Geometric invariant core for the vl and vh domains of immunoglobulin molecules. Protein Engineering 11, 1015–1025 (1998)
6. Gerstein, M., Altman, R.B.: Average core structure and variability measures for protein families: application to the immunoglobins. Journal of Molecular Biology 112, 535–542 (1995)
7. Holm, L., Sander, C.: Dali: a network tool for protein structure comparison. Trends Biochem Sci. 20(11), 478–480 (1995)
8. Huang, T.S., Blostein, S.D., Margerum, E.A.: Least-square estimation of motion parameters from 3-d point correspondences. In: Proc of the IEEE Conference on Computer Vision and Pattern Recognition, vol. 69 pp. 198–201 (1986)
9. Kolodny, R., Linial, N.: Approximate protein structural alignment in polynomial time. Proc. Natl Acad. Sci. 101, 12201–12206 (2004)
10. Leibowitz, N., Fligelman, Z.Y., Nussinov, R.: Multiple structural alignment and core detection by geometric hashing. In: Proc. 7th Int. Conf. Intell. Sys. Mol. Biol, pp. 169–177 (1999)
11. Li, M., Ma, B., Wang, L.: Finding similar regions in many strings. In: Proceedings of the thirty-first annual ACM symposium on Theory of computing (STOC), pp. 473–482, Atlanta (May 1999)
12. Moan, C., Rusu, I.: Hard problems in similarity searching. Discrete Appl. Math. 144(1-2), 213–227 (2004)
13. Orengo, C.: Cora-topological fingerprints for protein structural familie. Protein Science 8, 699–715 (1999)
14. Orengo, C., Taylor, W.: Ssap: Sequential structure alignment program for protein structure comparison. Methods in Enzymology 266, 617–635 (1996)
15. Shatsky, M., Shulman-Peleg, A., Nussinov, R., Wolfson, H.: The multiple common point set problem and its application to molecule binding pattern detection. Journal of Computational Biology 13(2), 407–428 (2006)
16. Shindyalov, I.N., Bourne, P.E.: Protein structure alignment by incremental combinatorial extension ce of the optimal path. Protein Eng. 11(9), 739–747 (1998)
17. Subbiah, S., Laurents, D.V., Levitt, M.: Structural similarity of dna-binding domains ofbacteriophage repressors and the globin core. Current Biology 3, 141–148 (1993)
18. Xu, J., Jiao, F., Berger, B.: A parameterized algorithm for protein structure alignment. In: RECOMB, pp. 488–499 (2006)
19. Ye, J., Janardan, R.: Approximate multiple protein structure alignment using the sum-of-pairs distance. Journal of Computational Biology 11(5), 986–1000 (2004)
20. Zhang, Y., Skolnick, J.: Tm-align: a protein structure alignment algorithm based on the tm-score. Nucleic Acids Research 33, 2302–2309 (2005)

Improved Algorithms for Inferring the Minimum Mosaic of a Set of Recombinants

Yufeng Wu and Dan Gusfield

Department of Computer Science
University of California, Davis
Davis, CA 95616, U.S.A.
{wuyu,gusfield}@cs.ucdavis.edu

Abstract. Detecting historical recombination is an important computational problem which has received great attention recently. Due to recombination, input sequences form a mosaic, where each input sequence is composed of segments from founder sequences. In this paper, we present improved algorithms for the problem of finding the minimum mosaic (a mosaic containing the **fewest** ancestral segments) of a set of recombinant sequences. This problem was first formulated in [15], where an exponential-time algorithm was described. It is also known that a restricted version of this problem (assuming recombination occurs only at predefined block boundaries) can be solved in polynomial time [15,11]. We give a polynomial-time algorithm for a special case of the minimum (blockless) mosaic problem, and a practical algorithm for the general case. Experiments with our method show that it is practical in a range of data much larger than could be handled by the algorithm described in [15].

1 Introduction

A grand challenge for post-genomic era is dissecting the genetic basis of complex diseases. An important connection between the sequences (the genotypes) and the traits of interest (the phenotypes) is the evolutionary history (the genealogy) of the chosen individuals. Thus, inferring genealogy from sequences has received much attention recently. A major difficulty in inferring genealogy is meiotic recombination, one of the principal evolutionary forces responsible for shaping genetic variation within species. Efforts to deduce patterns of historical recombination or to estimate the frequency or the location of recombination are central to modern-day genetics.

A central genetic model used throughout this paper (and used before in [15]) is that the current population evolved from a *small* number of founder sequences. Over time, recombination broke down ancestral sequences and thus a current sequence is a concatenation of segments from the founder set. The set of input sequences then looks like a **mosaic** of segments from the founder sequences, when sequences are arranged as aligned rows. Thus, we refer the model as the *mosaic model*. See Figure 1 for an illustration. The biological literature contains validations of this model. For example, it is stated in the Nature paper [14] that "The *Ferroplasma* type II genome seems to be a composite from three ancestral strains that have undergone homologous recombination to form a large population of mosaic genomes."

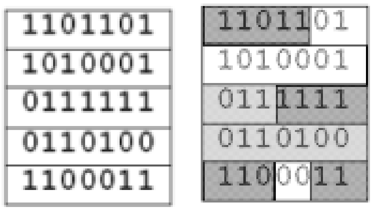
The mosaic pattern is potentially very informative in understanding the population evolution. The mosaic tells which sequences inherit their DNA from the same founder at a genomic site, and thus can be very useful in understanding the genetic basis of traits. In the context of inferring haplotypes from genotypes (i.e. the haplotype inference problem, also called phasing problem), Hidden Markov Model (HMM) based probabilistic approaches which exploit the mosaic patterns have been actively studied [3,9,10]. Therefore, understanding the genomic mosaic structure is an interesting problem, and better understanding of the mosaic pattern may be useful for population genetics problems.

Unfortunately, the mosaic boundaries (called *breakpoints*) are not readily seen from the sequences, and so we have the problem of inferring the true breakpoints (and the ancestral founder sequences) for the given input sequences. The breakpoints break the given sequences into segments of (possibly inexact) copies of ancestral materials that are inherited from some founder sequences of the population. The inexact copies of ancestral materials are often due to point mutations at nucleotide sites. In the context of recent human populations, however, the assumption is that the time period is short and the point mutation rates are low [15,11,1,2]. Hence, we assume throughout the paper that the input sequences inherit *exact* copies of ancestral material between two neighboring breakpoints. So every input sequence is a concatenation of segments of some founder sequences. Since there are a huge number of possible mosaic patterns for a set of input sequences, we need a biologically meaningful model to infer breakpoints and founders.

In 2002, Ukkonen [15] proposed a computational problem based on the mosaic model, given input of n binary sequences with m columns each. The model assume that the population evolves from a set of relatively small number of founders. The natural parsimonious objective is to construct a mosaic with fewest breakpoints. This motivates the following optimization problem.

The Minimum Mosaic Problem . Given n input sequences (each with m columns) and a number K_f, find K_f founder sequences that *minimize* the total number of breakpoints needed to be put in the input sequences, which break the input sequences into segments from the founder sequences. See Figure 1 for an example. The Minimum Mosaic Problem has also been turned into a graphical game, called the Haplotye Threading Game, developed at the University of North Carolina [5].

It is important to emphasize that we require each segment to be derived from the corresponding aligned positions of a founder sequence, although the breakpoints do not need to be the same in each of the input sequences. Also note that once founder sequences are known, it is straightforward (using e.g. a method in [12]) to place breakpoints in the input sequences, so that the number of breakpoints is minimized for *each* sequence and thus also for all input sequences together.



(a) Haplotypes (b) Mosaic

Fig. 1. An example illustrating the minimum mosaic problem on binary sequences. Figure 1(a) shows the input sequences. Figure 1(b) shows one way to partition the sequences in Figure 1(a) into segments, such that each segment comes from one of *three* founders: 0110100, 1101111 and 1010001. Note that there are totally four breakpoints, which is the minimum over all possible solutions with three founders.

In [15] (and also [11]), efficient algorithms were developed for a related but different problem. In addition, Ukkonen [15] also described a dynamic programming algorithm for the minimum mosaic problem described above. But the algorithm given in [15] does not scale well when the number of founders or the size of input matrix grows. Another unaddressed question is how to deal with *genotypes* (to be defined later), since most current biological data comes in the form of genotypes.

Our contributions. This paper focuses on the combinatorial properties of the minimum mosaic problem, on which *little* progress has been made since the work of Ukkonen [15]. We report on two main results.

1. For the special case where there are *two* founders, we show the minimum mosaic problem can be solved in $O(mn)$ time [1]. We also give an efficient algorithm for finding the minimum breakpoints when the input sequences consist of *genotype* data (instead of haplotype data).
2. For the general minimum mosaic problem, we present an efficiently computable lower bound on the minimum number of breakpoints. We also develop an algorithm which solves the minimum mosaic problem exactly. Simulations show that this method is practical when the number of founders is small, and the numbers of rows and columns are moderate.

1.1 Additional Definitions

In diploid organisms (such as humans) there are two (not completely identical) "copies" of each chromosome, and hence of each region of interest. A description of the data from a single copy is called a *haplotype*, while a description of the conflated (mixed) data on the two copies is called a *genotype*. Today, the underlying data that forms a haplotype is usually a vector of values of m *single nucleotide*

[1] Note that the algorithm proposed by Ukkonen [15] is also implicitly polynomial-time when $K_f = 2$. The advantage of our method is that we establish an easily-verified condition to construct a minimum mosaic for two founder sequences.

polymorphisms (SNP's). A SNP is a single nucleotide site where exactly two (of four) different nucleotides occur in a large percentage of the population. Genotype data is represented as n n by m 0-1-2 (ternary) matrix G. Each row is a genotype. A pair of binary vectors of length m (haplotypes) *generate* a row i of G if for every position c both entries in the haplotypes are 0 (or 1) if and only if $G(i,c)$ is 0 (or 1) respectively, and exactly one entry is 1 and one is 0 if and only if $G(i,c) = 2$.

Given an input set of n genotype vectors (i.e. matrix) G of length m, the *Haplotype Inference (HI) Problem* is to find a set (or matrix) H of n pairs of binary vectors (with values 0 and 1), one pair for each genotype vector, such that each genotype vector in G is generated by the associated pair of haplotypes in H. H is called an "HI solution for G". Genotype data is also called "unphased data", and the decision on whether to expand a 2 entry in G to $[{0 \atop 1}]$ or to $[{1 \atop 0}]$ in H, is called a "phasing" of that entry. The way that all the 2's in a column (also called a site) are expanded is called the phasing of the column (*site*). Note that if a genotype is 02 at two sites, we know the two haplotypes in an HI solution will be 00 and 01 at these two positions. We sometimes call this case *trivial* for these two sites. If a genotype is 22 instead, the HI solutions are *ambiguous*: an HI solution may be either 00/11 or 01/10 at the two sites.

Each input row r inherits a state at a site s from a particular founder. We say this founder is ancestral to r at site s.

2 The Two-Founder Case

We consider the special case where there are only two founders. Note that for *any* haplotype data H, there exists two founders that can derive H in a mosaic [15]: a trivial set of two founders consists an all-0 sequence and an all-1 sequence. However, it is not immediately clear which pair of founder sequences leads to a minimum mosaic.

2.1 Solution for Haplotype Data Input

For a haplotype matrix H at two sites s_i and s_j, there are four possible states (called gametes): 00, 01, 10, 11. We use $n_{i,j,g}$ to denote the number of times that a gamete g appears in H for two sites s_i and s_j. As an example, for the first two sites (sites 1 and 2) of the data in Figure 1, $n_{1,2,00} = 0$, $n_{1,2,01} = 2$, $n_{1,2,10} = 1$ and $n_{1,2,11} = 2$.

It is easy to see that we can remove from input sequences any site that is uniform (i.e. either all 0 or all 1). This will not reduce the minimum number of breakpoints in a minimum mosaic. Hence we assume there are two *founder* states at any site, one is 0 and the other is 1. We can also remove any site i which is identical to site $i+1$. A key observation is: at two neighboring sites s_i, s_{i+1}, we will have either $00, 11$ gametes or $01, 10$ gametes for the two *founder* sequences. We define the *distance* between a sites s_i and its neighboring site to the right s_{i+1} in H as $d_i = MIN(n_{i,i+1,00} + n_{i,i+1,11}, n_{i,i+1,10} + n_{i,i+1,01})$. We have the following simple lemma.

Lemma 1. *The minimum number of breakpoints between two neighboring sites s_i, s_{i+1} is at least d_i.*

Proof. Since the two founders have either 00/11 or 01/10 gametes at sites s_i and s_{i+1}, either there is a breakpoint between s_i and s_{i+1} for every gamete 01 and 10 (if the founders have 00/11 states at sites s_i and s_{i+1}), or between s_i and s_{i+1} for every gamete 00 and 11 (if the founders have 01/10 states). □

The above lemma implies that the minimum number of breakpoints is at least $n_{tb} = \sum_{i=1}^{m-1} d_i$. On the other hand, the following algorithm finds two founders that derive the input sequences with exactly n_{tb} breakpoints.

Algorithm 1. Polynomial-time algorithm for finding two founders F_1, F_2 that gives the minimum number of breakpoints

```
1. Let $F_1[1] \leftarrow 0$, and $F_2[1] \leftarrow 1$. And set $i \leftarrow 1$.
2. while $i \leq m - 1$
2.1. If $n_{i,i+1,00} + n_{i,i+1,11} \geq n_{i,i+1,10} + n_{i,i+1,01}$, then $F_1[i+1] = F_1[i]$, and $F_2[i+1] = F_2[i]$.
2.2. Otherwise, $F_1[i+1] = 1 - F_1[i]$, and $F_2[i+1] = 1 - F_2[i]$.
2.3 $i \leftarrow i + 1$
```

It is easy to verify that the above algorithm produces two founder sequences using exactly n_{tb} breakpoints. Intuitively, Algorithm 1 gives the optimal solution to the minimum mosaic problem by constructing founders from left to right. At each position (other than the leftmost site, i.e. s_1) the algorithm is only constrained by the single site to its immediate left. This means it can *always* choose a state to introduce exactly d_i breakpoints for each s_i and s_{i+1}. Thus, the solution is optimal due to Lemma 1. The running time of the algorithm is $O(mn)$. Thus, we have:

Proposition 1. *When $K_f = 2$, the minimum mosaic problem can be solved for haplotype data H in $O(nm)$ time.*

2.2 Solution for Genotype Data Input

Now we consider genotypes (not haplotypes) as input. This problem is important because most currently available biological data is genotypic. With genotype data, the minimum problem can be formulated as follows.

The minimum mosaic problem with genotypes. Given a genotype matrix G and a number K_f, find an HI solution H and K_f founder sequences such that the number of breakpoints needed to derive H from the founders is minimized among *all* possible HI solutions and K_f founder sequences.

We give a polynomial-time algorithm for the special case of $K_f = 2$. We begin with a lemma which extends Lemma 1 to genotypes. Note that for two genotypic sites i and j, the possible states are: $00, 01, 10, 11, 02, 20, 12, 21$, and 22. Similar to the haplotype case, we denote the number of times that gamete g appears at two sites i and j as $n_{i,j,g}$. We define the *distance* between two genotypic sites s_i and its right neighbor s_{i+1} as $d_i^g = MIN(2n_{i,i+1,00} + n_{i,i+1,02} + n_{i,i+1,20} + 2n_{i,i+1,11} + n_{i,i+1,12} + n_{i,i+1,21},\ 2n_{i,i+1,01} + n_{i,i+1,02} + n_{i,i+1,21} + 2n_{i,i+1,10} + n_{i,i+1,20} + n_{i,i+1,12})$.

Lemma 2. *The minimum number of breakpoints between two neighboring genotypic sites s_i, s_{i+1} is at least d_i^g.*

Proof. Note that d_i^g represents the *minimum* number of gametes 00/11 and gametes 01/10 for *all* possible ways of phasing these two sites. Note that one can always phase a "22" gamete at sites s_i and s_{i+1} to *agree* exactly with the two founders at s_i, s_{i+1}. Therefore, following the same idea in the proof of Lemma 1, it is easy to see d_i^g is a lower bound on the number of breakpoints between sites i and $i + 1$. □

Proposition 2. *When $K_f = 2$, the minimum mosaic problem with genotypes can be solved in $O(nm)$ time.*

Proof. Using a similar idea as in Proposition 1, the two-founder minimum mosaic problem can also be solved efficiently even when the input is genotypic. The number of minimum breakpoints is equal to $\sum_{i=1}^{m-1} d_i^g$. This can be done as follows.

We can construct two founders using a procedure similar to Algorithm 1. A small difference is that here we use the smaller term in d_i^g (rather than d_i) to decide whether to let founders have gametes 00/11 or 01/10. Now that we have constructed two founders, we derive an HI solution H as follows. We start from the leftmost site and move to the right by one site each time. We pick any feasible phasing for the leftmost site. Now we consider site s_{i+1} by assuming s_i has been properly phased. Note that the only ambiguous rows at sites s_i and s_{i+1} are those containing 22. We phase 22 so that the phased gametes agree with the founder states at sites s_i and s_{i+1}.

The only subtle issue left is whether there are will ever be an inconsistency during the process. That is, will we be prohibited from phasing s_{i+1} in the way described, due to the phasing of site s_i. But inconsistency will not occur. For one row r of G, suppose the above procedure dictates the two 2's in s_i, s_{i+1} are phased to 01 and 10. If column s_i (for row r) has been phased as $[\begin{smallmatrix}0\\1\end{smallmatrix}]$ we phase s_{i+1} (for row r) as $[\begin{smallmatrix}1\\0\end{smallmatrix}]$. Otherwise, we phase s_{i+1} as $[\begin{smallmatrix}0\\1\end{smallmatrix}]$. In either case, we will produce the needed binary pairs in sites s_i, s_{i+1} for row r. □

2.3 The Minimum Mosaic Problem with Unknown Site Order

We consider here a variation of the two-founder minimum mosaic problem, where the linear order of the m sites is *unknown*. Formally, we define an optimization problem as follows.

The two-founder minimum mosaic with permutation problem. Given a matrix M, we want to find a permutation Π of the sites, and two founder sequences, such that with those founders and by ordering the sites according to Π, the number of breakpoints used is the minimum over all possible site permutations and all possible pair of founders.

A Biological motivation. One biological motivation for allowing site permutation is the *linkage mapping* problem, which is to find the true ordering of multiple loci on a chromosome. Linkage maps remain important for species which have not yet been sequenced. See the recent paper [13] for a discussion of current interest in linkage mapping and a detailed explanation of computational issues involved in linkage mapping. In order to infer the true site ordering, a natural approach is to find the ordering of sites, and a small set of founders, so that the number of needed breakpoints is minimized with that number of founders.

We establish an interesting connection to the metric traveling salesman problem, which implies a 1.5-approximation solution to the minimum mosaic with permutation problem. Details are omitted due to lack of space.

3 The Case of Three or More Founders

When the number of founders is at least three, we do not have a polynomial-time algorithm for the minimum mosaic problem, although we conjecture that there is one. In this section, we first describe an efficiently computable lower bound on the minimum number of breakpoints for any fixed K_f with haplotype data. We also develop an algorithm that solves the minimum mosaic problem exactly. In our testing, the method is practical for many problem instances when the number of founders is three or four and the size of input matrix is moderate (e.g. with 50 sequences and 50 sites).

3.1 Lower Bound on the Number of Breakpoints for Haplotype Data

We now describe a simple lower bound on the minimum mosaic problem, inspired by the "composite haplotype bound", a lower bound developed for a different recombination model [7]. Consider a binary matrix H. We collect the set S of distinct rows together with their multiplicities (denoted as $(s_i, n_i) \in S$). Here, n_i records the number of times s_i appears in the input. We order S so that $\{n_i\}$ is *non-increasing*. If $|S| \leq K_f$, then the lower bound (denoted B_m) on the minimum number of breakpoints is simply 0. Otherwise, $\sum_{i=K_f+1}^{|S|} n_i$ is a lower bound on the minimum number of breakpoints.

This is a trivial bound. But Myers and Griffiths [7] introduced a general method (called the *composite method*) to *amplify* weaker bounds, to get much higher overall lower bounds. We apply the composite method to the minimum mosaic problem. Instead of computing a lower bound on the whole matrix, we compute lower bound B_m for each of the $\binom{m}{2}$ intervals, each with the above idea. Then we combine these bounds to form a *composite* bound as detailed in [7,6]. This improves the lower bounds, demonstrated by our empirical studies in Section 4. One of our major results in this paper is the demonstration of the effectiveness of the composite lower bound for the minimum mosaic problem.

3.2 Exact Method for the Minimum Mosaic Problem When $K_f \geq 3$

When $K_f \geq 3$, no polynomial-time algorithm is known for the minimum mosaic problem for either haplotypes or genotypes. Here we develop a method that solves the minimum mosaic problem exactly, and give heuristics that make it practical for a range of data of current biological interest. Simulation shows that our method works well for a large range of problem instances when $K_f = 3$, and for medium-size data (say 50 by 50 matrix) when $K_f = 4$. We describe our method for haplotypes, but remark that the method can be modified to handle genotype data. Practical performance of the method in [15] was not demonstrated there, but the method was implemented in program *haplovisual* (http://www.cs.helsinki.fi/u/prastas/haplovisual). Direct implementation of Ukkonen's method is expected to be prohibitive when n and m increase, even for a small number of founders. Our initial experiments with program *haplovisual* suggest it is not practical for 20 or more rows and three or more founders.

We start by developing some notation and terminology. The choice of binary states for each of the K_f founders at a site i is called the "founder setting at site i", and denoted $f(i)$. There are $2^{K_f} - 2$ possible founder settings at a site, assuming each site contains both 0's and 1's. A combined founder setting at each of the sites from 1 to i is denoted $F(i)$ and called a "founder setting up to i"; a founder setting up to m is denoted F and is called a "full founder setting". The founder setting at site i together with a legal mapping of input sequences to the K_f founders is called the "configuration at site i". A mapping is legal for site i if the state of each input sequence equals the state, at site i, of the founder it is mapped to. Clearly, given configurations at sites i and $i+1$, the number of breakpoints that occur between these two sites is the number of input sequences mapped to different founders at sites i and $i+1$, which is at most n. A combined configurations at each of the sites from 1 to i is denoted $C(i)$ and called a "configuration up to i", and the founder setting up to m is called the "full configuration".

Given a founder setting F, the problem of finding a full configuration that minimizes the number of breakpoints is called the "CF problem". Given F and i, the problem of finding a configuration $C(i)$ up to i to minimize the number of breakpoints in $F(i)$ is called the $CF(i)$ problem. Problem CF can be solved by a simple greedy algorithm [12] that is run *independently* for each input sequence s as follows. To start, set a variable p_s to 1 and find the longest match, starting at site p_s, between s and any of the founder sequences. If the longest match extends to site i and occurs between s and founder q, then map each site from 1 to i in s to founder q. If there are ties for longest match, q can be set to be any one of the tied founders. Next, set p_s to $i+1$ and iterate. Continue until the end of s is reached.

Now suppose that a full founder setting F, and the input sequences, are only given to the greedy algorithm one site at a time, in increasing order. Then the greedy algorithm doesn't know the full length of any match between a founder sequence and an input sequence. However, the CF problem can be solved with a "locally greedy algorithm" that implicitly records all the possible actions of the greedy algorithm on each $F(i)$. Since the greedy algorithm considers each input sequence separately, we describe the locally-greedy algorithm for one input sequence s. At each site i, the locally-greedy algorithm records a subset $SF_s(i)$ of founders, and a number $BF_s(i)$. To begin, let x denote the state of sequence s at site 1. The original greedy algorithm would map s to one of the founders that has state x at site 1, so in the locally-greedy algorithm we let $SF_s(1)$ be the set of all founders which have state x at site 1, and set $BF_s(1)$ to zero. For $i > 1$, let $A_s(i)$ be the subset of founders whose state at site i agrees with the state of s at site i. Then $SF_s(i) = SF_s(i-1) \cap A_s(i)$, and $BF_s(i) = BF_s(i-1)$, if the intersection is non-empty; otherwise, $SF_s(i) = A_s(i)$, and $BF_s(i) = BF_s(i-1) + 1$. $BF_s(m)$ is the number of breakpoints in the optimal solution to problem CF, given the full founder setting F. It is also easy to reconstruct the optimal configuration by a backwards trace from m to 1. Note that at each i, the SF sets compactly and implicitly encode all the optimal configurations for the $CF(i)$ problem that the greedy algorithm could find. Note also that the locally-greedy algorithm not only solves the CF problem, given F, but also solves each of the $CF(i)$ problems implied by each $F(i)$.

We now describe our method to solve the minimum mosaic problem; the method must find both an optimal F and a solution to the implied CF problem. At the high level, before optimizations to significantly speed it up, the method enumerates all possible founder settings $F(i)$, for i from 1 to m, dovetailing the execution of the locally-greedy algorithm on each growing $F(i)$. In more detail, the algorithm builds a branching tree T where the root is at level 0 and each node v at level i represents one possible founder setting at site i, denoted $f_v(i)$. The path from the root to v specifies a distinct founder setting up to i, denoted $F^v(i)$. Let w denote the predecessor of v in T; the path in T to w specifies a founder setting denoted $F^w(i-1)$. Suppose that the execution of the locally-greedy algorithm along the path to w has computed the subset of founders $SF_s^w(i-1)$ and the number of breakpoints $BF_s^w(i-1)$ (based on $F^w(i-1)$), for each input sequence s. Then, given $f_v(i)$, one step of the locally-greedy algorithm can easily compute the next set $SF_s^v(i)$ and the number $BF_s^v(i)$ for each input sequence s. Note that the algorithm at level i only needs information from level $i-1$, which allows significant space savings. The node at level m with smallest $\sum_s BF_s(m)$ identifies an optimal solution to the minimum mosaic problem. The correctness of this method follows from the correctness of the locally-greedy algorithm on any fully specified F, and the fact that all possible $F(i)$ are enumerated. However, without further speedups the method is only practical for very small data sizes.

The obvious speedup is to implement a branch-and-bound strategy, using a lower bound $L(i+1)$ on the number of breakpoints needed for the sites $i+1$ to m. If at node v, $\sum_s BF_s^v(i) + L(i+1)$ is greater or equal to the number of breakpoints needed in some known full configuration, then no expansion from node v is needed. We have implemented this strategy using the lower bound described earlier, but we have found the following speedup to be more effective.

If $\sum_s BF_s^v(i) - \sum_s BF_s^u(i) \geq n$, then no expansion from v is needed. To see this, note that if v were expanded, any configuration at a child of v (at level $i+1$) can be created from any one of the implicitly described configurations at u, using at most n breakpoints between sites i and $i+1$. Therefore, any path to level m from v requiring b breakpoints will require at most $b+n$ breakpoints from u. This idea can be greatly sharpened as follows. Suppose for some input sequence s, the set $SF_s^v(i) \subseteq SF_s^u(i)$, and consider a configuration c at a child of v at level $i+1$. If configuration c maps s to a founder in $SF_s^v(i)$, then configuration c can be created from at least one of the implicitly described configurations at u, with no breakpoints in s between sites i and $i+1$. If c maps s to a founder not in $SF_s^v(i)$ then there is one breakpoint used (for s) on that path out of v, and so one breakpoint can also be used on a path out of u to create the same mapping of s. Continuing with this reasoning, let n' be the number of sequences s where $SF_s^v(i) \subseteq SF_s^u(i)$. Then if $\sum_s BF_s^v(i) - \sum_s BF_s^u(i) \geq n - n'$, node u is as good or better than v, and v can be pruned. To fully implement this idea, we examine pairs of nodes at level i to find any node that is "beaten" by another node, and therefore can be pruned. While that is a relatively expensive step, without any pruning the size of T grows exponentially with i, and so it is worthwhile for the algorithm to spend time finding significant pruning. We have seen empirically that this approach is very effective in efficiently solving the minimum mosaic problem for a small number of founders (in the range 3 to 5) and a number of input sequences and sites which is generally larger than many biological applications today.

There is another speedup that can be introduced if the number of founders becomes large. As described above, tree T cannot contain two founder settings up to i, $F^u(i)$ and $F^v(i)$ at nodes u and v, where the *ordered* rows of $F^u(i)$ and $F^v(i)$ are identical. However, the rows of $F^u(i)$ can be the same as the rows of $F^v(i)$, but in a permuted order. We call such a pair of nodes "isomorphic", and in any isomorphic pair only one of the two nodes needs to be expanded; the other node and the subtree extending from it can be deleted. Redundant computation caused by isomorphism only becomes a significant problem when the number of founders is large, but isomorphism can be easily handled or avoided. One simple rule to handle it is to only expand a node u if the rows of $F^u(i)$ are in lexicographic sorted order (say lexicographically non-decreasing); any node whose rows are not in lexicographic sorted order can be pruned. That leads to the idea of only generating founder settings whose rows are in lexicographic order, avoiding isomorphic pairs entirely. Suppose inductively that at level $i-1$, every generated founder setting up to $i-1$, $F^w(i-1)$, has rows that are in lexicographic sorted order. Of course, all identical rows in $F^w(i-1)$ will be contiguous. Then for any set of k identical rows in $F^w(i-1)$, if a potential founder setting $f_v(i)$ for a child v of w, would set k' of those k rows to 0, and $k-k'$ to 1, at site i, place the zeros in the first k' of those k rows. In that way, the rows of $F^v(i)$ will be in lexicographic sorted order, and no isomorphism will be created at level i.

4 Simulation Results and Open Problems

We implemented the general method (without speedups to avoid isomorphism) in a C++ program, and ran our program on biological datasets on a standard 2.0GHz Pentium PC.

The first data is Kreitman's classic data [4]. After appropriate data reduction [12], there are 9 haplotypes and 16 sites left. The simulation results, including lower bound and exact minimum number of breakpoints, are shown in Table 1 for different K_f. As expected, as the number of founders increases, the minimum number of breakpoints decreases. Note that the *composite* lower bound using the composite method can be higher than the simple lower bound B_m on the entire data. For example, when $K_f = 3$, $B_m = 9 - 3 = 6$, while the composite bound reported in Table 1 is equal to 10.

Table 1. Solutions for minimum mosaic problem for Kreitman's data. The data is reduced from the original 11 haplotypes and 43 binary sites. After data reduction, there are 9 rows and 16 sites left. Both lower bound (LB) and exact minimum number of breakpoints (EMB) are shown. Running time (Time) is also displayed.

	$K_f = 2$	$K_f = 3$	$K_f = 4$	$K_f = 5$	$K_f = 6$
LB	27	10	7	4	3
EMB	37	15	8	6	4
Time (s)	< 1	< 1	1	12	1245

For a larger example, we use the full Jackson region of the LPL data [8] (with 40 haplotypes and 49 sites). After data reduction, it contains 37 haplotypes and 43 sites. When $K_f = 3$, it takes 27 seconds to find 241 as the minimum number of breakpoints. When $K_f = 4$, it takes a little over an hour to find 181 as the minimum number of breakpoints. The program, taking about 50 minutes, was also used to show that 53 breakpoints are the minimum needed in a dataset with 20 haplotypes and 36 sites and $K_f = 5$, posted at http://www.unc.edu/courses/2007spring/comp/790/087/. A heuristic greedy algorithm discussed there previously found a solution with 54 breakpoints.

Our program, called *RecBlock*, is available for download at the web page: http://wwwcsif.cs.ucdavis.edu/˜wuyu/.

Open problems. A major open problem is to determine the complexity of the minimum mosaic problem. Another interesting problem is to develop a (possibly parametrized) polynomial time algorithm when K_f is a small constant larger than two.

Acknowledgments. Work supported by grants CCF-0515278 and IIS-0513910 from National Science Foundation. We thank Leonard McMillan at UNC for providing the dataset discussed above. We also thank Yun S. Song for bringing the linkage mapping problem to our attention.

References

1. El-Mabrouk, N.: Deriving haplotypes through recombination and gene conversion, J. of Bioinformatics and Computational Biology 2, 241–256 (2004)
2. El-Mabrouk, N., Labuda, D.: Haplotype histories as pathways of recombinations, Bioinformatics 20, 1836–1841 (2004)
3. Kimmel, G., Shamir, R.: A block-free hidden markov model for genotypes and its application to disease association, J. of Comp. Bio. 12, 1243–1260 (2005)
4. Kreitman, M.: Nucleotide polymorphism at the alcohol dehydrogenase locus of drosophila melanogaster, Nature 304, 412–417 (1983)
5. McMillan, L., Moore, K.: `http://www.unc.edu/courses/2007spring/comp/790/087/?p=11`
6. Myers, S.: The detection of recombination events using DNA sequence data, PhD dissertation, Dept. of Statistics, University of Oxford, Oxford, England (2003)
7. Myers, S.R., Griffiths, R.C.: Bounds on the minimum number of recombination events in a sample history. Genetics 163, 375–394 (2003)
8. Nickerson, D., Taylor, S., Weiss, K., Clark, A., et al.: DNA Sequence Diversity in a 9.7-kb region of the human lipoprotein lipase gene. Nature Genetics 19, 233–240 (1998)
9. Rastas, P., Koivisto, M., Mannila, H., Ukkonen, E.: A Hidden Markov Technique for Haplotype Reconstruction. In: Proceedings of Workshop on Algorithm of Bioinformatics (WABI 2005) pp. 140–151 (2005)
10. Scheet, P., Stephens, M.: A fast and flexible statistical model for large-scale population genotype data: applications to inferring missing genotypes and haplotypic phase. Am. J. Human Genetics 78, 629–644 (2006)
11. Schwartz, R., Clark, A., Istrail, S.: Methods for Inferring Block-Wise Ancestral History from Haploid Sequences. In: Proceedings of Workshop on Algorithm of Bioinformatics (WABI'02), vol. 2452, pp. 44–59 (2002)
12. Song, Y.S., Wu, Y., Gusfield, D.: Efficient computation of close lower and upper bounds on the minimum number of needed recombinations in the evolution of biological sequences. Bioinformatics, vol. 421, pp. i413–i422. In: Proceedings of ISMB (2005)
13. Tan, Y., Fu, Y.: A Novel Method for Estimating Linkage Maps, Genetics 173, 2383–2390 (2006)
14. Tyson, G.W., Chapman, J., Hugenholtz, P., Allen, E., Ram, R., Richardson, P., Solovyev, V., Rubin, E., Rokhsar, D., Banfield, J.: Community structure and metabolism through reconstruction of microbial genomes from the environment, Nature 428, 37–43 (2004)
15. Ukkonen, E.: Finding Founder Sequences from a Set of Recombinants. In: Proceedings of Workshop on Algorithm of Bioinformatics (WABI'02), vol. 2452, pp. 277–286 (2002)

Computing Exact p-Value for Structured Motif

Jing Zhang[1], Xi Chen[1], and Ming Li[2]

[1] Computer Science, Tsinghua University, Beijing, 100084, China
{mitjj00,xichen00}@mails.thu.edu.cn
[2] School of Computer Science, University of Waterloo, Waterloo,
Ontario N2L 3G1, Canada
mli@uwaterloo.ca

Abstract. Extracting motifs from a set of DNA sequences is important in computational biology. Occurrence probability is a common used statistics to evaluate the statistical significance of a motif. A main problem is how to calculate the occurrence probability of the motif on the random model of DNA sequence efficiently and accurately. In this paper, we are interested in a particular motif model which is useful in transcription process. This motif, which is called structured motif, is composed two motif words on single nucleotide alphabet and with fixed spacers between them. We present an efficient algorithm to calculate the exact occurrence probability of a structured motif on a given sequence. It is the first non-trivial algorithm to calculate the exact p-value for such kind of motifs.

Keywords: Pattern and motif discovery, exact p-value, structured motif, dynamic programming.

1 Introduction

Transcription factors play a prominent role in gene regulation; identifying and characterizing their binding sites is central to annotating genomic regulatory regions and understanding gene-regulatory networks. More and more research works focus on this field. An important aspect of this is determining the statistical significance of the occurrences of transcription factor binding site (TFBS), also called motifs, in a DNA sequence. Statistical measures used for evaluating overabundance of patterns in sequences have been studied extensively, among which the *z-score* and *p-value* are most popular. P-value is the occurrence probability of a motif on the random model of DNA sequences for at least observed times.

A widely used random model of DNA sequences is Markov chain model which considers DNA sequences as a sequence of variable indexed by a finite state Markov chain. In our paper, we use 1-order markov chain to model DNA sequences. There are many different ways to model a motif and the p-value is different under different models. The basic motif model is a word on single nucleotide alphabet $\Sigma = \{A, C, G, T\}$ or IUPAC alphabet which allows more than one nucleotide to occupy a single position. To represent more complex motifs, general models such like PWM(position weight matrix) and PSSM(position specific scoring matrix) are introduced. In this representation, the nucleotide on each position of the motif is chosen from single nucleotide alphabet according to certain probabilities. A very useful model called "structured motif" is introduced by Marsan and Sagot [1,2]. The structured motif may be composed of two or more ordered of words, called "boxes". Each box is separated from the next one by a certain number of spacers("N"). The interval may be different for two pairs of consecutive boxes. We can find many motifs having the structured property in biological data such like *GAL*4 which can be written as "CGGNNNNNNNNNNNCCG".

The non-overlapped two boxes problem has been studied by van Helden *et al.* [4] in which the structured motifs are not allowed to overlap each other when they appear. Robin S. *et al.* gave an algorithm to calculate the approximation probability of occurrence of a motif composed of two of more boxes separated by variable number of spacers in [3]. However, there is no efficient algorithm to calculate the exact probability for motif composed of exact two boxes separated by a fixed number of spacers considering motif overlap. In this paper, we propose the first non-trivial and efficient algorithm to solve such a problem. The time complexity of the algorithm is polynomial when the ratio of the number of spacers to the length of the second box is a constant.

We will give the notations and the formal description of the problem in section 2. The details of the algorithm are introduced in section 3 and we will give the conclusion and future work in section 4. We will show the details of the time complexity analysis in Appendix A. We also implement the algorithms and do some computational experiments on yeast data. The results are shown in Appendix B. The software and materials are available on our website http://bio.dlg.cn.

2 Preliminary

2.1 Basic Notations

For any string S, we use $S[i]$ to denote the ith position of S, and $S[i,j]$ to denote the substring of S from $S[i]$ to $S[j]$ inclusive, i.e. $S[i]S[i+1]\ldots S[j]$. Let $\Sigma = \{A, C, G, T\}$ be the alphabet of nucleotides. The random model of DNA sequence is Markov chain with length n on Σ, i.e. we assume that it has been generated by a Markov chain. We denote by

$$m = m_1(\#N = t)m_2 \tag{1}$$

a structured motif composed of two words separated by t spacers which means that there can be any nucleotide between them. The lengths of m, m_1 and m_2 are l, l_1 and l_2 where $l = l_1 + l_2 + t$. A motif is considered to hit a string if the string contains the motif as a substring. That is, there $\exists i$, such that $m_1 = R[i, i+l_1-1]$ and $m_2 = [i+l_1+t, i+l-1]$. A_i represents an event that m hits at position i. The number of hits is the number of such different i, regardless of overlaps.

2.2 Problem Description

From a theoretical point of view, regulatory regions can be divided into two parts: the binding sites which play an important role in regulating gene expression, and the background which is not bound by transcription factors of interest. The key point for discriminating the signals from the background is to estimate if the motif is over-represented under the null hypothesis. To evaluate the statistical significance of the motif, we have to calculate the probability of a structured motif m hits a Markov region at least k times, where k is the appearance times of the motif on the given sequence. We give the formal description as following:

Input: A structured motif $m = m_1(\#N = t)m_2$ where m_1 and m_2 are motif words over alphabet $\Sigma = \{A, C, G, T\}$, an integer $k > 0$ and a Markov Region R with length n
Output: $\Pr(m$ hits region R at least k times$)$

3 Algorithm

To sketch the idea of the algorithm and make the description clean, we first give the algorithm for an independent and identically distributed (i.i.d) model and we will extend it to Markov model later. The main technic used here is dynamic programming. But if we use dynamic programming directly, the number of terms to calculate will expand too larger. To avoid such a problem, first we do transformations to the target probability and decompose the probability into terms which can be calculated using dynamic programming and the number of terms is constrained.

We will present the details of our algorithm for case $k = 1$, i.e the motif hits the region at least one times and extend it to general k later. When a motif m hits a region R, it can appear on any position of R, so the target probability equals to $\Pr(A_1 \bigcup A_2 \bigcup \ldots \bigcup A_n)$ where A_i represents an event that m hits at position i. We can use the principle of inclusion and exclusion to decompose the target probability. Each term in the equation is the sum of probability that the motif hits at some positions simultaneously, for example, motif m hits region at $b, b + 2$ in Figure 2 at the same time. We further decompose the term by the first hit position and second hit position. After decomposition, the term is the probability that the motif hits a shorter region with a well-defined prefix for fewer times and we can use dynamic programming to calculate it.

3.1 Decomposition and Transformation

First, We use the principle of inclusion and exclusion to decompose the target probability and classify the hit events by the first hit position.

$$\begin{aligned}\Pr(m \text{ hits } R) &= \Pr(A_1 \bigcup A_2 \bigcup \ldots \bigcup A_n) \\ &= \sum_{a=1}^{n} (-1)^{a-1} \sum_{1 \le i_1 < i_2 < \ldots < i_a \le n} \Pr(\bigcap_{v=1}^{a} A_{i_v}) \qquad (2)\end{aligned}$$

$$= \sum_{a=1}^{n}(-1)^{a-1}\sum_{b=1}^{n}\sum_{i_1=b<i_2<\ldots<i_a\leq n}\Pr(\bigcap_{v=1}^{a}A_{i_v}) \tag{3}$$

The equation (2) is from the principle of inclusion and exclusion . We classify the events according to different value of i_1 to get the equation (3). Let

$$P(a,b) = \sum_{i_1=b<i_2<\ldots<i_a\leq n}\Pr(\bigcap_{v=1}^{a}A_{i_v}) \tag{4}$$

The key problem is how to calculate $P(a,b)$ for all a and b from 1 to n.

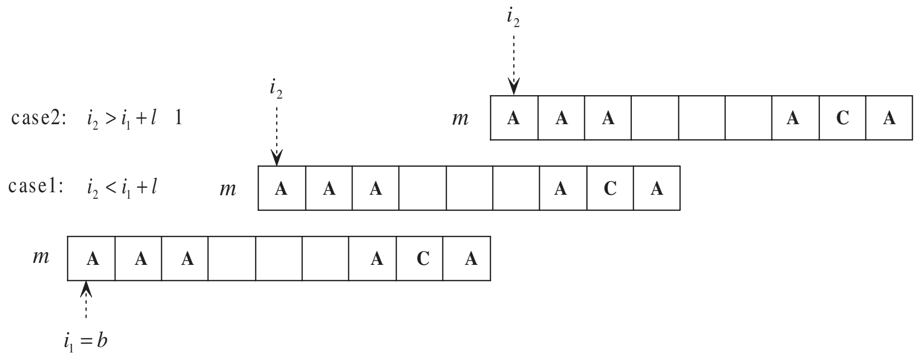


Fig. 1. Two cases of i_2

We do further decomposition to $P(a,b)$. When i_1 decided, there are two kinds of relation between i_1 and i_2, one is that the motif starting from i_1 does not overlap that starting from i_2 and the other is overlap case. We classify the events by different value of i_2 in equation (5):

$$\begin{aligned} P(a,b) &= \Pr(A_b) \times \sum_{b_1=b+l}^{n} P(a-1,b_1) \\ &+ \sum_{b_2=b+1}^{b+l-1}\sum_{i_1=b<i_2=b_2<\ldots<i_a<n}\Pr(\bigcap_{v=1}^{a}A_{i_v}) \end{aligned} \tag{5}$$

Let

$$P_1(a,b) = \sum_{b_2=b+1}^{b+l-1}\sum_{i_1=b<i_2=b_2<\ldots<i_a<n}\Pr(\bigcap_{v=1}^{a}A_{i_v}) \tag{6}$$

$$P_2(a,b) = \Pr(A_b) \times \sum_{b_1=b+l}^{n} P(a-1,b_1)$$

We can see that P_1 corresponds to the overlap case and P_2 the non-overlap case. P_2 can be calculated by $P(a-1, b_1)$ which is already known in dynamic programming, but we can not calculate P_1 using dynamic programming directly and we keep on doing transformations to P_1.

In fact, the events A_{i_1} and A_{i_2} can not hold simultaneously for many choices of i_2. For two given motif m' and m'', we define the **compatible** position set.

Definition 1. *For given motif m' and m'' with length l on alphabet $\{A, C, G, T, N\}$. Let $Q(m', m'')$ denote the* ***compatible*** *position set of m' and m''. For any i that $0 < i \le l$, $i \in Q(m', m'') \iff \forall 1 \le j < i, m'[j] = m''[l-i+j]$ or $m'[j] = N$ or $m''[l-i+j] = N$.*

In another word, $Q(m', m'')$ are all the possible positions for m'' to appear in when there is already a motif m' appears. We constrain the choices of i_2 in equation (6) and rewrite it.

$$\begin{aligned} P_1(a,b) &= \sum_{d \in Q(m,m)} \sum_{i_1=b<i_2=b+d<\ldots<i_a\le n} \Pr(\bigcap_{v=1}^{a} A_{i_v}) \\ &= \sum_{d\in Q(m,m)} \Pr(m[1,d] = R[b, b+d-1]) \times \sum_{b+d<i_3<\ldots<i_a\le n} \\ &\quad \Pr(s_d \text{ is the prefix of } R[b+d,n], \bigcap_{v=3}^{a} A_{i_v}) \end{aligned} \tag{7}$$

in which $\Pr(m[1,d] = R[b, b+d-1])$ can be calculated using the parameters of i.i.d model directly. We show the meaning of s_d in Figure 2 for case $d = 9$. The basic idea of equation (7) is to divide the region $R[b, n]$ into two parts at i_2. A suffix of the motif m which starts from $i_1 = b$ locates in region $R[b+d, n]$. The combination of the suffix and the motif m which starts from $i_2 = b + d$ forms the prefix s_d.

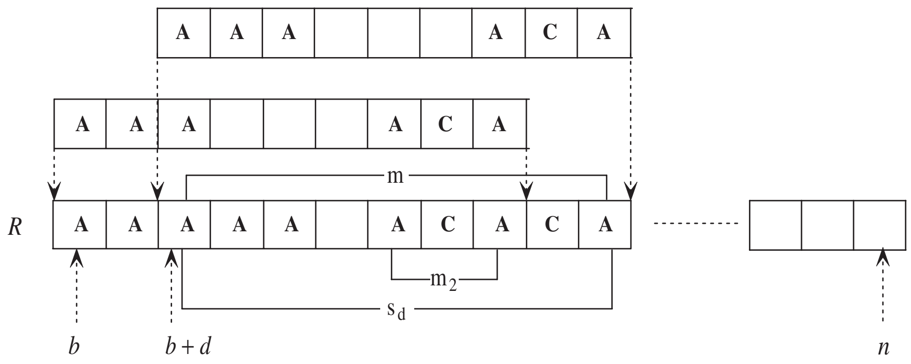


Fig. 2. Cut the region from $b + d$ and form the prefix s_d

Till now, we keep on doing transformations to the target probability and try to find some variants which are easy to calculate using dynamic programming and the number of variants does not expand too fast during the recursion. We will define another probability in subsection 3.2 and show that $P(a, b)$ and the well-defined probability can be calculated by dynamic programming together.

3.2 Dynamic Programming

The probability $I(x, y, z)$ is defined to be:

$$I(x, y, z) = \sum_{y \le i_1 < i_2 < \ldots < i_x \le n} \Pr(z \text{ is a prefix of } R[y, n], \bigcap_{v=1}^{x} A_{i_v}) \tag{8}$$

in which $1 \le x, y \le n$ and $s \in SP(m)$. $SP(m)$ is the structured prefix set of m which is defined as following:

Definition 2. *A string s with length l belongs to the* ***structured prefix set*** *of a given motif $m = m_1(\#N = t)m_2$ when it satisfies the three constraints:*
Constraint 1: $s[1, l_1] = m_1$ and $s[l - l_2 + 1, l] = m_2$
Constraint 2: $s[l - d - l_2 + 1, l - d] = m_2$, where $d \in Q(m, m)$.
Constraint 3: Other positions are covered by spacers or several overlapped m_2.

We can see that when the other positions are only covered by spacers, it is exactly the prefix s_d. Assume the number of m_2 used to do covering is r, we illustrate all the possible structured prefixes when $m = AAA(\#N = 5)ACA$ and $d = 9$ in Figure 3. We can regard it as using several m_2 to cover a string with length t and filling the rests with spacers. It is possible that several m_2 overlap each other and overlap $s[1, l_1]$ or $s[l - l_2 + 1, l]$. The constraint 3 is meant to make the prefix set complete in dynamic programming.

Then we prove that $P(a, b)$ and $I(x, y, z)$ can be calculated by dynamic programming. Since the complexity is related to the size of $SP(m)$, we just use $|SP(m)|$ to represent the size of $SP(m)$ in the time complexity analysis and the details of the estimation to $|SP(m)|$ are given in Appendix A.

Lemma 1. *For $1 \le x, y \le n$ and $z \in SP(m)$, $1 \le a, b \le n$, all $I(x, y, z)$ and $P(a, b)$ can be calculated using Dynamic Programming. If the size of $SP(m)$ is $|SP(m)|$, the total time complexity is $O(n^3|SP(m)|)$.*

Proof. It is easy to see that $P_1(a, b) = \sum_{d \in Q(m,m)} \Pr(m[1, d] = R[b, b + d - 1]) \times I(a - 2, b + d, z_d)$, so $P(a, b)$ can be calculated by $P(a', b')$ and $I(x, y, z)$ already known in dynamic programming. We calculate $I(x, y, z)$ for y from n to 1, x from 1 to $n - y$ and z in the structured prefix set of m in arbitrary order. When $x = 0$, $I(x, y, z) = \Pr(z \text{ is a prefix of } R[y, n])$ is easy to calculate. Assume we have got the value of $I(x', y', z)$ for all $y' > y$, $x' < x$, any $z \in SP(m)$. We do decomposition to $I(x, y, z)$ according to the first hit position:

$$I(x,y,z) = \sum_{b=y}^{n} \sum_{i_1=b<i_2<\ldots<i_x<n} \Pr(z \text{ is a prefix of } R[y,n], \bigcap_{v=1}^{x} A_{i_v}) \tag{9}$$

$$= \sum_{b_1=y+l}^{n} (\Pr(z \text{ is a prefix of } R[y,n]) \times P(x,b_1)) \tag{10}$$

$$+ \sum_{b_2-y\in Q(z,m)} (\Pr(z[1,b_2-y+1] = R[y,b_2]) \times I(x-1,b_2,z')$$

Since z' is a string covered by the prefix m and $z[b_2-y,l]$, it is still in $SP(m)$. The number of different $I(x,y,z)$ is $n^2|SP(m)|$ and the time to calculate each $I(x,y,z)$ is $O(n)$, so the total complexity is $O(n^3|SP(m)|)$.

We conclude the recursion formula for $I(x,y,z)$ and $P(a,b)$ as following:

$$P(a,b) = \Pr(A_b) \times \sum_{b_1=b+l}^{n} P(a-1,b_1)$$

$$+ \sum_{d\in Q(m,m)} \Pr(m[1,d] = R[b,b+d-1]) \times I(a-2,b+d,z_d)$$

$$I(x,y,z) = \sum_{b_1=y+l}^{n} (\Pr(z \text{ is a prefix of } R[y,n]) \times P(x,b_1))$$

$$+ \sum_{b_2-y\in Q(z,m)} (\Pr(z[1,b_2-y+1] = R[y,b_2]) \times I(x-1,b_2,z')$$

We can see that the DP algorithms to calculate $P(a,b)$ and $I(x,y,z)$ get involved with each other. □

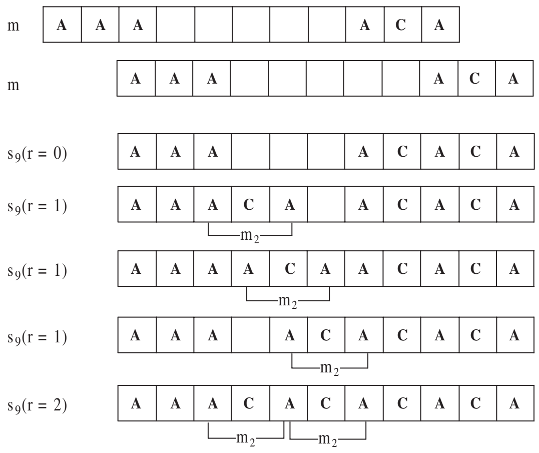


Fig. 3. all the prefixes in the structured prefix set when $m = AAA(\#N = 5)ACA$ and $d = 9$. r is the number of m_2 to do covering in constraint 3

Given all $P(a,b)$ for $1 \leq a, b \leq n$, we can calculate the target probability by

$$\Pr(m \text{ hits region } R \text{ at least } k \text{ times}) = \sum_{a=k}^{n} (-1)^{(a-k)\%2} \sum_{b=1}^{n} P(a,b) \qquad (11)$$

In Appendix A, we prove that the size of the structured prefix set of a structured motif $m = m_1(\#N = t)m_2$ is $O((2l_2+2)^{2t/l_2+2})$. When t/l_2 is a constant, it is $O(l_2^c)$ for some constant c. We have the following theorem:

Theorem 1. *Given a structured motif $m = m_1(\#N = t)m_2$ on alphabet $\Sigma = \{A, C, G, T\}$ and an i.i.d region R of length n, we can calculate* $\Pr(m$ *hits region R at least k times) in $O(n^3 \times s_{max} + t_{max} \times r_{max})$ where $s_{max} = (2l_2+2)^{2t/l_2+2}$ and $r_{max} = 2 \times \frac{l_2+t}{l_2+1}$. When t/l_2 is some constant c, the time complexity is $O(n^3 \times l_2^c)$.*

3.3 Sketch of Algorithm on Markov Model

The algorithm on markov model is quite similar to that on i.i.d model. The key idea for the extension is to maintain the last char before the region. We can rewrite the Subsection 3.1 and 3.2 using this idea and the frame of the algorithm and time complexity analysis remains the same.

We take equation (4)∼(7) in decomposition and transformation as an example do show the extension. We rewrite equation (4) as

$$P_M(a,b,c) = \sum_{c \in \Sigma} \sum_{i_1 = b < i_2 < \ldots < i_a \leq n} \Pr(\bigcap_{v=1}^{a} A_{i_v} | R[b-1] = c)$$

which is the sum over conditional probabilities with different char before region $R[b,n]$. We can also classify the events by different value of i_2 similar to equation (5) as following:

$$P_M(a,b,c) = \sum_{c \in \Sigma} (\Pr(A_b | R[b-1] = c) \sum_{d \in \Sigma} \sum_{b_1 = b+l}^{n} (P(a-1, b_1, d) \times \Pr(R[b_1 - 1]$$
$$= d | A_b)) + \sum_{b_2 = b+1}^{b+l-1} \sum_{i_1 = b < i_2 = b_2 < \ldots < i_a < n} \Pr(\bigcap_{v=1}^{a} A_{i_v} | R[b-1] = c))$$

The conditional probability in the non-overlap part P_{M_2} can be calculated using the transition probabilities of markov model. For the overlap part P_{M_1}, we rewrite it like equation (7):

$$P_{M_1} = \sum_{c \in \Sigma} (\sum_{d \in Q(m,m)} \Pr(m[1,d] = R[b, b+d-1] | R[b-1] = c) \times \sum_{b+d < i_3 < \ldots < i_a \leq n}$$
$$\Pr(s_d \text{ is the prefix of } R[b+d, n], \bigcap_{v=3}^{a} A_{i_v} | R[b+d-1] = m[d]))$$

The extension in Dynamic Programming is quite similar to that of decomposition and transformation part.

4 Conclusion and Future Work

In this paper, we present a non-trivial and efficient algorithm to calculate the probability of the occurrence of a structured motif $m = m_1(\#N = t)m_2$ where m_1 and m_2 are motif words on basic alphabet $\Sigma = \{A, C, G, T\}$ with t spacers. We do transformations to the target probability and define two variants which can be calculated by DP algorithm. The time complexity of the algorithm is $O(n^3 \times l_2^c)$ for some constant c when t/l_2 is a constant where l_1 and l_2 are the length of m_1 and m_2.

The algorithm can be used to evaluate the statistical significance of motif candidates with structured property. This kind of motifs are often with larger length in real biological data and it is harder to calculate the occurrence probability. Our algorithm is the first non-trivial and efficient algorithm to calculate the exact p-value of such kind of motifs.

One problem is that the algorithm is still an exponential time algorithm in worst case. Finding a polynomial time algorithm or proving that it is NP-hard are two main directions in the future work.

Acknowledgement

ZJ' and CX's work is supported by the National Natural Science Foundation of China Grant 60553001 and the National Basic Research Program of China Grant 2007CB807900, 2007CB807901. ML's work was partially supported by the Chang Jiang Scholarship Program, NSERC, and Canada Research Chair program.

References

1. 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. Biol. 7, 345–362
2. Marsan, L., Sagot, M.F: Extracting structured motifs using a suffix tree-algorithm and application to promoter consensus identification. In: RECOMB'00 Proceedings of Fourth Annual International Conference on Computational Molecular Biology, pp. 210–219. ACM Press, New York (2000)
3. Robin, S., Daudin, J.-J., Richard, H., Sagot, M.-F., Schbath, S.: Occurrence probability of structured motifs in random sequences. J. Comp. Biol. 9, 761–773 (2002)
4. Van Helden, J., Rios, A.F., Collado-Vides, J.: Discovering and Regulatory elements in non-coding sequences by analysis of spaced dyads. Nucl. Acids Res. 28, 1808–1818
5. Zhu, J., Zhang, M.Q.: SCPD: A promoter database of yearst saccharomyces cerevisiae. Bioinformatics 15, 607–611 (1999)

A Estimation of the Size of Structured Prefix Set

The following lemma is about the size of the structured prefix set of m.

Lemma 2. *The size of the structured prefix set of a structured motif* $m = m_1(\#N = t)m_2$ *is* $O((2l_2 + 2)^{2t/l_2+2})$. *When* t/l_2 *is a constant, it is* $O(l_2^c)$ *for some constant* c.

Proof. According to definition 2, for any string s in $SP(m)$, the contents of the first l_1 and the last l_2 length of s are fixed. Although there must be a m appeared at some position in constraint 2, this constraint is just for the convenience to define s_d from $SP(m)$ and it has been contained in constraint 3. We can omit it when estimating the size of $SP(m)$. The number of different $s[l_1+1, l_1+t]$ is exact the size of $SP(m)$, so we prove that the number of different s^* is at most $O((2l_2+2)^{2t/l_2+2})$ where $s^* = s[l_1+1, l_1+t]$.

First, we give the upper bound of the needed number of m_2 to cover the whole s^*. Assume we use $m_2^{(1)}, m_2^{(2)}, \ldots, m_2^{(t)}$ to cover s^*. $m_2^{(j_1)}$ appears before $m_2^{(j_2)}$ if $j_1 < j_2$. Make sure that every $m_2^{(j)}$ is needed. Here, the needed means that there exists a substring of s^* which is only covered by $m_2^{(j)}$. We have the following to facts:

Fact 1. *$\forall 1 \le j \le r$, $m_2^{(j)}$ covers at a substring of s^* with length at least* 1.

Fact 2. *$m_2^{(j)}$ can not overlap $m_2^{(j+2)}$.*

Fact 2 is right because if they overlap each other, $m_2^{(j+1)}$ is not needed. Combine the two facts, we have an upper bound of r,

$$r \le 2 \times \frac{l_2+t}{l_2+1} = r_{max}$$

The number of different s^* is bounded by the number of choosing r positions from l_2+t possible start positions for m_2 in which r is ranging from 0 to r_{max}.

$$\begin{aligned}\text{number of different } s^* &\le \sum_{r=0}^{r_{max}} \binom{l_2+t}{r} \le \left(\frac{l_2+t}{r_{max}}\right)^{r_{max}} \\ &= (2l_2+2)^{2\times\frac{l_2+t}{l_2+1}} \le (2l_2+2)^{2t/l_2+2}\end{aligned}$$

Let s_{max} denote $(2l_2+2)^{2v/l_2+2}$. We can construct the structured prefix set just by trying all the possible choices of r from 0 to r_{max} increasingly and the total construction time complexity is bounded by $O(r_{max} \times s_{max})$.

B Computational Experiment

We implement the algorithm in C++ and do computational experiment on yeast data. GAL4 is a family of transcription activator of genes. It has a typical Zn-binding region and is reported to bind to a 17 base-pair palindromic site. The consensus binding motif is "CGGNNNNNNNNNNNCCG" which can be regarded as a structured motif of two boxes with length 3 and 11 spacers between them.

In *Saccharomyces crevisiae* Promoter Database (SCPD) [5], transcription factor *GAL4* is reported to bind to 7 Genes: GAL1, GAL2, GAL4, GAL7, GAL10, GAL80 and GCY1. We extract upstream sequence, of length 1000 bp, for these 7 genes and estimate the occurrence probabilities of $\{A, C, G, T\}$ on each sequence.

We calculate the exact p-value of motif $GAL4$ to appear on each sequence for observed times. The p-value and consumed time[1] are shown in Table 1. If we use $\alpha = 0.05$ in hypothesis test to decide whether motif GAL4 is over-represented in the promoter region, we can see that all the p-values are significant below α except GAL80. The result shows the power of the algorithm to discriminate the transcription signals from the background. The average time is $12.6s$ which is acceptable in application. We also enumerate all the structured motifs with

Table 1. The p-value of motif $GAL4$ on corresponding Genes

Gene Name	P-value of motif GAL4	Time
GAL1	9.61361E-06	12813ms
GAL2	2.16228E-09	12734ms
GAL4	0.006578594	12609ms
GAL7	0.034349462	12532ms
GAL10	9.61361E-06	12641ms
GAL80	0.092351709	12282ms
GCY1	0.045182862	12640ms

$l_1 = l_2 = 3$ and $t = 11$. The number of such motifs is $4^6 = 4096$. We calculate the exact p-value of all such motifs on Gene GAL4 and rank them according to p-value increasingly. The top 20 p-values are shown in Table 2. We can see that the rank of "CGG(#N=11)CCG" is the 11-th of all 4096 structured motifs of similar structure, that is in the top 0.2%.

Table 2. The top 20 p-value of structured motif on Genes $GAL4$

Structured Moitf	P-value	Structured Moitf	P-value
TTT(#N=11)TTT	0.00135	CGG(#N=11)CCG	0.00658
ACA(#N=11)AGG	0.00141	AAA(#N=11)CTT	0.00689
AGA(#N=11)CAG	0.00151	ATT(#N=11)ACA	0.00704
GTG(#N=11)AGA	0.00163	GGA(#N=11)GGC	0.00858
AGC(#N=11)TCA	0.00182	CGG(#N=11)AGG	0.00858
GAC(#N=11)CTA	0.00183	TTA(#N=11)TTC	0.00963
TTT(#N=11)CGC	0.00273	GGT(#N=11)CGG	0.01004
ATT(#N=11)GTG	0.00393	GGG(#N=11)TCC	0.01020
TTT(#N=11)GAG	0.00430	GTC(#N=11)CGG	0.01037
CAC(#N=11)TTT	0.00516	TTT(#N=11)CTT	0.01238

[1] The compiler is Microsoft VC 6.0 and The test PC is equipped with a Pentium 4 running at 2.8GHz and 512MB of RAM.

Improved Sketching of Hamming Distance with Error Correcting

Ely Porat[1] and Ohad Lipsky[2]

[1] Bar-Ilan University, Dept. of Computer Science, 52900 Ramat-Gan, Israel and Google Inc.
poratelу@cs.biu.ac.il

[2] Bar-Ilan University, Dept. of Computer Science, 52900 Ramat-Gan, Israel
ohadlipsky@yahoo.com

Abstract. We address the problem of sketching the hamming distance of data streams. We develop *Fixable Sketches* which compare data streams or files and restore the differences between them. Our contribution: For two streams with hamming distance bounded by k we show a sketch of size $O(k \log n)$ with $O(\log n)$ processing time per new element in the stream and how to restore all locations where the two streams differ in time linear in the sketch size. Probability of error is less than $1/n$.

1 Introduction

Massive data streams are fundamental part in many data processing applications. The increasing size of data sets raises the need for improved algorithms for processing these massive data sets. We discuss various problems and present improved algorithms for problems in the streaming model and communication complexity. We use novel techniques of sketching ("fingerprint") and encodings. See [12] for detailed motivation. Indyk [10] presented a wide range of very useful tools in data streams algorithms using stable distributions. In the streaming model the algorithm can make only one pass on the data and maintain a small "sketch" for further processing and queries. In [7] they present a sketch of size $O(\log(n|\Sigma|)\log(1/\delta)/\epsilon^2)$ that allows approximating the L^1-difference between two streams up to a factor of $1 \pm \epsilon$ and error probability δ. Another known problem in the streaming model is maintaining histograms of the stream, as discussed in [9, 8]. In [4] the problem of estimating the hamming norm for massive data streams is presented. They give a sketch of size $O(1/\epsilon^2 \cdot \log 1/\delta)$ that allows hamming distance approximation with a factor of $1 \pm \epsilon$ and error probability δ. A similar result of approximating the hamming distance is presented in [11]. Let $s(x)$ denote the sketch of stream x. There are three parameters that are relevant when comparing different sketching algorithms:

- The sketch size, i.e. the number of bits it requires.
- Time to process new element in the stream.
- Time to compute target function $f(x, y)$ given $s(x)$ and $s(y)$.

In [6] a sketch of size $O(k^4 \log^2 n)$ that allow computation of the hamming distance between binary streams is presented, with error probability exponentially small in k, where k is a bound on the hamming distance between the streams. We can combine this with a constant size sketch that identify cases where the hamming distance is greater than $2k$ in order to discard these cases. Bar-Yossef, Jayram, Kumar and Sivakumar et al. [2] presented a sketch with reduced size of $O(k \log^2 k)$ that allows computation of the hamming distance between binary streams with error probability polynomially small in k, time of processing new element is $O(\log k)$ and query time linear in the sketch size. This sketch can be changed to be of size $O(k^2 \log k)$ in order to have exponentially small in k error probability. We solve a more complicated problem; we design a sketch, that will enable us, not only to compute the hamming distance between two streams, but also to restore the locations the streams differed. Moreover, the specific symbols that appeared in each stream in these locations can be restored. Sketch size is $O(k \log n)$ ($O(k \log k)$ without the ability of error correcting), time to process new element is $O(\log n)$ ($O(\log k)$) and time to compute hamming distance linear in the sketch size. An easier version of the problem is studied in [5]. However, they do not work in the streaming model, and therefore can go over the input more than once. They have a small sketch indeed, but time to process a query is exponential.

The possibility to use such error correcting sketches is useful in a wide range of applications in network synchronization, pda synchronization, and more, for example see [13, 14].

2 Outline

In section 3 we summarize the sketch algorithm of [2] and we enhance it to fit general alphabet Σ rather than only binary one. In section 4 we show how to produce a sketch with reduced size of $O(2^{\log^* k} k \log k)$, processing time per element $O(2^{\log^* k} \log k)$ and query time linear in the sketch size. The final algorithm with the error correcting capabilities appears in section 5; understanding of this section does not require prior knowledge of sections 3 and 4.

3 Sketching Hamming Distance First Algorithm

Our first algorithm is a generalization of the algorithm of Bar-Yossef, Jayram, Kumar and Sivakumar in [2] to work for any alphabet Σ (rather than only binary one). The idea is to encode the stream into a binary stream and use the sketching technique on the encoded stream. Note that our sketch size is independent of the stream length.

The binary encoding:
We can assume w.l.o.g. that $\Sigma = \{1, 2, \ldots, |\Sigma|\}$. Define $c(\sigma) = c_1(\sigma)c_2(\sigma)\ldots c_{|\Sigma|}(\sigma)$ where $c_i(j) = 1$ if $i = j$ or 0 otherwise. For a string $\sigma_1 \ldots \sigma_n$

define $c(\sigma_1,\ldots,\sigma_n) = c(\sigma_1)\ldots c(\sigma_n)$. Clearly $HD(c(x),c(y)) = 2HD(x,y)$ and $HD(c(x),c(y)) < 2k$. Each new element x_i in the x stream is translated to $|\Sigma|-1$ zeros, and only one 1-bit in location $(i-1)|\Sigma| + x_i$. Updates to the sketch are done only when a 1-bit occur, therefore one element in the original stream cause update of one element in the binary encoded stream $c(x)$.

The sketch:

Pick at random $2k$-wise independent hash functions $h : [n|\Sigma|] \to [k/3\ln k]$ and $h' : [n|\Sigma|] \times [2\ln k] \to [c\ln^2 k]$ $(c = 72e)$.

Our sketch $s(x)$ is of size $\frac{2c}{3}k\ln^2 k$ bits, initialized by zeros.
For each new element σ_i (in the stream x) the following updates are done to the sketch:

```
Let bucket ← $h((i-1)|\Sigma| + \sigma_i)$.
For every $t = 1, 2, \ldots, 2\ln k$
    – sub_bucket ← $h'((i-1)|\Sigma| + \sigma_i, t)$
    – $d \leftarrow (bucket - 1)\cdot 2c\ln^3 k + (t-1)\cdot c\ln^2 k + sub\_bucket$.
    – $s(x)_d \leftarrow s(x)_d \oplus 1$.
```

Claim. Let x and y be two binary streams of the same length with $HD(x,y) < 2k$. Let $i_1, i_2, \ldots, i_{k'}$ be the locations where x differ from y (where $k' \le 2k$). $Pr(\forall bucket \in [1, k/3\ln k] |\{i_z | z \in [1,k'], h(i_z) = bucket\}| \le 12\ln k) \ge 1 - \frac{1}{k\ln k}$.

Proof. It is assumed that $k' < 2k$. Since our hash functions are $2k$-wise independent, For every $bucket \in [1, k/3\ln k]$

$$E(|\{i_z | z \in [1,k'], h(i_z) = bucket\}|) \le 6\ln k.$$

This imply that

$$Pr(\exists 1 \le bucket \le k/3\ln k |\{i_z | z \in [1,k'], h(i_z) = bucket\}| > 12\ln k)$$

$$\le \frac{k}{2\ln k} Pr(|\{i_z | z \in [1,k'], h(i_z) = bucket\}| > 12\ln k)$$

$$< \frac{k}{2\ln k}\cdot\frac{1}{k^2} = \frac{1}{k\ln k}$$

(Using Chernoff inequality).

Claim. Let x and y be two binary streams of the same length with $HD(x,y) < 2k$. Let $i_1, i_2, \ldots, i_{k'}$ be the locations where x differ from y (where $k' \le 2k$) and let h be a hash function s.t. $\forall b \in [1, k/3\ln k] |\{i_z | z \in [1,k'], h(i_z) = b\}| \le 12\ln k$.

$\forall b \in [1, k/3\ln k], t \in [1, 2\ln k] Pr(\exists i, i' \in \{i_1, \dots, i_{k'}\} h(i) = h(i') = b \wedge h'(i,t) = h'(i',t)) < e^{-1}$.

Proof. For each bucket b there are at most $12\ln k$ indexes s.t. $h(i_z) = b$. They are hashed into $c\ln^2 k$ buckets.

$$Pr(\exists i, i' h(i) = h(i') = b s.t. h'(i,t) = h'(i',t)) < \frac{\binom{12\ln k}{2}}{c\ln^2 k} < e^{-1}$$

Corollary 3.1. *$Pr(\forall b \in [1, k/3\ln k] \exists t \in [1, 2\ln k] \forall i_u, i_v \in \{i_1, \dots, i_{k'}\} h(i_u) = h(i_v) = b : h'(i_u,t) \neq h'(i_v,t)) > 1 - \frac{1}{k\ln k}$.*

Computing $HD(c(x), c(y))$**:**

```
1. For every bucket $b \in [1, k/3\ln k]$:
       Let $d \leftarrow (b-1)\cdot 2c\ln^3 k + (j-1)c\ln^2 k$.
       $H(b_j) \leftarrow \Sigma_{l=d+1}^{d+c\ln^2 k} neq(s(x)_l, s(y)_l)$.
2. $H(b) \leftarrow \max_{j\in[1,2\ln k]} H(b_j)$
3. $Dist \leftarrow \Sigma_b H(b)$
```

Claim. Let $i_1, i_2, \dots, i_{k'}$ be the locations where $c(x)$ differ from $c(y)$ ($k' < 2k$). If h and h' obey the following conditions:

– $\forall b \in [1, k/3\ln k] |\{i_z | z \in [1, k'], h(i_z) = b\}| \leq 12\ln k$

– $\forall b \in [1, k/3\ln k] \exists t \in [1, 2\ln k]$ s.t. $\forall u, v \in [1, k']$ if $h(i_u) = h(i_v) = b$ then $h'(i_u, t) \neq h'(i_v, t)$.

Then $HD(x,y) = \frac{Dist}{2}$.

Proof: Let $b \in [1, k/3\ln k]$ and let t be the value for whom $\forall i, i'$ s.t. $h(i) = h(i') = b$ $h'(i',t) \neq h'(i,t)$. It implies that at most one location that differ between $c(x)$ and $c(y)$ is mapped into each bit in the sketch segment between $(b-1)\cdot 2c\ln^3 k + (t-1)\cdot c\ln^2 k + 1$ and $(b-1)\cdot 2c\ln^3 k + (t)\cdot c\ln^2 k$, which in turn imply that $H(b_j) = |\{i | h(i) = b\}|$ and therefore also $H(b) = |\{i | h(i) = b\}|$. Since every location is mapped into exactly one bucket b We can conclude that $HD(c(x), c(y)) = Dist = \Sigma_b H(b)$. As mentioned before $HD(x,y) = \frac{1}{2} HD(c(x), c(y))$.

Theorem 3.2. *Given a stream x over alphabet Σ, there is an algorithm that maintain a sketch $s(x)$ that requires $O(k\log^2 k)$ bits of memory and $O(\log k)$ processing time per element and has the following property: Using sketches $s(x)$, $s(y)$ for two streams x and y where $HD(x,y) \leq k$ we can compute $HD(x,y)$ with probability of error less than $1/k$.*

Proof. Immediate, combining claims 3, 3 and corollary 3.1 together.

4 Sketching Hamming Distance - Second Algorithm

In the second algorithm we reduce the sketch size to be only $O(2^{\log^* k} k \log k)$ bits of memory.

In the first algorithm the stream partitioned into buckets, and each bucket into sub-buckets. Here, We recursively partition the stream into smaller buckets, till we have buckets with at most one difference (with high probability). For each new element we update the relevant buckets. The sketch will be only the lowest level bucket's parity. We use the same binary encoding as in the previous sketching.

We define $\alpha(k) = \frac{k}{4 \ln k (\ln^* k)^2}$, $\beta(k) = \frac{2 \ln k}{\ln[4 \ln k (\ln^* k)^2]}$ and $\gamma(k) = (1 + \frac{1}{\ln^* k}) 4 \ln k (\ln^* k)^2$. We will divide the stream into $\alpha(k)$ buckets, and for each bucket b we will run independent $\beta(k)$ times the sketch in recursion, with $k' = \gamma(k)$ as bound.

The sketch:
Pick at random $2k$-wise independent hash functions:
$h_1 : [n|\Sigma|] \to [\alpha(k)]$
$h_2 : [n|\Sigma|] \times [\beta(k)] \to [\alpha(\gamma(k))]$.
$h_3 : [n|\Sigma|] \times [\beta(\gamma(k))] \to [\alpha(\gamma(\gamma(k)))]$
$\vdots$
$h_i : [n|\Sigma|] \times [\beta(\gamma^{(i-2)}(k))] \to [\alpha(\gamma^{(i-1)}(k))]$
$\vdots$
When $\gamma^{(i)}(k)$ becomes small enough s.t. $\alpha(\gamma^{(i)}) < 4$ we can bound $\gamma(i)$ by some constant c', and use a simple constant size sketch with error probability of less than $\frac{1}{2c'}$. It is easily seen that it occurs when $i < \ln^* k$. Denote by i' the index it occurred.

For each new element σ_i (in the stream x) the following updates are done to the sketch:

```
– $depth \leftarrow 1$
– $l \leftarrow (i-1)|\Sigma| + \sigma_i$
– $b_1 \leftarrow h_1(l)$
– For every $t_1 = 1, 2, \ldots, \beta(k)$
– **Update procedure** $(l, t_1, \gamma(k), depth)$

Where Update$(l, t, k, depth)$ Defined by:
If $depth = i'$ then update the sketch segment relevant to $b_1, b_2, \ldots, b_{depth}$.
else:

  $depth \leftarrow depth + 1$
  $b_{depth} \leftarrow h_{depth}(l, t)$
  For every $t_{depth} = 1, 2, \ldots, \beta(k)$
  Update$(l, t_{depth}, \gamma(k), depth)$
```

Claim. The sketch size is of $O(2^{\ln^* k} k \log k)$.

Proof: Let $d(z) = 4 \ln z (\ln^* z)^2$.

$$Size(k) = \alpha(k) \cdot \beta(k) Size(\gamma(k)) =$$
$$= \alpha(k)\beta(k)\alpha(\gamma(k))\beta(\gamma(k))Size(\gamma(\gamma(k))) =$$
$$= \ldots = \Pi_{i=0}^{i'} \alpha(\gamma^{(i)}(k)) \cdot \Pi_{i=0}^{i'} \beta(\gamma^{(i)}(k))$$

Where

$$f^{(i)}(k) = \begin{cases} f(k) & \text{if } i = 1 \\ f(f^{(i-1)}(k)) & \text{if } i > 1 \end{cases}$$

Observe that $\gamma(z) = d(z) \cdot (1 + \frac{1}{\ln^* k})$.

$$\Pi_{i=0}^{i'} \alpha(\gamma^{(i)}(k)) = \frac{k}{d(k)} \cdot \frac{\gamma(k)}{d(\gamma(k))} \cdot \frac{\gamma(\gamma(k))}{d(\gamma(\gamma(k)))} \cdots \frac{\gamma^{(i')}(k)}{d(\gamma^{(i')(k)})} = \frac{k}{d(c')} \cdot (1 + \frac{1}{\ln^* k})^{i'} \in O(k)$$

(Using that $i < \ln^* k$).

$$\Pi_{i=0}^{i'} \beta(\gamma^{(i)}(k)) \in O(2^{\ln^* k} \ln k)$$

Claim. Processing time per new element is $O(2^{\ln^* k} \log k)$.

Proof: Each element appears in

$$\beta(k)\beta(\gamma(k))\beta(\gamma(\gamma(k))) \ldots \beta(\gamma^{(\log^* k)}(k)) = O(2^{\log^* k} \log k)$$

buckets.

Computing $HD(x, y)$: For each of the $\alpha(k)$ buckets we take the maximum over the $\beta(k)$ repetitions. This is done recursively.

Claim. With probability at least $1 - 1/k$ $HD(x, y)$ is computed correctly.

Proof: Denote X_i the event that the algorithm failed when $k = i$. Let assume $Pr(X_i) < \frac{1}{i}$ for $i < k$. $Pr(X_k) = Pr$[One of the $\alpha(k)$ sub problems of size $\gamma(k)$ failed in all the $\beta(k)$ times] $\leq \alpha(k) Pr(X_{\gamma(k)})^{\beta(k)} < \frac{1}{k}$

Theorem 4.1. *Given a stream x over alphabet Σ, there is an algorithm that maintain a sketch $s(x)$ that requires $O(2^{\log^* k} k \log k)$ bits of memory and $O(2^{\log^* k} \log k)$ processing time per element and has the following property: Using sketches $s(x)$, $s(y)$ for two streams x and y where $HD(x, y) \leq k$ we can compute $HD(x, y)$ with probability of error less than $1/k$.*

5 Error correction and Fixable Sketches - Third Algorithm

In this section we present a totally different technique. We first present the notion of *Fixable sketch.* Fixable sketch is a sketching algorithm s that has the following property: Given a sketch $s(x)$ for a stream $x = x_1 \cdots x_n$, error location $i \in [1, n]$, and symbols x_i and c, one can compute $s(x')$, where $x' = x_1, x_2, \cdots, x_{i-1}, c, x_{i+1}, \cdots, x_n$. Informally, fixable sketch is for the case where we kept moving on the stream, and we are interrupted by some correction from elements we have already passed, in this case we would like to be able to fix the sketch according to the error corrected. This property can be useful for example in a case that relatively few errors encountered in the stream cause the sketch to be worthless. In the previous sections we presented sketches (which are, indeed, fixable sketches) that compute the hamming distance only if it is bounded by some k. Now, assume that in a stream x a random segment of size $2k$ was received, and later corrected, we still cannot use the sketch $s(x)$ as is in order to compute the distance from another stream y (given only $s(y)$) because probably it exceeds k even if y is identical to the corrected stream x'. Since we can adapt the sketch to the corrected stream it is possible to answer queries as needed. An important parameter in fixable sketches is the time it cost to make an error correction. In all the sketches described in this paper it is the same time as processing new element in the stream.

In the following we present a sketching technique that given the sketches of two streams x and y, s.t. $HD(x, y) \leq k$ we can restore (w.h.p.) at least $HD(x, y) - \frac{k}{2}$ locations $i_1, i_2, \ldots$ where $x_{i_j} \neq y_{i_j}$ and restore the values x_{i_j} and y_{i_j}. Such sketching will serve as the main building block of our sketch.

5.1 Error Correction Fixable sketch

We present here a sketch $s = s^k$ with the following properties:

1. The size of the sketch is of $O(k(\log|x| + \log|\Sigma|))$.
2. Time to process new element (or error correction) is constant
3. Using sketches $s(x)$ and $s(y)$ for streams x and y respectively, we can find (with probability $> 1 - \frac{1}{|x|}$) at least $HD(x, y) - \frac{k}{2}$ locations where x differ from y and restore the symbols of both x and y in these locations.

Let $|x| = n$ and let $p \in O(n^3|\Sigma|)$ be prime. All the computations will be done in $\mathbb{F}_q$ unless mentioned otherwise.

The sketch

Our sketch consists $8k$ quadruplets of variables: $a_j, b_j, c_j, d_j \in \mathbb{F}_p$ ($j \in [1, 8k]$), initialized by zeros. Pick at random k-wise independent hash functions $h : [n] \times [8] \rightarrow [8k]$ and $h' : [n] \times [8k] \rightarrow \mathbb{F}_p$.

When we receive a new element in the stream x_i we apply the following to the sketch:

```
Let $Q_i = \{h(i,t) | t \in [1,8]\}$ (i.e. $Q_i$ is of size $\leq 8$)
For every $j \in Q_i$
    $r \leftarrow h'(i,j)$
    Add $r \cdot i \cdot x_i$ to $a_j$
    Add $r \cdot x_i$ to $b_j$
    Add $r^2 \cdot x_i$ to $c_j$
    Add $r \cdot x_i^2$ to $d_j$
```

Clearly the sketch size is of $O(k \log p) = O(k \log n)$.

Finding Mismatches between streams
The input is $s(x), s(y)$ of two streams x and y s.t. $HD(x,y) \leq k$.
The notation will be $a_j(x)$ and $a_j(y)$ referring to the variables in $s(x)$ and $s(y)$, respectively. The other variables will be denoted in similar way. We assume both h' and h are common for both, we can later drop this assumption by using k-wise independent pseudo random generator with seed of size $O(\log n)$. The idea of the sketch is that when looking at specific $j \in [1, 8k]$ we get, with constant probability, the case where only in one location i that x differ from y some value is added to a_j, b_j, c_j and d_j. If this is the case, we are able to restore i, x_i and y_i easily. We formalize it in the following claim.

Lemma 1. *Let $L = \{i_1, i_2, \ldots, i_{k'}\}$ be the locations where x differ from y (where $k' \leq k$). If $k' > \frac{k}{2}$ then for every $j \in [1, 8k] Pr(\exists i' \in L, t \in [1,8] h(i,t) = j \wedge \forall i \in L, i \neq i' \forall t \in [1,8] h(i,t) \neq j) > \frac{1}{2e}$.*

Proof. For every $i \in L$: Since $h(i,t)$ is chosen in random, the probability $h(i,t) = j$ for some given j is exactly $\frac{1}{8k}$. Now, since t runs from 1 to 8, it is $\frac{8}{8k} = \frac{1}{k}$. If we have k' elements, and we choose each one with probability $\frac{1}{k}$ then the probability we chose exactly one is $k' \cdot \frac{1}{k}(1 - \frac{1}{k})^{k'-1}$. Given that $k' < k$ and $k' > k/2$ this probability is at least $\frac{1}{2e}$.

This leads us to the following procedure:

```
Initialize an empty set $Mismatches$.
For each $j \in [1, 8k]$:
    $r' \leftarrow \frac{c_j(x) - c_j(y)}{b_j(x) - b_j(y)}$.
    $i' \leftarrow \frac{a_j(x) - a_j(y)}{b_j(x) - b_j(y)}$.
    If $h'(i', j) = r'$ add $(i', x'_i, y'_i)^*$ to $Mismatches$.

* $x'_i$ and $y'_i$ can be easily extracted from $b_j(x) - b_j(y) = r'(x'_i - y'_i)$
    and $d_j(x) - d_j(y) = r'(x'^2_i - y'^2_i)$.
```

Lemma 2. *Let $L = \{i_1, i_2, \ldots, i_{k'}\}$ be the locations where x differ from y (where $k' \leq k$).$\forall j \in [1, 8k]$ If $\exists \gamma, \delta \in L s.t. j \in Q_\gamma \wedge j \in Q_\delta$ then $Pr(h'(i', j) = r') < \frac{1}{p}$ where i' and r' are the ones computed in the j-th step of procedure above.*

Proof: Since $h'(i', j)$ is a random number in F_p, the probability that it equals r' is $\frac{1}{p}$.

Conclusion: $Pr(|Mismatches| < HD(x,y) - k/2) < 1/n^2$
Note that for $k < \log n$ some minor changes needed to be done.
In order to find all locations of errors we maintain sketches as above for $k, k/2, k/4, \ldots, 1$ and after querying the sketch of k-bound we have $k/2$ errors found, we fix the rest of the sketches according to this errors, then we use the $k/2$-sketch to find another $k/4$ errors, fixing the remaining sketches, and so on. The total sketch size will be of $O(k \log p) = O(k \log n)$.

Theorem 5.1. *Given a stream x over alphabet Σ, there is an algorithm that maintain a sketch $s(x)$ that requires $O(k \log n)$ bits of memory and $O(\log n)$*

processing time per element and has the following property: Using sketches $s(x)$, $s(y)$ for two streams x and y where $HD(x,y) \leq k$ we can compute all locations i where $x_i \neq y_i$, and output the pair (x_i, y_i) for each of these locations. The probability of error is less than $\frac{1}{|x|}$.

Obviously, one can use this method also if he is interested only in $HD(x,y)$, and not in the exact locations that x differ from y and the values on those locations. this leads to the following theorem.

Theorem 5.2. *Given a stream x over alphabet Σ, there is an algorithm that maintain a sketch $s(x)$ that requires $O(k \log n)$ bits of memory and $O(\log n)$ processing time per element and has the following property: Using sketches $s(x)$, $s(y)$ for two streams x and y where $HD(x,y) \leq k$ we can compute $HD(x,y)$ with probability of error less than $1/n$.*

Proof: We randomly hash x into k^3 buckets. Next, we create x' by $x'[i] =$ sum of bucket i ($\mod \mathbb{F}_p$). We encode x' into binary stream $c(x')$ (as described in the first algorithm) and run the last sketching technique with x'. For two streams x, y The probability that two differences fall into the same bucket is less than $\frac{1}{2k}$, and therefore with a confidence of $1 - \frac{1}{2k}$ we have $HD(x,y) = HD(x',y') = \frac{1}{2}HD(c(x'), c(y'))$. Next,$O(k(\log|c(x')| + \log 2)) = O(k \log k)$.

6 Summary and Open Problem

We considered in this paper the sketching of the hamming distance. we presented a sketch of size $O(k \log n)$ that allows restoring the mismatches between different streams in time linear in the sketch size. Processing time per element is $O(\log n)$. Our sketch also fit the case that the stream does not come in the original order, for example if the data arrives in packets, or for the case we are informed of specific errors. Considering the more general case, of Edit distance it is still an open problem to present such a sketch. Although a sketch of size $O(k \log n)$ can be easily constructed by random linear combinations of the data, but the query time (restoring the errors between streams) will be exponential. In [1] a constant size sketch for a gap version of this problem, they identify between the case of distance $< k$ and the case where the distance exceeds k^2. Another algorithm to distinguish between $\Omega(n)$ and n^α edit distance is presented in [3].

References

[1] Bar-Yossef, Z., Jayram, T.S., Krauthgamer, R., Kumar, R.: Approximating edit distance efficiently. In: FOCS, pp. 550–559. IEEE Computer Society Press, Los Alamitos (2004)

[2] Bar-Yossef, Z., Jayram, T.S, Kumar, R., Sivakumar, D.: Manuscript (2003)

[3] Batu, T., Ergün, F., Kilian, J., Magen, A., Raskhodnikova, S., Rubinfeld, R., Sami, R.: A sublinear algorithm for weakly approximating edit distance. In: STOC, pp. 316–324. ACM, New York (2003)

[4] Cormode, G., Datar, M., Indyk, P., Muthukrishnan, S.: Comparing data streams using hamming norms (how to zero in). IEEE Trans. Knowl. Data Eng. 15(3), 529–540 (2003)

[5] Cormode, G., Paterson, M., Sahinalp, S.C, Vishkin, U.: Communication complexity of document exchange. In: SODA '00: Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms, pp. 197–206. Society for Industrial and Applied Mathematics (2000)

[6] Feigenbaum, J., Ishai, Y., Malkin, T., Nissim, K., Strauss, M., Wright, R.: Secure multiparty computation of approximations. In: Orejas, F., Spirakis, P.G., van Leeuwen, J. (eds.) ICALP 2001. LNCS, vol. 2076, pp. 927–938. Springer, Heidelberg (2001)

[7] Feigenbaum, J., Kannan, S., Strauss, M., Viswanathan, M.: An approximate l1-difference algorithm for massive data streams. SIAM J. Comput (and in Proceedings of the 40th Annual Symposium on Foundations of Computer Science), 32(1) 131–151, (2002) Appeared in Proceedings of the 40th Annual Symposium on Foundations of Computer Science, pp. 501–511 (1999)

[8] Gilbert, A.C, Guha, S., Indyk, P., Kotidis, Y., Muthukrishnan, S., Strauss, M.: Fast, small-space algorithms for approximate histogram maintenance. In: STOC '02: Proceedings of the thiry-fourth annual ACM symposium on Theory of computing, pp. 389–398. ACM Press, New York (2002)

[9] Guha, S., Koudas, N., Shim, K.: Data-streams and histograms. In: STOC '01: Proceedings of the thirty-third annual ACM symposium on Theory of computing, pp. 471–475. ACM Press, New York (2001)

[10] Indyk, P.: Stable distributions, pseudorandom generators, embeddings and data stream computation. In: FOCS '00: Proceedings of the 41st Annual Symposium on Foundations of Computer Science, Washington, DC, USA, p. 189. IEEE Computer Society Press, Los Alamitos (2000)

[11] Kushilevitz, E., Ostrovsky, R., Rabani, Y.: Efficient search for approximate nearest neighbor in high dimensional spaces. SIAM J. Comput. 30(2), 457–474 (2000)

[12] Muthukrishnan, S.: Data streams: algorithms and applications. In: SODA '03: Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms, pp. 413–413, Philadelphia, PA, USA, Society for Industrial and Applied Mathematics (2003)

[13] Starobinski, D., Trachtenberg, A., Agarwal, S.: Efficient pda synchronization. IEEE Trans. Mob. Comput. 2(1), 40–51 (2003)

[14] Trachtenberg, A., Starobinski, D., Agarwal, S.: Fast pda synchronization using characteristic polynomial interpolation. In: INFOCOM (2002)

Deterministic Length Reduction: Fast Convolution in Sparse Data and Applications

Amihood Amir[⋆], Oren Kapah, and Ely Porat

Department of Computer Science, Bar-Ilan University, Ramat-Gan 52900, Israel, +972 3 531-8770
{amir,kapaho,porately}@cs.biu.ac.il

Abstract. In this paper a deterministic algorithm for the length reduction problem is presented. This algorithm enables a new tool for performing fast convolution in sparse data. The proposed algorithm performs the convolution in $O(n_1 \log^3 n_1)$, where n_1 is the number of non-zero values in V_1. This algorithm assumes that V_1 is given in advance, and the V_2 is given in running time.

1 Introduction

The *d-Dimensional point set matching problem*, and its generalization - the *d-Dimensional sparse wildcard matching problem*, serve as powerful tools in numerous application domains. In the d-Dimensional point set matching problem, two sets of points $T, P \in \mathbb{N}^d$ consisting of n, m points, respectively, are given. The goal is to determine if there is a rigid transformation under which all the points in P are covered with points in T.
The d-Dimensional sparse wildcard matching problem is similar, except that every point in $\mathbb{N}^d$ is associated with a value. A match is declared if the values of coinciding points are equal.

Among the important application domains to which these problems contribute are the following:

Model based object recognition

In model-based recognition problems, a model of an object undergoes some geometric transformation that maps the model into a sensor coordinate system. This may be an image plane, a coordinate system from a 3d scanner, or a multidimensional feature space. When seeking occurrences of an object in a larger space, one needs to solve a problem similar to the point-set matching problem. For many applications one needs affine transformations, rather than just translations, in order to declare a match (e.g. [8,3]). Nevertheless, the point-set matching problem is a natural tool to use when seeking occurrences of an object in feature space, or in applications where translation alone suffices [9].

[⋆] The first author was partially supported by ISF grant 53/05.

Pharmacophore Identification

One of the purposes of structural computational biology is the development of pharmaceutical drugs. A pharmacophore is a three dimensional map of biological properties common to all active conformations of a set of ligands which exhibit a particular activity. Conceptually, a pharmacophore is a distillation of the functional attributes of ligands which accomplish a specific task. Pharmacophores are conceptual templates for drug design. Once extracted, a pharmacophore can be used as a model for the design of other molecules that can accomplish the same activity. The pharmacophore is an abstraction of a molecule, which must be congruent to the functional components of another desired molecule [6]. Again, the point-set matching is a tool towards solution.

Searching In Music Archives

Currently, many music archives are stored in MIDI format. In this representation, among other information, the pitch level of every played note is stored with a time-stamp. This information can be represented as points in a 2-Dimensional space where the axes are time and pitch. The problem of seeking a given melody in the archive is immediately reduced to a point set matching problem in 2-dimensional space [10].

Given the importance of the point-set matching problem it is not surprising that the problem has been widely considered in the literature in many variations, not the least of which in the algorithms literature. In [9] Cardoze and Schulman used a randomized algorithm to reduce the space size of T, P and then apply solve the problem in the reduced space. In [4] Cole and Hariharan proposed a solution to the *d-Dimensional Sparse Wildcard Matching*. Their solution consists of two steps. The first step is a *Dimension Reduction* where the inputs T, P are linearized into raw vectors T', P' of size polynomial in the number of non-zero values. The second step was a *Length Reduction* where each of the raw vectors T', P' was replaced by $\log n$ short vectors of size $O(n)$ where n is the number of non-zeros. The idea is that the mapping to the short vectors preserves the distances in the original vectors, thus the problem is reduced to a matching problem of short vectors, to which efficient solutions exist. The problem with the *length reduction* idea is that more then one point can be mapped into the same location, thus it is no longer clear whether there is indeed a match in the original vectors. The proposed solution of Cole and Hariharan was to create a set of $\log n$ pairs of vectors using $\log n$ hash function rather then a single pair of vectors. Their scheme reduced the failure probability. To our knowledge, all currently known efficient solutions to the problem are randomized.

In this paper we present the first **deterministic** algorithm for finding $\log n$ hash functions that reduce the size of the vectors to $O(n \log n)$. We guarantee that each non-zero value appears with no collisions in *at least* one of the vectors, thus eliminating the possibility of en error. Our algorithm also has the advantage that it is surprisingly simple and easy to implement.

In Section 4 we use our length reduction algorithm to solve the *Sparse Convolution* problem posed in [7], where the aim is to find the convolution vector

W of the two vectors V_1, V_2. It is assumed that the two vectors are not given explicitly, rather they are given as a set of $(index, value)$ pairs. Using the Fast Fourier Transform (FFT) algorithm, the convolution can be calculated in running time $O(N_1 \log N_2)$[5]. In our context, though, the vectors V_1, V_2 are sparse. The aim of the algorithm is to compute W in time proportional to the number of non-zero entries in W, which may be significantly smaller than $O(N_1)$.

It should be noted that our algorithm assumes that the point-set matching is done on an *archive*, i.e. we assume a given set of points which we pre-process in time quadratic in the number of non-zero points. Subsequently, we can solve any point-set matching query in time linear times polylog of the number of points. In the sparse convolution setting this means that the larger vector is fixed. The best known current deterministic solution takes time proportional to the product of the number of non-zero values in the two vectors, even if the same vector is re-used.

In Section 5 we demonstrate our fast sparse convolution algorithm on two applications: the *d-Dimensional Point Set Matching* and *Searching in Music Archives* problems. In both cases there is no known deterministic algorithm whose time is better than $O(n_1 n_2)$. Our algorithm offers a solution in time $O(n_1 \log^3 n_1)$.

2 Preliminaries and Notations

Throughout this paper, a capital letter (usually N) is used to denote the size of the vector, which is equivalent to the largest index of a non-zero value, and a small letter (usually n) is used to denote the number of non-zero values. It is assumed that the vectors are not given explicitly, rather they are given as a set of $(index, value)$ pairs, for all the non-zero values.

A convolution uses two initial functions, v_1 and v_2, to produce a third function w. We formally define a discrete convolution.

Definition 1. *Let V_1 be a function whose domain is $\{0, ..., N_1 - 1\}$ and V_2 a function whose domain is $\{0, ..., N_2 - 1\}$. We may view V_1 and V_2 as arrays of numbers, whose lengths are N_1 and N_2, respectively. The* discrete convolution of V_1 and V_2 *is the polynomial multiplication*

$$W[j] = \sum_{i=0}^{N_2-1} V_1[j+i] V_2[i].$$

In the general case, the convolution can be computed by using the Fast Fourier Transform (FFT) [5]. This can be done in time $O(N_1 \log N_2)$, in a computational model with word size $O(\log N_2)$. In the sparse case, many values of V_1 and V_2 are 0. Thus, they do not contribute to the convolution value. In our convention, the number of non-zero values of $V_1(V_2)$ is $n_1(n_2)$. Clearly, we can compute the convolution in time $O(n_1 n_2)$. The question posed by Muthukrishnan [7] is whether the convolution can be computed in time $o(n_1 n_2)$.

Cole and Hariharan's suggestion was to use *length reduction*. Suppose we can map all the non-zero values into a smaller vector, say of size $O(n_1 \log n_1)$. Suppose also that this mapping is alignment preserving in the sense that applying the same transformation on V_2 will guarantee that the alignments are preserved. Then we can simply map the the vectors V_1 and V_2 into the smaller vectors and then use FFT for the convolutions on the smaller vectors, achieving time $O(n_1 \log^2 n_1)$.

The problem is that to-date there is no known mapping with that alignment preserving property. Cole and Hariharan [4] suggested a randomized idea that answers the problem with high probability. The reason their algorithm is not deterministic is the following: In their length reduction phase, several indices of non-zero values in the original vector may be mapped into the same index in the reduced size vector. If the index of only one non-zero value is mapped into an index in the reduced size vector, then this index is denoted as *singleton* and the non-zero value is said to appear as a *singleton*. If more then one non-zero value is mapped into the same index in the reduced size vector, then this index is denoted as *multiple*. The multiple case is problematic since we can not be sure of the right alignment. Fortunately, Cole and Hariharan showed a method whereby in $O(\log n_1)$ tries, the probability that some index will *always* be in a multiple situation is small. In the next section we present a deterministic solution to the multiple problem.

3 Deterministic Length Reduction

Let P_i be a prime number. Construct the array V_i achieved by mapping the $(index, value)$ pair (i, v) to $(i \bmod P_i, v)$. Call array V_i the *reduced length vector* by P_i.

Our idea is to deterministically find a set $\{P_1, ..., P_i, ..., P_{\log n_1}\}$ of prime numbers in the preprocessing phase, such that when constructing the reduced length vectors $V_{1,i},\ i = 1, ..., \log n_1$, each non-zero input is guaranteed to appear as a *singleton* in **at least one** of the new vectors. Note that in order to save space, there is no need to keep the new vectors, since they can be reconstructed in running time given the set of prime numbers.

The rest of this section is devoted to finding this set of prime numbers. We distinguish between two cases.

1. The size of the original vector is polynomial in n_1 $(N_1 < n_1^c)$.
2. The size of the original vector is exponential in n_1.

Most of the practical applications fall into the first category. In addition, it is important to note that in case of N_1 exponential in n_1, just encoding the pairs requires $O(n_1^2)$ space. However, we provide an algorithm for the second case for the sake of completeness. This algorithm works in the *arithmetic model*, i.e. we assume that an address can fit in a computer word and that a word operation can be done in constant time.

3.1 Case 1: N_1 Is Polynomial in n_1

In the first case we find a set of $\log n_1$ prime numbers which enables the creation of $\log n_1$ vectors of size $O(n_1 \log n_1)$ such that each non-zero appears in at least one of the vectors as a *singleton*. For every prime number P_i the corresponding vector $V_{1,i}$ is created by setting the index of each non-zero value (i, v) to be the $i \bmod P_i$.

Observation 1. *For every i, two non-zeros can be mapped into the same location in $V_{1,i}$ only if P_i divides the distance between them.*

The following lemma is crucial to our algorithm.

Lemma 1. *Any two non-zeros can be mapped to the same location in at most c vectors.*

Proof. The distance between any two non-zeros is bounded by the size of the original vector $N_1 < n_1^c$, thus the number of different prime numbers greater then n_1 which divide the distance between them is bounded by $\log_{n_1} n_1^c = c$. $\square$

Since any non-zero can be mapped into the same location with at most $n_1 - 1$ other non-zeros, and with each of them at most c times, due to Lemma 1, then we get the following Corollary:

Corollary 1. *Any non-zero can appear as a multiple in not more then $n_1 c$ vectors.*

In the preprocessing step $2cn_1$ prime numbers of size $O(n_1 \log n_1)$ are selected. Corollary 1 assures us that they can be selected in any way. For example, the first $2cn_1$ prime numbers which are greater than $n_1 \log n_1$ can be chosen. For each prime number P_i the reduced length vector V_i is constructed. Then a set of $\log n_1$ prime numbers is chosen from them such that each of the non-zeros will appear as a *singleton* in at least one of the corresponding reduced length vectors.

The selection of the $\log n_1$ prime number is done as follows: Construct table A with n_1 columns and $2cn_1$ rows. Row i correspond to a prime number P_i and its reduced length vector $V_{1,i}$. A column corresponds to a non-zero value in V_1. The value of A_{ij} is set to 1 if non-zero j appears as a *singleton* in vector $V_{1,i}$. Due to Corollary 1, the number of zeros in each column can not exceed $n_1 c$. Thus, in each column there are 1's in at least half of the rows, which means that the table is at least half full. Since the table is at least half full there exist a row in which there is one in at least half of the columns. The prime number which generated this row is chosen. All the columns where there was a 1 in the selected row are deleted from the table.

Recursively another prime number is chosen and the table size is halved again, until all the columns are deleted. Since at each step at least half of the columns are deleted, the number of prime number chosen can not exceed $\log n_1$.

The algorithm appears in detail below.

```
Algorithm – $N_1 < n_1^c$, for constant $c$

1. Create a matrix $A$ of $n_1$ columns and $2cn_1$ rows.
2. For $i = 1$ to $2cn_1$
   (a) Select a new prime number $P_i$ of size $O(n_1 \log n_1)$.
   (b) Create a new vector $V_{1,i}$ from the input by mapping the index $\ell$ of each
       non-zero pair $(\ell, v)$ to: $\ell \bmod P_i$.
   (c) For each non-zero input $j$ from $1$ to $n_1$
   (d) If $j$ appears as a singleton in $V_{1,i}$ set $A_{ij} = 1$, else set $A_{ij} = 0$
3. Choose a row which is at least half full.
4. Delete all the columns in which $1$ appears in the selected row from the
   matrix.
5. while the matrix is not empty go to step 3.

end Algorithm
```

Correctness: Immediately follows from the discussion.

Time: Creating vector $V_{1,i}$ (row i) takes $O(n_1)$ time. Since we start with a full matrix of $O(n_1)$ rows then the initialization takes $O(n_1^2)$ time. Choosing the $\log n_1$ primes is done recursively. The recurrence is:

$$t(n_1^2) = n_1^2 + t(\frac{n_1^2}{2})$$

The closed form of this recurrence is $O(n_1^2)$.

Primality Testing: The set of prime numbers $\{P_1, ..., P_{\log n_1}\}$ is independent of the vector V_1. The same set can be used for all vectors with n_1 non-zero values. Thus we may assume that a prime numbers table is available. In any event, primality testing of number n can be currently done in $O(\log^{4+\epsilon} n)$ time [1], thus one can also generate prime numbers online.

3.2 Case 2: N_1 Is Exponential in n_1

In this case, the algorithm reduces the original vector to a single vector of size $O(n_1^4)$, where all the non-zeros appear as *singletons*, and then continues as in the first case. This is achieved by choosing a prime number of size $O(n_1^4)$, such that all of the non-zeros appear as *singletons* in the reduced length vector. When such a prime number is found, the algorithm in Subsection 3.1 is applied on the reduced length vector.

Finding such a prime number, which generates a reduced length vector in which all the non-zeros appear as *singletons*, is achieved by choosing sequentially prime numbers of size $O(n_1^4)$ and testing them.

Lemma 2. *There are at most* $\log N_1$ *prime number greater than* n_1^4 *that map two non-zeros of* V_1 *to the same location.*

Proof. The distance between any two non-zeros is bounded by the size N_1 of the original vector. Thus there are at most $\log N_1$ prime numbers that divide the distance between them. □

Since the number of distinct distances between any pair of non-zeros is bounded by n_1^2, then the number of prime numbers which generate a *multiple* location is bounded by $n_1^2 \times \log N_1$. Thus, at most $n_1^2 \times \log N_1$ prime numbers need to be tested before finding the desired prime number.

After finding such a prime number Q, a vector T' of size $2Q$ is created for the text, where each non-zero from index l in the text is mapped into two indices: (1) $l \text{mod} Q$. (2) $(l \text{mod} Q) + Q$.

The vector P' which is created from the pattern is of size Q, in which, each non-zero from index l in the pattern is mapped into $l \text{mod} Q$.

After obtaining the vectors T' and P', the algorithm continues as described in the previous section on the newly obtained vectors.

```
Algorithm – $N_1$ is exponential in $n_1$
  1. Select a new prime number $Q$ of size $O(n_1^4)$.
  2. Create a new vector ${V'}_1$ from the input by mapping the index $\ell$ of each
     non-zero pair $(\ell, v)$ to: $\ell \bmod Q$ and to: $(\ell \bmod Q) + Q$.
  3. If not all the non-zeros appear as singletons in $V_{1,i}$ go to step 2.
  4. Apply the algorithm for case 1 on the resulting vector.
end Algorithm
```

Correctness: Immediate from the discussion.

Time: Checking if the vector contains only *singletons* takes time $O(n_1)$ thus the time spent on steps 1 and 2 is $O(n_1)$. we repeat these steps at most $O(n_1^2) \times \log N_1$ times until we find a prime number which creates a vector with no *multiples*, thus the total time for the algorithm is $O(n_1^3 \times \log N_1) = O(n_1^4)$.

4 Fast Convolution in Sparse Data Using Length Reduction

We return to the problem of finding the convolution vector W of the two vectors V_1, V_2. It is assumed that the two vectors are not given explicitly, rather they are given as a set of $(index, value)$ pairs. While in the regular fast convolution the running time is $O(N_1 \log N_2)$, the aim here is to compute W in time proportional to the number of non-zero entries in W. This problem was posed in [7]. Note that the result of the convolution needs to be computed only for the locations in which every non-zero in V_2 is aligned with non-zero in V_1. In [4] Cole and Hariharan proposed a Las Vegas randomized algorithm which works in time $O(w \log^2 n_2)$ whose failure probability is inverse polynomial in n_2. We now present the first **deterministic** solution to this problem.

4.1 The Main Idea

The algorithm works in 2 steps: (1) Find the locations in which all non-zeros from V_2 are aligned with non-zeros in V_1 [4], and (2) Calculate the desired convolution by performing convolutions over all the $\log n_1$ vectors $V_{1,i}, V_{2,i}$, and sum the results for every location where an alignment was found. In order not to sum the results of a non-zero value more then once, if it appears as *singleton* in more then one reduced size vector, its value is zeroed after the first time.

Note that when reducing the vector size, more then one index in the original vector may be mapped into the same index in the reduced size vector. It may, therefore, happen that two aligned *singletons* will appear in the reduced size vectors $V_{1,i}, V_{2,i}$, while in the original vectors the non-zero in V_2 was aligned with an *empty* index in V_1. In order to eliminate this possibility the first step is mandatory. The first step eliminates this problem due to the fact that it ensures that every non-zero in V_2 is aligned with a non-zero in V_1, thus by the property of the length reduction, every non-zero in the reduced size vector $V_{2,i}$ has to be aligned with at least the value which was aligned with it in the original vector. Thus, if there is only one non-zero in $V_{1,i}$ aligned with a non-zero in $V_{2,i}$, they must have originated from two non-zero values which where aligned in the original size vectors V_1, V_2.

The algorithm for the first step uses the solution of Cole and Hariharan [4] for the **Shift Matching** problem.

Definition 2. *Let T be a length n text and P a length m pattern over alphabet $\mathbb{N} \cup \{\phi\}$, where ϕ is the* wildcard *symbol. We say that P* shift matches *T at location i if there exist an integer ℓ_i such that one of the following conditions holds for all non-wildcard symbols $P[j]$ in P:*

1. *The text character $T[i+j]$ aligned with $P[j]$ is a wildcard.*
2. $T[i+j] - P[j] = \ell_i$.

Cole and Hariharan [4] provide an algorithm for solving this problem in time $O(n \log m)$.

Note that for a location i where every non-zero $V_2[j]$ is aligned with a non-zero $V_1[i+j]$ in V_1, there exists a *shift match* between the indices of V_2 and the indices of V_1 and the shift is exactly j.

In the same paper Cole and Hariharan also give the following definition and Lemmas:

Definition 3. *A* wrap around placement of $V_{2,i}$ starting at location ℓ in $V_{1,i}$ *is a placement such that $V_{2,i}[k]$ is aligned with $V_{1,i}[(\ell + k) \bmod P_i]$*

Lemma 3. *Consider a wrap-around placement of $V_{2,i}$ in $V_{1,i}[j \bmod P_i]$. Then for each k, $0 \le k \le N_2$, $V_{2,i}[k \bmod P_i]$ is aligned with $V_{1,i}[(j+k) \bmod P_i]$.*

Lemma 4. *For any location where all non-zeros from V_2 are aligned with non-zeros in V_1, if a non-zero appears as a singleton in $V_{1,i}$ then it appears as a singleton in $V_{2,i}$.*

Following these lemmas, checking for existence of such an alignment is performed as follows:

For every P_i create the corresponding vectors $V_{1,i}, V_{2,i}$ where the indices of each non-zero are filled in the new vectors for every non-zero which appears as *singleton*. All the locations defined as *multiples* are assigned with a wildcard (zeroed). Perform a shift matching between $V_{1,i}, V_{2,i}$. Subsequently, the same vectors are created, this time with the value 1 instead of the indices, and the convolution is performed again to count the number of non-zeros which where considered in the shift match of every location.

Lemma 5. *A placement of V_2 in V_1 where all non-zeros in V_2 are aligned with non-zeros in V_1 exists if and only if there exist a shift match in all of the created vectors with the same shift (the shift amount is the location where the alignment exist), and the number of non-zeros which participate is n_2.*

Proof. $\Rightarrow$ Consider a location j where such an alignment exists. Now consider a non-zero at location $j+k$ in V_1 which aligns with a non-zero at location k in V_2. Due to the construction of the reduced vectors, there exist an i such that this non-zero appears as a *singleton* in $V_{1,i}$, and due to Lemma 4, the aligned non-zero, at location k in V_2 appears as a *singleton* in $V_{2,i}$. A shift match between them will give the difference between the indices which is j, and this is the same for all the n_2 non-zeros which participate in this alignment.
$\Leftarrow$ Consider a shift match where the indices of n_2 distinct non-zeros in $V_{2,i}, 0 \leq i \leq \log n_1$ are in distance j with the indices of non-zeros in $V_{1,i}$. Thus there exist n_2 non-zeros at locations $k_1, ..., k_{n_2}$ in V_2 and n_2 non-zeros at locations $j+k_1, ..., j+k_{n_2}$ in V_1. This means that at location j all the non-zeros in V_2 are aligned with non-zeros in V_1. □

To calculate the result of the convolution, a pair of vectors $V_{1,i}, V_{2,i}$ is created for every P_i. The values of all non-zeros are filled in the new vectors for every non-zero which appears as a *singleton*. All the locations defined as *multiples* are zeroed. Each value of non-zero in the V_1 is filled in only one of the vectors and in the rest of the vectors it is zeroed. In V_2 values are zeroed only if they appears at locations which are defined as *multiples*.

We are now ready to present the algorithm in detail.

4.2 The Algorithm

```
1. For every $i = 1$ to $\log n$
   (a) For each non-zero value $x \in V_1$
       i. If $V_{1,i}[index(x) \bmod P_i] = 0$ and not marked as multiple,
          – Set $V_{1,i}[index(x) \bmod P_i] = OriginalIndex(x)$.
       ii. Else,
          – Mark it as multiple.
          – Set $V_{1,i}[index(x) \bmod P_i] = 0$.
   (b) For each non-zero value $x \in V_2$ set
       i. If $V_{2,i}[index(x) \bmod P_i] = 0$ and not marked as multiple,
          – Set $V_{2,i}[index(x) \bmod P_i] = OriginalIndex(x)$.
       ii. Else,
          – Mark it as *multiple*.
          – Set $V_{2,i}[index(x) \bmod P_i] = 0$.
   (c) Perform Shift-Matching.
   (d) Replace all the indices in $V_{1,i}$ with the value of the non-zero and delete all
       the non-zeros which appeared in previous vectors.
   (e) Replace all the indices in $V_{2,i}$ with the value of the non-zero and delete all
       the non-zeros which appeared in previous vectors.
   (f) Calculate the convolution of $V_{1,i}, V_{2,i}$ using FFT.
   (g) Replace all non-zeros in $V_{1,i}, V_{2,i}$ with ones.
   (h) Calculate the convolution of $V_{1,i}, V_{2,i}$ using FFT.
2. Mark all locations where the results of the Shift Match is consistent over all the
   $\log n$ vectors.
3. For each such location
   (a) Sum the results of the convolution of the ones vectors.
   (b) If the sum equal $N_2$,
       i. Sum the results of the convolution of the values vectors.
       ii. Set $ResultVector[shift]$ to be the calculated sum.
```

Correctness: Follows from the discussion above.
Time: We perform $O(\log n_1)$ convolutions of the reduced length vectors, whose lengths are $O(n_1 \log n_1)$. Each convolution's time is therefore $O(n_1 \log^2 n_1)$ for a total time of $O(n_1 \log^3 n_1)$.

5 Applications

5.1 d-Dimensional Point Set Matching

In this problem, the input is given as d-dimensional text and pattern $T, P \in \mathbb{N}^d$, of size n, m respectively. The goal is to find the translations of P, under which each point in P is covered by a point in T. This problem is easily solved using the algorithm for fast convolution in sparse data. The first step is a dimension reduction [4], where from the d-dimensional input, a long raw vector is created. This is performed by simply concatenating the indices of each point, i.e for each point with indices $i_1, ..., i_d$, the index in the raw vector will be $i_1 * \mathbb{N}^{d-1} + ... + i_d * \mathbb{N}^0$. This is done for both T, P creating T', P', where 1 is placed in each index where there is a point. Then the fast convolution as suggested in this paper is performed. Any index where the result of the convolution is m gives the translation under which a match is found. The running time of this algorithm is the time of the convolution which is $O(n \log^3 n)$. This is the first deterministic algorithm which performs better the naive algorithm which takes time $O(nm)$.

5.2 Searching in Music Archives

This problem deals with archive files in MIDI format. Among other information, this format records the pitch level with the time-stamp for every note that was played. This information can be represented as points in a 2-Dimensional space where one axis is time, and the other axis of the pitch. Given such a database of melodies D of n notes, and a new melody P, the goal is to find wether P appears in D. Usually, if the pitch was translated by a constant value, P is still referred as the same melody, thus this problem is reduced to a point set matching problem in 2-dimensional space. The time of this algorithm will be $O(n \log^3 n)$, with a preprocessing step on the database in time $O(n^2)$ if $N < n^c$ or $O(n^4)$ if $N \in Exp(n)$.

Another possible problem may be finding an exact match with no translation on the pitch. This can be solved by creating from the input a vector where the time is the index, and the pitch is the value. Using the fast sparse convolution algorithm presented in this paper, this problem can be solved in time $O(n \log^3 n)$. To our knowledge, this is the first deterministic solution to the problem whose time is $o(nm)$, where m is the number of notes in the pattern melody [10].

6 Conclusion and Open Problems

Deterministic algorithms for Length Reduction and Sparse Convolution where presented in this paper. These can be used as tools to provide faster algorithms for several well known problems.

Several problems still remained open, such as the problem of *d-Dimensional Point Set Matching* under transformations such as rotation, or the problem of *Searching in Music with Scaling* (in both the pitch and the time), where nothing better then the naive algorithm is known.

Another important problem remains: Can the Length Reduction and Sparse Convolution problems be solved in real time without the need of the preprocessing step.

References

1. Berrizbeitia, P.: Sharpening primes is in p for a large family of numbers (November 2002) http://arxiv.org/abs/math.NT/0211334
2. Gottesfeld Brown, L.: A survey of image registration techniques. ACM Computing Surveys 24(4), 325–376 (1992)
3. Cheung, K.-W., Yeung, D.-Y., Chin, R.T.: Bidirectional deformable matching with application to handwritten character extraction. IEEE Transactions on Pattern Analysis and Machine Intelligence 24(8), 1133–1139 (2002)
4. Cole, R., Hariharan, R.: Verifying candidate matches in sparse and wildcard matching. In: Proc. 34st Annual Symposium on the Theory of Computing (STOC), pp. 592–601 (2002)
5. Cormen, T.H., Leiserson, C.E., Rivest, R.L.: Introduction to Algorithms. MIT Press, McGraw-Hill, Cambridge (1992)

6. Kavraki, L.: Pharmacophore identification and the unknown receptor problem. J Comput Aided Mol Des. 16(8-9), 653–681 (2002)
7. Muthukrishnan, S.: New results and open problems related to non-standard stringology. In: Galil, Z., Ukkonen, E. (eds.) Combinatorial Pattern Matching. LNCS, vol. 937, pp. 298–317. Springer, Heidelberg (1995)
8. Rucklidge, W.J.: Efficient visual recognition using the hausdorff distance. Springer, Heidelberg (1996)
9. Schulman, L., Cardoze, D.: Pattern matching for spatial point sets. In: Proc. 39th IEEE FOCS, pp. 156–165 (1998)
10. Ukkonen, E., Lemström, K., Mäkinen, V.: Sweepline the music! In: Klein, R., Six, H.-W., Wegner, L. (eds.) Computer Science in Perspective. LNCS, vol. 2598, pp. 330–342. Springer, Heidelberg (2003)

Guided Forest Edit Distance: Better Structure Comparisons by Using Domain-knowledge

Zeshan Peng and Hing-fung Ting*

Department of Computer Science, University of Hong Kong
{zspeng,hfting}@cs.hku.hk

Abstract. We introduce the guided forest edit distance problem, which measures the similarity of two forests under the guidance of a third forest. We give an efficient algorithm for the problem. Our problem is a natural generalization of many important structure comparison problems such as the forest edit distance problem, constrained sequence alignment problem and the longest constrained common subsequence problem. Our algorithm matches the performance of the best known algorithms for these problems.

1 Introduction

We propose and study an interesting extension of the classical *Forest Edit Distance* (`FED`) problem [11, 13], which asks for a sequence of edit operations with minimum cost that transforms a forest E to another forest F. This minimum cost reflects the similarity of the two forests; the smaller the cost, the larger the similarity. Since forests are commonly used to represent 2-D structures such as the secondary structures of RNA and XML documents, the `FED` problem has found applications in comparing these structures. To provide some natural mechanism for guiding the comparisons, we introduce the *Guided Forest Edit Distance* (`GFED`) problem. In addition to the forests E and F, the `GFED` problem has another input G, which is a sub-forest of both E and F. The problem asks for a sequence of edit operations with minimum cost that transforms E and F to some forests E' and F' such that E' and F' are identical and both E' and F' include G. Intuitively, G captures some domain knowledge on some important sub-structure of E and F that must be included in the comparison. It can be proved that when G is empty, our `GFED` problem is equivalent to the `FED` problem.

1.1 Some Motivations

Structure comparison is a fundamental technique in bioinformatics and there are many computational tools applying this technique. Although these tools have helped us tremendously in discovering new knowledge, they have the common weakness of not allowing users to guide/control the comparisons. This weakness becomes serious as we now have accumulated much biological knowledge and in

* This research is supported in part by Hong Kong RGC grant HKU-7172/06E.

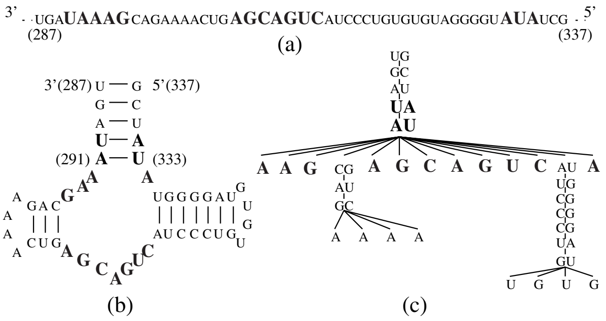

3' · · UGAUAAAGCAGAAAACUGAGCAGUCAUCCCUGUGUGUAGGGGUAUAUCG · · 5'
(287)
(337)
(a)
(b)
(c)

Fig. 1. (a) RNA primary structure, (b) its secondary structure and (c) its forest representation

many cases we have good ideas on how the structures should be compared. Many problems in bioinformatics need new structure comparison tools that provide natural mechanisms for guiding the comparisons.

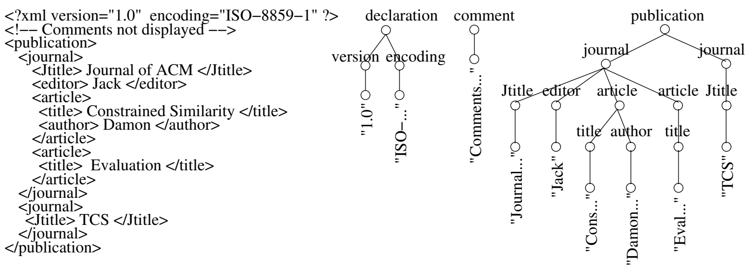
For example, the classification of RNAs can be done by comparing their secondary structures. As observed in [14], many of these structures can be represented by an ordered labelled forest (see Figure 1 for an example). There are many algorithms that determine the similarity of RNA by comparing their representing forests [3,10,13,6]. However, we may have better results if we can guide the comparisons using our knowledge of the RNA's motifs, which are some functionally important sub-structures in the RNAs and can also be represented by forests. For example, to decide whether a newly discovered RNA E belongs to some family, we compare it with the consensus F of the family and use their similarity to determine whether it belongs to the family. A important requirement for the comparison is that a motif G common to E and F should be aligned. This can be done by solving the GFED problem on input E, F and G.

Another motivation of the GFED problem comes from Extensible Markup Language (XML) documentations. The structures of XML documents are hierarchical and can be modelled by ordered labelled forests (see Figure 2). There are many forest-oriented algorithms for comparing XML docuements. For example, Nierman and Jagadish [8] used forest edit distance to measure the similarity of two XML documents. Cobéna *et al* [2] proposed a tree-based algorithm *diff* to detect changes in XML documents. To compare two XML documents E and F with the same Document Type Descriptor G[1], which is a sub-forest of both E and F, we should include and align this "bone-structure" G in our comparison; again, this problem can be formulated as the GFED problem.

1.2 The Results

The classical FED problem has been studied extensively. Tai [11] gave an algorithm for the problem that uses $O(|E||F|dp(E)^2dp(F)^2)$ time and space where

[1] For example, XML documents in different versions of warehouses.



```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!-- Comments not displayed -->
<publication>
  <journal>
    <Jtitle> Journal of ACM </Jtitle>
    <editor> Jack </editor>
    <article>
      <title> Constrained Similarity </title>
      <author> Damon </author>
    </article>
    <article>
      <title> Evaluation </title>
    </article>
  </journal>
  <journal>
    <Jtitle> TCS </Jtitle>
  </journal>
</publication>
```

Fig. 2. An XML document and its forest representation

$|X|$ is the number of nodes in forest X and $dp(X)$ is the depth of X. Zhang and Shasha [13] gave a better algorithm that uses $O(|E||F|)$ space and runs in $O(|E||F|\min\{|L(E)|, dp(E)\}\min\{|L(F)|, dp(F)\})$ time where $L(X)$ is the set of leaves of the forest X. Klein [5] later proposed an algorithm that runs in $O(|E|^2|F|\log|F|)$ time and use $O(|E||F|)$ space. Note that Zhang and Shasha's algorithm has performance better than that of Klein's in most cases.

We give an $O(|E||F||G|\min\{dp(E), |L(E)|\}\min\{dp(F), |L(F)|\}|L(G)|^2)$ time and $O(|E||F|\ |G||L(G)|^2)$ space algorithm for the `GFED` problem with input forests E and F, and guiding forest G. Note that when the guiding forest G is empty, our `GFED` problem becomes the traditional `FED` problem, and in this case, the time and space complexity of our algorithm becomes respectively $O(|E||F|\min\{dp(E), |L(E)|\}\min\{dp(F), |L(F)|\})$ and $O(|E||F|)$, matching those of Zhang and Shasha's algorithm [13].

Observe that a sequence can be regarded as a forest containing only one leaf, where the ancestor-descendent relationship among the nodes in the forest preserve the sequential relationship for the characters in the sequence. This observation enables us to use our algorithm to solve some constrained alignment problems on sequences, such as the *Constrained Sequence Alignment* (`CSA`) problem [7,9] and the *Longest Constrained Common Subsequence* (`LCCS`) problem [1,12]. Given two sequences e and f, the `CSA` problem and the `LCCS` problem ask to find respectively an optimal alignment and a longest common subsequence that include a constrained sequence g as a subsequence. For these problems, our algorithm runs in $O(|e||f||g|)$ time, matching the fastest known `CSA` algorithm [9,7] and `LCCS` algorithms [1].

2 Definitions and Notations

Ordered labelled trees are rooted trees whose nodes are labelled and the left-to-right orders among siblings are fixed. An *ordered labelled forest* is a sequence of ordered labelled trees. We define three edit operations between two such forests E and F as follows. Let a be a node in E and b a node in F.

– Operation (a, b): Relabelling a by the label of b.

– Operation $(a, -)$: Deleting a from the forest E. If a is not the root, we make every child of a to be a child of its parent $p(a)$.
– Operation $(-, b)$: Similar to $(a, -)$, except that we now delete node b from the forest F.

Every edit operation $(a, b) \neq (-, -)$ is associated with a cost $\delta(a, b)$ where $\delta(a, b) = 0$ if a, b are nodes having the same label, and $\delta(a, b) > 0$ otherwise. Furthermore, we have $\delta(a, b) = \delta(b, a)$. The cost $\delta(\Pi)$ of a sequence Π of edit operations is the total cost of all operations in Π. The *edit distance* $\delta(E, F)$ between E and F is the minimum cost of a sequence of edit operations (i.e., relabelling and deletions) that transforms E and F to two forests E' and F' such that E' and F' are identical[2].

We say that a forest G is a sub-forest of E if there exists a sequence of delete operations that transforms E to G. Suppose that G is a sub-forest of both E and F. The *guided edit distance* $\delta(E, F, G)$ of E and F with respect to G is the minimum cost of a sequence of edit operations that transforms E and F to forests E' and F' such that (i) E' and F' are identical, and (ii) G is a sub-forest of both E' and F'. We say that G is the *guiding forest*.

Tai [11] and Zhang and Shasha [13] also studied *edit mapping*, which is closely related to edit distance and will help us analyze our algorithm. Given two forests E and F, an *edit mapping* from E to F is a set $\mathcal{M}$ of node pairs (a, b) where $a \in E$ and $b \in F$, and any two pairs $(a_1, b_1), (a_2, b_2) \in \mathcal{M}$ satisfy the following conditions.

Uniqueness: $a_1 = a_2$ if and only if $b_1 = b_2$.
Ancestor-descendent: a_1 is an ancestor of a_2 if and only if b_1 is an ancestor of b_2.
Left-Right: a_1 is to the right of a_2 if and only if b_1 is to the right of b_2.

Define $\mathtt{Dom}(\mathcal{M}) = \{a \mid (a, b) \in \mathcal{M}\}$ and $\mathtt{Rng}(\mathcal{M}) = \{b \mid (a, b) \in \mathcal{M}\}$. Given any subset S of nodes of a forest X, define $X\|_S$, the sub-forest of X *restricted on* S, to be the one obtained by deleting from X those nodes that are not in S. The cost of $\mathcal{M}$ is defined to be

$$\delta(\mathcal{M}) = \sum_{(a,b)\in\mathcal{M}} \delta(a, b) + \sum_{i\in E\setminus\mathtt{Dom}(\mathcal{M})} \delta(a, -) + \sum_{b\in F\setminus\mathtt{Rng}(\mathcal{M})} \delta(-, j).$$

To study guided edit distance, we extend edit mapping to guided edit mapping as follows. Suppose that G is a sub-forest of both E and F. A *guided edit mapping* from E to F with respect to G is an edit mapping $\mathcal{M}$ from E to F such that there exists a subset $\mathcal{M}' \subseteq \mathcal{M}$ where $F\|_{\mathtt{Rng}(\mathcal{M}')} = G$. (It is easy to verified that we also have $E\|_{\mathtt{Dom}(\mathcal{M}')} = G$.) We say that $\mathcal{M}$ is optimal if $\delta(\mathcal{M})$ is minimum among all guided edit mappings. The following lemma shows that guided edit distance and guided edit mapping are closely related.

[2] Note that in the standard definition of edit distance we also includes insertions. It is easy to prove that our definition is equivalent to the standard one. We use this non-standard definition because we want to make the definition of guided edit distance more intuitive.

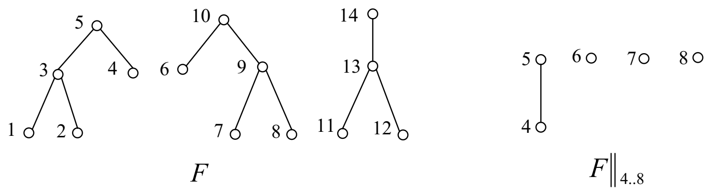


Fig. 3. A restricted sub-forest $F\|_{4..8}$ of forest F

Lemma 1. *Suppose that $\mathcal{M}$ is an optimal guided edit mapping from E to F with respect to G. Then, $\delta(\mathcal{M}) = \delta(E, F, G)$. Furthermore, we can construct a sequence Π of edit operations from $\mathcal{M}$ that gives $\delta(E, F, G)$ as follows: For every $(a, b) \in \mathcal{M}$, insert the relabelling operation (a, b), and for every $a \in E \setminus \mathtt{Dom}(\mathcal{M})$ and $b \in F \setminus \mathtt{Rng}(M)$, insert the delete operations $(a, -)$ and $(-, b)$ to Π.*

Proof. Lemma 2 of [13] implies this lemma for the case when $G = \emptyset$. It is straightforward to generalize their proof for the general case. □

Consider any forest F. Suppose that we have performed a post-order traversal of F. A node in F has name i if it is the ith visited node during the traversal. For any node i, let $F[i]$ denote the subtree rooted at i, and define $s(i)$ to be the node with the smallest name in $F[i]$. Define $i : j$ to be the set of nodes k in F such that $i \le k \le j$. (Note that if $j < i$, $i : j$ is the empty set.) Figure 3 shows a forest F and its sub-forest $F\|_{4..8}$ restricted on the set of nodes 4..8. Note that $s(3) = s(5) = 1$ and $s(9) = 7$.

Fact 1. *For any node i in F, $s(i)$ is the leftmost leaf in $F[i]$ and $F[i] = F\|_{s(i)..i}$. Furthermore, for any leaf ℓ, and any node k with $\ell \le k$, $F\|_{\ell..k}$ is a forest of rooted trees, i.e., $F\|_{\ell..k} = \langle F[h_1], F[h_2], \ldots, F[h_r]\rangle$ for some nodes $h_1 < h_2 < \cdots < h_r$.*

3 An Algorithm for the `GFED` Problem

In this section, we describe the algorithm `GFED` for computing $\delta(E, F, G)$. Note that if G is not a sub-forest of E or F, then $\delta(E, F, G)$ is undefined; for this case, we let $\delta(E, F, G) = \infty$. In the rest of paper, we make the assumption that G is a sub-forest of both E and F, which can be verified in $O((|E| + |F|)|G|)$ time [4]. Furthermore, to simplify our description, we assume that E and F are indeed trees. This is not a serious restriction because of the following fact.

Fact 2. *Let E' be a tree obtained by adding a root to the forest E and making the roots of all trees in E to be its children. Define F' from F similarly. Label the root of E' and F' by some label that does not appear in E, F and G. Then $\delta(E, F, G) = \delta(E', F', G)$.*

Since E and F are trees, $E = E[n]$ and $F = F[m]$ where n and m are the number of nodes of E and F, respectively. The core of our algorithm `GFED` is

Procedure DPtree$(E[i_o], F[j_o], G)$

```
1: Use Zhang and Shasha's algorithm to find $\delta(E[i], F[j], \emptyset)$ for all $i \in E$ and $j \in F$.
2: for $i := s(i_o), \cdots, i_o$ do
3:    for $j := s(j_o), \cdots, j_o$ do
4:       for $h := 1, \cdots, |G|$ do
5:          for $k := 1, \cdots, (|G| - h + 1)$ do
6:             if node $k$ is a leaf in $G$, find $\delta(E\|_{s(i_o)..i}, F\|_{s(j_o)..j}, G\|_{k..(k+h-1)})$
```

the following dynamic programming procedure DPtree$(E[i_o], F[j_o], G)$, which computes $\delta(E[i_o], F[j_o], G)$ for the trees $E[i_o], F[j_o]$.

Thus, to find $\delta(E, F, G)$, we call DPtree$(E[n], F[m], G)$. The task of GFED is simply to prepare all the necessary values for the dynamic programming procedures to proceed by calling them systematically as follows: for $i = 1, \ldots, n$ and for $j = 1, \ldots, m$, call DPtree$(E[i], F[j], G)$.

Let $1 \leq i_o \leq n$ and $1 \leq j_o \leq m$. Since $G = G\|_{1..|G|}$, $E[i_o] = E\|_{s(i_o)..i_o}$ and $F[j_o] = F\|_{s(j_o)..j_o}$, we have $\delta(E[i_o], F[j_o], G) = \delta(E\|_{s(i_o)..i_o}, F\|_{s(j_o)..j_o}, G\|_{1..|G|})$. DPtree finds this value in a bottom-up manner. As a preprocessing, the first step of the procedure calls Zhang and Shasha's algorithm to compute $\delta(E, F)$, or equivalently, $\delta(E, F, \emptyset)$. Their algorithm also uses dynamic programming, and after the execution, we find $\delta(E[i], F[j], \emptyset)$ for all $i \in E$ and $j \in F$. We now explain how to find the value in Line 6 of the procedure. Suppose that k is a leaf of G. Note that if $G\|_{k..(k+h-1)} \neq \emptyset$ and one of the forests $E\|_{s(i_o)..i}$ and $F\|_{s(j_o)..j}$ is empty, then $\delta(E\|_{s(i_o)..i}, F\|_{s(j_o)..j}, G\|_{k..(k+h-1)})$ is trivially ∞. Otherwise, we find the value using the Lemma 2.

Lemma 2. *Let $r(G\|_{k..l}) = \{i_1, i_2, \ldots, i_q\}$ be the roots of the subtrees in the forest $G\|_{k..l}$ where $k \leq l$. Then, for any $s(i_o) \leq i \leq i_o$ and $s(j_o) \leq j \leq j_o$, $\delta(E\|_{s(i_o)..i}, F\|_{s(j_o)..j}, G\|_{k..l})$ is equal to the minimum of the following values:*

(i) $\delta(E\|_{s(i_o)..i-1}, F\|_{s(j_o)..j}, G\|_{k..l}) + \delta(i, -)$,

(ii) $\delta(E\|_{s(i_o)..i}, F\|_{s(j_o)..j-1}, G\|_{k..l}) + \delta(-, j)$,

(iii) $\delta(E\|_{s(i_o)..s(i)-1}, F\|_{s(j_o)..s(j)-1}, G\|_{k..l}) + \delta(E[i], F[j], \emptyset)$,

(iv) $\delta(E\|_{s(i_o)..s(i)-1}, F\|_{s(j_o)..s(j)-1}, G\|_{k..s(p)-1}) +$
$\delta(E\|_{s(i)..i-1}, F\|_{s(j)..j-1}, G\|_{s(p)..l}) + \delta(i, j)$ *for each* $p \in r(G\|_{k..l})$,

(v) $\delta(E\|_{s(i_o)..s(i)-1}, E\|_{s(j_o)..s(j)-1}, G\|_{k..s(l)-1}) +$
$\delta(E\|_{s(i)..i-1}, F\|_{s(j)..j-1}, G\|_{s(l)..l-1}) + \delta(i, j)$ *if l has the same label of j.*

Proof. Let $\mathcal{M}$ be the mapping from $E\|_{s(i_o)..i}$ to $F\|_{s(j_o)..j}$ with respect to $G\|_{k..l}$ given by Lemma 1 such that $\delta(E\|_{s(i_o)..i}, F\|_{s(j_o)..j}, G\|_{k..l}) = \delta(\mathcal{M})$, and let Π be the sequence of edit operations constructed from $\mathcal{M}$ as described by the lemma. We consider the following cases.

Case a: Node i is not mapped to any node in F (i.e., $i \notin$ Dom$(\mathcal{M})$). Lemma 1 asserts that Π has the operation $(i, -)$. Furthermore, by the principle of optimality, the sequence $\Pi - \langle(i, -)\rangle$, the one obtained by removing the operation $(i, -)$ from Π, has optimal cost $\delta(E\|_{s(i_o)..i-1}, F\|_{s(j_o)..j}, G\|_{k..l})$. Hence,

$\delta(\mathcal{M}) = \delta(E\|_{s(i_o)..i-1}, F\|_{s(j_o)..j}, G\|_{k..l}) + \delta(i, -)$, which is the value (i) in the lemma.

Case b: Node j is not mapped by any node in E (i.e., $j \notin \texttt{Rng}(\mathcal{M})$). By similar argument, we have $\delta(\mathcal{M}) = \delta(E\|_{s(i_o)..i}, F\|_{s(j_o)..j-1}, G\|_{k..l}) + \delta(-, j)$, which is value (ii).

Case c: Both i and j are mapped (i.e., $i \in \texttt{Dom}(\mathcal{M})$ and $j \in \texttt{Rng}(\mathcal{M})$). Since i and j are respectively the rightmost roots in $E\|_{s(i_o)..i}$ and $F\|_{s(j_o)..j}$, by the Ancestor-descendent and Left-Right conditions, i must be mapped to j by $\mathcal{M}$. It follows that for any $(a, b) \in \mathcal{M}$, a is a node of $E[i]$ if and only if b is a node of $F[j]$ (because of the Ancestor-descendent condition). Define $\mathcal{M}_1 = \{(a, b) \in \mathcal{M} \mid b \notin F[j]\}$ and $\mathcal{M}_2 = \{(a, b) \in \mathcal{M} \mid b \in F[j]\}$, and we have

$$\begin{aligned}
&\mathcal{M}_1 \text{ is an edit mapping from } E\|_{s(i_o)..s(i)-1} \text{ to } F\|_{s(j_o)..s(j)-1},\\
&\mathcal{M}_2 - \{(i, j)\} \text{ is an edit mapping from } E\|_{s(i)..i-1} \text{ to } F\|_{s(j)..j-1}, \text{ and}\\
&\delta(\mathcal{M}) = \delta(\mathcal{M}_1) + \delta(\mathcal{M}_2 - \{(i, j)\}) + \delta(i, j). \qquad (*)
\end{aligned}$$

Since $\mathcal{M}$ is a guided edit mapping with guiding forest $G\|_{k..l}$, there is a subset $\mathcal{M}'$ of $\mathcal{M}$ such that $(F\|_{s(j_o)..j})\|_{\texttt{Rng}(\mathcal{M}')} = G\|_{k..l}$. We consider the following subcases.

– $\texttt{Rng}(\mathcal{M}')$ does not contain any node in $F[j] = F\|_{s(j)..j}$. Then, for every $(a, b) \in \mathcal{M}'$, $b \notin F[j]$ and hence $\mathcal{M}' \subseteq \mathcal{M}_1$, $\mathcal{M}' \cap \mathcal{M}_2 = \emptyset$. It follows that

$$(F\|_{s(j_o)..s(j)-1})\|_{\texttt{Rng}(\mathcal{M}')} = (F\|_{(s(j_o)..s(j)-1)\cup(s(j)..j)})\|_{\texttt{Rng}(\mathcal{M}')} = G\|_{k..l}.$$

Together with (*), and by the principle of optimality, we conclude $\mathcal{M}_1$ is an optimal edit mapping from $E\|_{s(i_o)..s(i)-1}$ to $F\|_{s(j_o)..s(j)-1}$ with respect to $G\|_{k..l}$ and hence $\delta(\mathcal{M}_1) = \delta(E\|_{s(i_o)..s(i)-1}, F\|_{s(j_o)..s(j)-1}, G\|_{k..l})$. Similarly, we can argue that $\delta(\mathcal{M}_2 - \{(i, j)\}) = \delta(E\|_{s(i)..i-1}, F\|_{s(j)..j-1}, \emptyset)$. Substitute them in the equation of (*), we conclude that value (iii) is a bound.

– $\texttt{Rng}(\mathcal{M}')$ contains some node in $F[j]$ but $j \notin \texttt{Rng}(\mathcal{M}')$. Define $\mathcal{M}'_1 = \mathcal{M}' \cap \mathcal{M}_1$ and $\mathcal{M}'_2 = \mathcal{M}' \cap \mathcal{M}_2$. Note that $\mathcal{M}' = \mathcal{M}'_1 \cup \mathcal{M}'_2$, and since $(F\|_{s(j_o)..j})\|_{\texttt{Rng}(\mathcal{M}')} = G\|_{k..l} = \langle G[i_1], G[i_2], \ldots, G[i_q]\rangle$, there is some $i_p \in r(G\|_{k..l})$ such that $(F\|_{s(j_o)..s(j)-1})\|_{\texttt{Rng}(\mathcal{M}'_1)} = \langle G[i_1], G[i_2], \ldots, G[i_{p-1}]\rangle = G\|_{k..i_{p-1}} = G\|_{k..s(i_p)-1}$ and $(F\|_{s(j)..j})\|_{\texttt{Rng}(\mathcal{M}'_2)} = \langle G[i_p], G[i_{p+1}], \ldots, G[i_q]\rangle = G\|_{s(i_p)..i_q} = G\|_{s(i_p)..l}$. Together with $\mathcal{M}'_1 \subseteq \mathcal{M}_1$, $\mathcal{M}'_2 \subseteq \mathcal{M}_2$, $j \notin \texttt{Rng}(\mathcal{M}')$ and (*), we conclude

$$\begin{aligned}
\delta(\mathcal{M}_1) &= \delta(E\|_{s(i_o)..s(i)-1}, F\|_{s(j_o)..s(j)-1}, G\|_{k..s(i_p)-1}), \text{ and}\\
\delta(\mathcal{M}_2 - \{(i, j)\}) &= \delta(E\|_{s(i)..i-1}, F\|_{s(j)..j-1}, G\|_{s(i_p)..l}).
\end{aligned}$$

Substitute them in the equation in (*), we have value (iv) in the lemma.

– $j \in \texttt{Rng}(\mathcal{M}')$. In this case, node j of $F\|_{s(j_o)..j}$ and node l of $G\|_{k..l}$ must have the same label. The analysis of this case is almost identical to that of the previous case. The only difference is that we now have $j \in \texttt{Rng}[\mathcal{M}'_2]$. Since j is the root of $F[j]$, we conclude that $(F\|_{s(j)..j})\|_{\texttt{Rng}(\mathcal{M}'_2)}$ is a single tree, and as $F[j]$ is the rightmost tree in F, $(F\|_{s(j)..j})\|_{\mathtt{Rng}(\mathcal{M}_2')}$ must be the right most tree in $G\|_{k..l}$, namely $G[l] = G\|_{s(l)..l}$. We conclude that

$$\delta(M_1) = \delta(E\|_{s(i_o)..s(i)-1}, F\|_{s(j_o)..s(j)-1}, G\|_{k..s(l)-1}), \text{ and}$$
$$\delta(M_2 - \{(i,j)\}) = \delta(E\|_{s(i)..i-1}, F\|_{s(j)..j-1}.G\|_{s(l)..l-1}).$$

(Note that the guiding forest in the last equality is $G\|_{s(l)..l-1}$, not $G\|_{s(l)..l}$ because the root l is associated with the root j of $E[j]$.) Substitute them in (*), we get value (v).

All cases are considered and $\delta(\mathcal{M})$ is the minimum among the above cases. □

Theorem 3. *When E and F are trees, the algorithm* `GFED` *correctly finds $\delta(E, F, G)$ using $O(|E|^2|F|^2|G||L(G)|^2)$ time and $O(|E||F||G||L(G)|^2)$ space.*

Proof. To be given in the full paper.

□

We explain how to speed up this algorithm and give complete proofs on the correctness and complexities of this faster algorithm in the next section.

4 Speeding up `GFED`

Recall that the depth $dp(i)$ of a node i is defined to be the number of ancestors of i, and the depth $dp(X)$ of a forest X is the maximum depth of a node in X. In this section, we adapt a clever trick of Zhang and Shasha [13] to reduce the running time of `GFED` to $O(|E||F||G| \min\{dp(E), |L(E)|\} \min\{dp(F), |L(F)|\}|L(G)|^2)$. In our discussion, we still assume that E and F are trees. For any node $i \in E$, we say that i is a *key-root* if either it is the root of E or it has a left sibling. Let $\kappa(E)$ be the set of key-roots of E. The following lemma relates $\kappa(E)$ to the set $L(E)$ of leaves of E.

Lemma 3. $|\kappa(E)| \leq |L(E)|$.

Proof. Note that for any key-root $i \in \kappa(E) - \{n\}$, $s(i) \in L(E) - \{1\}$; $s(i) \in L(E)$ because of Fact 1 and it is not equal to the leave 1 because by definition, i has a left-sibling i' and by the property of post-order traversal, $1 \leq i' < s(i)$. We claim that the function $\varphi : (\kappa(E) - \{n\}) \to (L(E) - \{1\})$ where $\varphi(i) = s(i)$ is one-to-one, and the lemma follows. Suppose to the contrary that there are two key-roots i and i' such that $s(i) = s(i') = \ell$. Since E is a tree, one of the key-root, say i', is on the path from the other key-root i to ℓ. Since $s(i) = \ell$ is the leftmost leaf in $E[i]$ (Fact 1 again), we conclude that the nodes on the path from i to ℓ do not have any left sibling; in particular, i' does not have any left sibling and $i' \notin \kappa(E)$, a contradiction. □

The main observation for speeding up `GFED` is that to prepare enough values for the bottom-up computation to proceed, we do not need to call the procedure `DPtree`$(E[i], F[j], G)$ for all possible i, j; we only need to focus on those i, j that are both key-roots. The following algorithm `fastGFED` implements this modification.

Algorithm `fastGFED`(E, F, G)

```
1: Use Zhang and Shasha's algorithm to find $\delta(E[i], F[j], \emptyset)$ for all $i \in E$ and $j \in F$.
2: for $i = 1, \ldots, n$ do
3:    for $j = 1, \ldots, m$ do
4:       if $i \in \kappa(E)$ and $j \in \kappa(F)$, call DPtree$(E[i], F[j], G)$;
```

Theorem 4. *The space and time complexity of* `fastGFED` *is* $O(|E||F||G||L(G)|^2)$ *and* $O(|E||F||G| \min\{dp(E), |L(E)|\} \min\{dp(F), |L(F)|\}|L(G)|^2)$*, respectively.*

Proof. Obviously, the algorithm uses no more that the $O(|E||F||G||L(G)|^2)$ space of `GFED`. For the time, note that for any $1 \le i \le |E|$ and $1 \le j \le |F|$, the procedure `DPtree`$(E[i], F[j], G)$ computes $|E[i]||F[j]||G||L(G)|$ different values of $\delta(E\|_{s(i)..x}, F\|_{s(j)..y}, G\|_{a..b})$ and each of them can be found by Lemma 2 using $O(|L(G)|)$ time. Thus, `DPtree`$(E[i], F[j], G)$ runs in $O(|E[i]||F[j]||G||L(G)|^2)$ time, and `fastGFED` runs in time (in the order of)

$$\sum_{i\in\kappa(E)} \sum_{j\in\kappa(F)} |E[i]||E[j]||G|(|L(G)|)^2 = \Big(\sum_{i\in\kappa(E)} |E[i]|\Big)\Big(\sum_{j\in\kappa(F)} |F[j]|\Big)|G|(|L(G)|)^2.$$

Note that $\sum_{i\in\kappa(E)} |E[i]| = \sum_{x\in E} \sum_{i\in E} \alpha(i, x)$ where $\alpha(i, x) = 1$ if $i \in \kappa(E)$ and $x \in E[i]$, and 0 otherwise. Since $x \in E[i]$ if and only if i is an ancestor of x, we conclude that $\sum_{i\in E} \alpha(i, x) = |\{i \mid i \in \kappa(E) \text{ and } i \in anc(x)\}| = |\kappa(E) \cap anc(x)| \le \min\{|anc(x)|, |\kappa(E)|\} \le \min\{dp(E), |L(E)|\}$ (Lemma 3) where $anc(x)$ is the set of ancestors of x. Hence, $\sum_{i\in\kappa(E)} |E[i]| = \sum_{x\in E} \sum_{i\in E} \alpha(i, x) \le |E| \min\{dp(E), |L(E)|\}$. Similarly, $\sum_{j\in\kappa(F)} |F[j]| = |F| \min\{dp(F), |L(F)|\}$, and the theorem follows. □

Theorem 5. *Algorithm* `fastGFED`(E, F, G) *finds* $\delta(E, F, G)$ *correctly.*

Proof. Since $n \in \kappa(E)$ and $m \in \kappa(F)$, the last `DPtree` called by the algorithm is `DPtree`$(E[n], F[m], G)$, which computes $\delta(E, F, G) = \delta(E\|_{1..n}, F\|_{1..m}, G)$ before exiting. Thus, `fastGFED` finds $\delta(E, F, G)$ correctly if during the execution, it has pre-computed enough values to complete every `DPtree`$(E(i_o), F(j_o), G)$ it calls. Recall that `DPtree`$(E(i_o), F(j_o), G)$ uses Lemma 2 to compute iteratively the values $\delta(E\|_{s(i_o)..i}, F\|_{s(j_o)..j}, G\|_{k..l})$. Note that for those values required by the lemma that are of the form $\delta(E\|_{s(i_o)..x}, F\|_{s(j_o)..y}, G\|_{a..b})$, we have either $x < i$ or $y < j$ and thus are computed earlier in the execution of `DPtree`$(E[i_o], F[j_o], G)$. Those values $\delta(E[i], F[j], \emptyset)$ are computed by Zhang and Shasha's algorithm executed at Line 1 of `fastGFED`.

Note that the remaining values are $\delta(E\|_{s(i)..i-1}, E\|_{s(j)..j-1}, G\|_{a..b})$ where $s(i_o) \le i \le i_o$ and $s(j_o) \le j \le j_o$, which appear in (iv) and (v) of Lemma 2. By the property of post-order traversal, we have $i \in E[i_o]$ and $j \in E[j_o]$. Let i' be the least ancestor of i that is a key-root. Note that $i' \le i_o$ because by the design of `fastGFED`, i_o is a key-root ancestor of i. Note that all the nodes on the path from i to i' do not have any left sibling (they are not key-roots) and this implies $s(i') = s(i)$. We define j' similarly and we also have $s(j') = s(j)$. Since $i' \le i_o$ and $j' \le j_o$ and both of them are key-roots, `fastGFED` would have called `DPtree`$(E[i'], F[j'], G)$. If either $i' < i_o$ or $j' < j_o$, the call `GFED`$(E[i'], F[j'], G)$ would have finished and computed the value $\delta(E\|_{s(i')..i-1}, E\|_{s(j')..j-1}, G\|_{a..b}) = \delta(E\|_{s(i)..i-1}, E\|_{s(j)..j-1}, G\|_{a..b})$. If $i = i_o$ and $j = j_o$, $\delta(E\|_{s(i)..i-1}, E\|_{s(j)..j-1}, G\|_{a..b}) = \delta(E\|_{s(i_o)..i-1}, E\|_{s(j_o)..j-1}, G\|_{a..b})$ is computed before the computation of $\delta(E\|_{s(i_o)..i}, E\|_{s(j_o)..j}, G\|_{k..l})$. Hence, all the necessary values for computing $\delta(E\|_{s(i_o)..i}, E\|_{s(j_o)..j}, G\|_{k..l})$ are ready. □

References

1. Chin, F.Y.L., Santis, A.D., Ferrara, A.L., Ho, N.L., Kim, S.K.: A simple algorithm for the constrained sequence problems. Information Processing Letters. 90(4), 175–179 (2004)
2. Cobena, G., Abiteboul, S., Marian, A.: Detecting changes in XML documents. In: Proc. of the 18th IEEE International Conference on Data Engineering, pp. 41–52 (2002)
3. Jiang, T., Lin, G., Ma, B., Zhang, K.: A general edit distance between RNA structures. Journal of Molecular Biology 9(2), 371–388 (2002)
4. Kilpelainen, P., Mannila, H.: Ordered and unordered tree inclusion. SIAM Journal on Computing 24(2), 340–356 (1995)
5. Klein, P.N.: Computing the edit-distance between unrooted ordered trees. In: Proc. of the 6th European Symposium on Algorithms(ESA 1998), pp. 91–102 (1998)
6. Lin, G.H., Ma, B., Zhang, K.: Edit distance between two RNA structures. In: Proc. of the 5th international conference on Computational molecular biology, pp. 211–220 (2001)
7. Lu, C.L., Huang, Y.P.: A memory-efficient algorithm for multiple sequence alignment with constraints. Bioinformatics 21(1), 20–30 (2004)
8. Nierman, A., Jagadish, H.V.: Evaluating structural similarity in XML documents. In: Proc. of the 5th International Workshop on the Web and Databases, pp. 61–66 (2002)
9. Peng, Z.S., Ting, H.F.: Time and space efficient algorithms for constrained sequence alignment. In: Proc. of the 9th International Conference on Implementation and Application of Automata, pp. 237–246 (2004)
10. Shapiro, B.A., Zhang, K.: Comparing multiple RNA secondary structures using tree comparisons. Bioinformatics 6, 309–318 (1990)
11. Tai, K.C.: The tree-to-tree correction problem. Journal of the ACM 26(3), 422–433 (1979)
12. Tsai, Y.T.: The constrained longest common subsequence problem. Information Processing Letters, 88(4) (2003)
13. Zhang, K., Shasha, D.: Simple fast algorithms for the editing distance between trees and related problems. SIAM Journal on Computing 18(6), 1245–1262 (1989)
14. Zuker, M., Sankoff, D.: RNA secondary structures and their prediction. Bull. Math. Biol. 46, 591–621 (1984)

Space-Efficient Algorithms for Document Retrieval

Niko Välimäki and Veli Mäkinen*

Department of Computer Science, University of Helsinki, Finland.
{nvalimak,vmakinen}@cs.helsinki.fi

Abstract. We study the *Document Listing* problem, where a collection D of documents $d_1, \ldots, d_k$ of total length $\sum_i d_i = n$ is to be preprocessed, so that one can later efficiently list all the ndoc documents containing a given query pattern P of length m as a substring. Muthukrishnan (SODA 2002) gave an optimal solution to the problem; with $O(n)$ time preprocessing, one can answer the queries in $O(m + \text{ndoc})$ time. In this paper, we improve the space-requirement of the Muthukrishnan's solution from $O(n \log n)$ bits to $|CSA| + 2n + n \log k(1 + o(1))$ bits, where $|CSA| \leq n \log |\Sigma|(1 + o(1))$ is the size of any suitable *compressed suffix array* (CSA), and Σ is the underlying alphabet of documents. The time requirement depends on the CSA used, but we can obtain e.g. the optimal $O(m+\text{ndoc})$ time when $|\Sigma|, k = O(\text{polylog}(n))$. For general $|\Sigma|, k$ the time requirement becomes $O(m \log |\Sigma| + \text{ndoc} \log k)$. Sadakane (ISAAC 2002) has developed a similar space-efficient variant of the Muthukrishnan's solution; we obtain a better time requirement in most cases, but a slightly worse space requirement.

1 Introduction and Related Work

The *inverted file* is by far the most often utilized data structure in the Information Retrieval domain, being a fundamental part of Internet search engines such as Google, Yahoo, and MSN Search. In its simplest form, an inverted file consists of a set of *words*, where each word is associated with the list of documents containing it in a given document collection D. The *Document Listing* problem is then solved by printing out the document list associated with the given query word.

This approach works as such only for documents consisting of distinct words, such as natural language documents. If one considers the more general case, where a document consists of a sequence of symbols without any word boundaries, then the inverted file approach may turn out to be infeasible. There are basically two alternative approaches to proceed; (i) create an inverted file over all substrings of D; or (ii) create an inverted file over all *q-grams* of D, i.e., over all substrings of D of length q.

Approach (i) suffers from (at least) quadratic space requirement, and approach (ii) suffers from the slow time requirement affected by the size of intermediate

* Funded by the Academy of Finland under grant 108219.

results; In approach (ii) the search for a pattern P of length $m > q$ proceeds by covering the pattern with q-grams, and searching each q-gram from the inverted file. In this case, each q-gram needs to be associated with a list of exact occurrence positions inside the documents. One can then merge the occurrence lists to spot the co-occurrences of the q-grams covering P. The total time requirement is determined by the length of the longest intermediate occurrence list associated with a q-gram of P. This can be significantly more than the number of occurrences of whole P.

Despite the above mentioned weakness, a q-gram based inverted file is very effective in practice, and it has found its place in important applications such as in a popular search tool BLAST for genome data [1]. One of the attractive properties of an inverted file is that it is easily compressible while still supporting fast queries [13]. In practice, an inverted file occupies space close to that of a compressed document collection.

Nevertheless, it is both practically and theoretically interesting to study index structures having the same attractive space requirement properties as inverted files and at the same time having provably good performance on queries.

If one can afford to use more space, then so-called *full-text indexes* such as *suffix arrays* and *suffix trees* [3] provide good alternatives to inverted files. These structures can be used, among many other things, to efficiently solve the *Occurrence Listing* problem, where all occurrence positions of a given pattern P are sought for from the document collection.

Recently, several *compressed full-text indexes* have been proposed that achieve the same good space requirement properties as inverted files [10]. These indexes also provide provably good performance in queries. Many of these indexes have been implemented and experimented, and shown effective in practice as well.[1]

Puglisi et al. [11] made an important sanity-check study on the practical efficiency of compressed full-text indexes when compared to inverted files. They concluded that, indeed, pattern searches are typically faster using compressed full-text indexes. However, when the number of occurrence positions becomes high (e.g. more than $5,000$ on English text [11]), inverted files become faster. This experimental observation is consistent with the theoretical properties of the indexes; In compressed full-text indexes each occurrence position must be decoded, and this typically takes $O(\log n)$ time per occurrence. On the other hand, the shorter the pattern, the more occurrences there are, meaning that in a q-gram based inverted file, the intermediate occurrence lists are not significantly larger than the final result.

Let us now get back to the original motivation — the Document Listing problem. Unlike for the Occurrence Listing problem, there is yet no simultaneously space- and time-efficient solution known for this problem. Interestingly, even achieving a good time requirement is nontrivial. One can, of course, solve the problem by pruning the answer given by any solution to the Occurrence Listing problem. This is clearly not optimal, since the *number of occurrences*, nocc,

[1] See http://pizzachili.dcc.uchile.cl/

may be arbitrarily greater than the *number of documents*, ndoc, containing these occurrences!

Matias et al. [8] gave the first efficient solution to the Document Listing problem; with $O(n)$ time preprocessing of a collection D of documents $d_1, \ldots, d_k$ of total length $\sum_i d_i = n$, they could answer the document listing query on a pattern P of length m in $O(m \log k + \text{ndoc})$ time. The algorithm uses a *generalized suffix tree* augmented with extra edges making it a directed acyclic graph.

Muthukrishnan [9] simplified the solution in [8] by replacing the augmented edges with a divide and conquer strategy. This simplification resulted into optimal query time $O(m + \text{ndoc})$.

The space requirements of both the above solutions are $O(n \log n)$ bits. This is significantly more than the collection size, $O(n \log |\Sigma|)$ bits, where Σ is the underlying alphabet. More importantly, the space usage is much more than that of an inverted file.

Sadakane [12] has developed a space-efficient version of the Muthukrishnan's algorithm. He obtains a structure taking $|CSA| + 4n + O(k \log \frac{n}{k}) + o(n)$ bits. However, his structure is not anymore time-optimal, as document listing queries take $O(m + \text{ndoc} \log^{\epsilon} n)$ time on a constant alphabet, where $|CSA|$ is the size of any *compressed suffix array* that supports *backward search* and $O(\log^{\epsilon} n)$ time retrieval of one suffix array value. Here $\epsilon > 0$ is a constant affecting the size of such compressed suffix array.

Very recently, Fischer and Heun [6] have shown how the space of the Sadakane's structure can be reduced to $|CSA| + 2n + O(k \log \frac{n}{k}) + o(n)$ bits; they also improve the extra space needed to build the Sadakane's structure from $O(n \log n)$ bits to $O(n \log |\Sigma|)$ bits.

In this paper, we give an alternative space-efficient variant of Muthukrishnan's structure that is capable of supporting document listing queries in optimal time under realistic parameter settings. Our structure takes $|CSA| + 2n + n \log k(1 + o(1))$ bits, where $|CSA| \leq n \log |\Sigma|(1 + o(1))$ is the size of any compressed suffix array supporting backward search. (We do not require the CSA to support the retrieval of suffix array values as the Sadakane's structure.) The time requirement depends on the underlying CSA used, but we can obtain e.g. optimal $O(m + \text{ndoc})$ time when $\Sigma, k = O(\text{polylog}(n))$. For general Σ, k the time requirement becomes $O(m \log |\Sigma| + \text{ndoc} \log k)$.

We also show that a recent data structure by Bast et al. [2] proposed for *output-sensitive autocompletion search* can be used for the Document Listing problem. The space requirement is asymptotically the same as above but the time requirement is slightly inferior.

We provide some preliminary experimental evidence to show that our structure is significantly faster than an inverted file, especially when ndoc << nocc.

2 Preliminaries

A *string* $T = t_1 t_2 \cdots t_n$ is a sequence of *symbols* from an ordered *alphabet* Σ. A *substring* of T is any string $T_{i \ldots j} = t_i t_{i+1} \cdots t_j$, where $1 \leq i \leq j \leq n$. A *suffix* of

T is any substring $T_{i\dots n}$, where $1 \leq i \leq n$. A *prefix* of T is any substring $T_{1\dots j}$, where $1 \leq j \leq n$. A *pattern* is a short string over the alphabet Σ. We say that the pattern $P = p_1 p_2 \cdots p_m$ *occurs* at the position j of the *text* alias *document* string T iff $p_1 = t_j, p_2 = t_{j+1}, \dots, p_m = t_{j+m-1}$. Length of a document T is denoted $|T|$.

Definition 1 (Document Listing problem). *Given a collection D of documents $d_1, d_2, \dots, d_k$ of total length $\sum_{i=1}^{k} |d_i| = n$, the Document Listing problem is to build an index for D such that one can later efficiently support the* document listing query *of listing for any given pattern P of length m the documents that contain P as a substring.*

The output of the document listing query in Def. 1 is a subset of document identifiers $\{1, 2, \dots, k\}$.

We say that a *space-optimal solution* to the Document Listing problem is an index that occupies $n \log \Sigma (1 + o(1))$ bits. This is the asymptotic lower-bound given by the Kolmogorov complexity for any representation of D. Here we count the representation of D as part of the index. For compressible documents, it is possible to achieve better bounds.

Likewise, we say that a *time-optimal solution* to the Document Listing problem is an index that can be constructed in $O(n)$ time, and that supports listing of documents in $O(m + \text{ndoc})$ time, where ndoc is the size of the output.

3 Time-Optimal Document Listing

Muthukrishnan [9] obtained a time-optimal solution to the document listing problem, but the solution was yet not space-optimal. In the sequel, we show that one can adjust the solution to obtain the space-optimality as well in certain parameter settings.

Let us describe the Muthukrishnan's solution in a level suitable for our purposes. We use a generalized suffix array instead of the generalized suffix tree used in the original proposal.

Let the document collection $d_1, d_2, \dots, d_k$ be represented as a concatenation $D = d_1 d_2 \cdots d_k$. A *generalized suffix array* for the document collection D is then an array $A[1 \dots n]$ containing the permutation of $1, 2, \dots, n$ such that $D_{A[i],n} <_b D_{A[i+1],n}$ for $1 \leq i < n$. Here $<_b$ is the normal lexicographic order $<$ of strings, except that the document boundaries are handled separately; the order of suffixes is computed assuming (virtually) a special symbol $\$_i$ inserted after each document d_i, such that $\$_1 < \$_2 < \cdots < \$_k < c$, where $c \in \Sigma$.[2]

Using two binary searches on A, one can easily locate the maximal range $[sp, ep]$ of A such that the pattern P is a prefix of all the suffixes $D_{A[sp],n}$, $D_{A[sp+1],n}$, ..., $D_{A[ep],n}$. Now, the remaining task is to report the ndoc documents containing those suffixes without having to spend time on each occurrence.

[2] Concrete insertion of symbols $\$_i$ is the standard definition. However, here it would make the alphabet size grow to $|\Sigma| + k$, affecting the later results. Therefore we opt for the virtual handling of boundaries.

Muthukrishnan introduces a divide and conquer strategy on two arrays $C[1 \ldots n]$ and $E[1 \ldots n]$ for this task. The array E simply lists the document numbers of each suffix in the order they appear in the suffix array A, that is, $E[i] = j$ if $A[i]$ points inside the document d_j in the concatenation D. The array C is defined as

$$C[i] = \max\{j \mid j < i, E[i] = E[j]\} \tag{1}$$

(if there is no such $j < i$, then $C[i] = -1$). The algorithm is based on the following observation.

Lemma 1 ([9]). *Let $[sp, ep]$ be the maximal range of A corresponding to suffixes that have pattern P as a prefix. The document k' contains P if and only if there exists precisely one $j \in [sp, ep]$ such that $E[j] = k'$ and $C[j] < sp$.*

To proceed, the table C is assumed to be preprocessed for constant time *Range Minimum Queries (RMQ)*. That is, on a given interval I, one can compute $\min_{i \in I} C[i]$ in constant time (as well as the argument i giving the minimum). This preprocessing can be done in $O(n)$ time [4].

The divide and conquer algorithm starts by finding the $i \in [sp, ep]$ such that $C[i]$ is the smallest (using the constant time RMQ). If $C[i] > sp$ then there is no document to report and the process stops. Otherwise, we output $E[i]$ and repeat the same process recursively on $[sp, i-1]$ and on $[i+1, ep]$. One can see that all the documents containing P are reported and each of them only once [9].

By replacing the generalized suffix array with a generalized suffix tree, one can find the range $[sp, ep]$ in $O(m)$ time on a constant size alphabet (on general alphabets, this takes $O(m \log |\Sigma|)$ time). Afterward, the reporting of the documents takes $O(\text{ndoc})$ time.

Theorem 1 ([9]). *Document Listing problem can be solved using an index structure occupying $O(n \log n)$ bits and having an $O(n)$ time construction algorithm. The index supports document listing queries in $O(m + ndoc)$ time on constant size alphabets. On general alphabets, the query time becomes $O(m \log |\Sigma| + ndoc)$.*

4 Space-Optimal Document Listing

We will derive a space-efficient version of the index structure derived in the previous section. This is accomplished by representing the arrays A, C, E, as well as the structure for RMQ, compressed. The algorithm for answering the queries stays the same.

Representing A. Instead of suffix array A we can use any compressed suffix array supporting the backward search [10]. Different time/space tradeoffs are possible, e.g. one can find the range $[sp, ep]$ of A containing the occurrences of P in $O(m)$ time, when $|\Sigma| = O(\text{polylog}(n))$, using an index of size $nH_h + o(n \log |\Sigma|)$ bits [5]. Here $H_h = H_h(D) \leq \log |\Sigma|$ is the *h-th order empirical entropy* of the text collection D (lower bound for the average number of bits needed to code a symbol using a fixed code table for each h-context in D). The given space bound holds for small h (see [5]), but for our purposes it is enough to use an estimate $H_h \leq \log|\Sigma|$ that is independent of h. That is, the index size can be expressed as $n \log|\Sigma|(1+o(1))$ bits. The space bound is valid for general alphabets as well, but the time requirement becomes $O(m \log|\Sigma|)$.

Representing C and E. The crucial observation is that the array C is not needed at all, but instead it can be represented implicitly via the array E. Recall the definition of C in Eq. (1). We can re-express the definition as

$$C[i] = select_{E[i]}(E, rank_{E[i]}(E,i) - 1), \tag{2}$$

where $rank_{k'}(E,i)$ gives the number of times the value k' appears in $E[1,i]$ and $select_{k'}(E,j)$ gives the index of E containing the j-th occurrence of the value k' (and we define $select_{k'}(E,0) = -1$ to handle the boundary case). It is easy to see that Eqs. (1) and (2) are identical; both express the link from the value $E[i]$ to its predecessor in E.

The array E can be seen as a sequence of symbols from the alphabet $\Sigma' = \{1, 2, \ldots, k\}$. The functions *rank* and *select* on such sequences are an essential part of the index structure we are already using as a representation of the suffix array A [5]. That is, we can represent E using $n \log|\Sigma'|(1+o(1)) = n \log k(1+o(1))$ bits of space for a so-called *generalized wavelet tree* [5]. Each value of E as well as the queries $rank_{k'}(E,i)$ and $select_{k'}(E,j)$ can then be computed in constant time when $k = O(\text{polylog}(n))$. On general $k \leq n$, the space stays the same, but the time requirement becomes $O(\log k)$.

Representing RMQ structure. The algorithm requires range minimum queries on C. As C is now implicitly represented via E, some modifications to the existing RMQ structures are needed. Sadakane [12] gives a succinct representation of the RMQ structure in [4] requiring $4n+o(n)$ bits on top of the array indexed. Fischer and Heun [6] have recently found another constant time RMQ representation occupying only $2n + o(n)$ bits on top of the array indexed. We can use either of these representations on top of our implicit representation of C. Explicit values of C are mainly needed only during construction; queries access a constant number of values in C, which can then be computed from the generalized wavelet tree representation of E.

We have obtained the following result.

Theorem 2. *The Document Listing problem can be solved using an index structure occupying $n \log|\Sigma|(1+o(1)) + 2n + n \log k(1+o(1))$ bits and having an $O(n \log|\Sigma| + n \log k)$ time construction algorithm. The index supports document listing queries in $O(m + ndoc)$ time when $|\Sigma|, k = O(polylog(n))$. On general alphabets and on general document collection sizes $k \leq n$, the query time component $O(m)$ becomes $O(m \log|\Sigma|)$ and $O(ndoc)$ becomes $O(ndoc \log k)$, respectively.*

Proof. The space and query time bounds should be clear from the above discussion. The construction time is achieved by building first the suffix array of D

using e.g a linear time construction algorithm, and then building the generalized wavelet tree on the Burrows-Wheeler transform and other structures to form the compressed representation of the suffix array [5]. The array E and its generalized wavelet tree are constructed similarly. The bottleneck is the generalized wavelet tree construction. Although not explicitly stated in [5], it can be constructed in time linear in the final result size in bits. $\square$

Notice that the obtained structure is space-optimal when $k = o(|\Sigma|)$ and $|\Sigma| = O(\text{polylog}(n))$.

The space requirement of the $O(n \log |\Sigma|)$ time construction algorithm is $O(n \log n)$ bits. A slower construction algorithm, taking $O(n \log n \log |\Sigma|)$ time, that uses the same asymptotic space as the final structure, is easy to derive using the dynamic wavelet tree proposed in [7]. Also the RMQ-solution by Fischer and Heun can be constructed within these space and time limits.

4.1 Extended Functionality

So far our structure can list the documents containing the pattern. A useful extension would be to *list the occurrences inside the selected documents.*

For motivation, imagine that the collection represents a file system; files are concatenated into D in the order of a depth-first traversal. A search on the file system would return the documents containing the query word. A user could select documents of interest from the list, and ask to see the occurrence positions. Most likely the user would like to see a *context* around each occurrence to judge the relevance.

Also, to guide the selection of relevant documents, it would be good to have the matching documents listed in the order of expected relevance; one way would be to *list the documents in the order of number of occurrences of the pattern.*

We can support the above described functionalities with our index. First, to have the matching documents listed in the order of relevance, one may use the fact that the number of occurrences $\text{nocc}_{k'}$ in document k' can be computed by

$$\text{nocc}_{k'} = rank_{k'}(E, ep) - rank_{k'}(E, sp - 1). \tag{3}$$

After sorting the numbers $\text{nocc}_{k'}$, one has achieved the goal.

Second, to list the occurrences lying in a selected document (or in the range of documents lying in a subdirectory), one may use the existing functionalities of compressed suffix arrays to list *all* the occurrences. To make this faster than just pruning the list, one can proceed as follows. Let k' be a selected document known to contain occurrences of P. To find the last occurrence inside the suffix array range $[sp, ep]$, one can query $i = select_{k'}(E, rank_{k'}(E, ep))$ and compute the occurrence position $A[i]$ by decoding the entry from the compressed suffix array. The process is continued step-by-step with $i = select_{k'}(E, rank_{k'}(E, i - 1))$ until $i < sp$. At each step one can also output a context around each occurrence.

5 Autocompletion Search and Document Listing

Recently, Bast et al. [2] studied a related problem of providing a space-efficient index for the so-called *output-sensitive autocompletion search* problem.

Consider a user typing a query in a document retrieval tool. The tool can provide repeatedly a list of matching documents while the user is still completing the query. This interactive feature is called *autocompletion*.

To avoid the trivial solution of starting the query on each new symbol from scratch, Bast et al. proposed an online output-sensitive method that focuses the query W on the so-far matching documents, say $D' \subset D$. They developed an index structure supporting this query assuming a text collection consisting of distinct words. As they mention [2, p. 153], the structure can be extended to the case of full-text sequences by using e.g. a *suffix array* on top of their index.

Let us now consider how the structure of Bast et al., when applied to the full-text setting, can be used to solve the Document Listing problem. Their AUTOTREE structure is basically a succinct version of the following; a balanced binary tree built on the lexicographically ordered suffixes of D such that each node lists the documents containing suffixes in its subtree. In fact, the succinct coding they propose (TREE+BITVEC) is almost identical to a balanced wavelet tree built on our array E! However, the difference comes in the queries supported; they engineer the representation suitable for fast autocompletion searches and do not exploit the divide and conquer strategy to speed up the search. Instead they avoid reporting the same occurrence repeatedly by pruning the tree suitably.

Nevertheless, one cannot obtain exactly as fast reporting time for the Document Listing problem using AUTOTREE as what we obtain in Theorem 2; the reason is that AUTOTREE outputs word-in-document pairs where the prefix of the word matches the query pattern. In our case, this means outputting all suffixes whose prefix matches the pattern (that is, all occurrences). However, one can adjust the search algorithm in [2, page 154, step 2] to work only on the $O(\log n)$ nodes covering the search range; in the worst case, each document can appear in each of those nodes and be reported $O(\log n)$ times. This gives $O(\mathrm{ndoc} \log n)$ reporting time which is still inferior to our structure. The space usage of both structures are closely the same, since in our terminology the size of AUTOTREE is $n\lceil \log k \rceil$ bits (their N equals our number of suffixes n, and their n is our k).

6 Comparison to Sadakane's Solution

Our solution is very similar to the Sadakane's solution [12]. The difference is in the use of generalized wavelet trees to represent the document numbers associated with the suffixes. Sadakane is able to represent this information in $O(k \log \frac{n}{k})$ bits, as we use $n \log k(1 + o(1))$ bits. However, to retrieve the document numbers, he needs to use the expensive $O(\log^{\epsilon} n)$ time operation to retrieve a suffix array value. Choosing a small value of ϵ affects the multiplicative constant factor in the size of the underlying compressed suffix array inversely. We can do this in constant time using generalized wavelet tree, when the number of documents is $k = \text{polylog}(n)$.

7 Preliminary Experimental Results

Extrapolating from the experimental results in [11] for the Occurrence Listing problem, one could expect that our structure is superior to inverted files in typical document listing settings, that is, where the inverted file needs to examine all occurrences and our index can work directly on the document level. The space should be quite close as well; Inverted files use more space in representing the document collection as such, while in our index the document collection is compressed inside the index. Our $n \log k(1 + o(1))$ may exceed the space needed for the inverted file, when k is large.

We have a preliminary implementation ready of our structure, which is yet not fully optimized for time or space usage. However, it is enough for validating the claim of being faster when the number of occurrence positions nocc is large but the number of matching documents ndoc is small. To see this, we compared our structure to the same inverted file implementation as used in [11]. We used different size prefixes of a catenated English text collection as the input and partitioned each prefix to k equal size "documents", with varying k. We selected randomly from the text collection two sets of patterns, each containing 1000 patterns of length $m = 3$ and $m = 4$, respectively. The small pattern length was used to guarantee that ndoc << nocc. Table 1 shows the results for the inverted file, and Table 2 shows the results for our structure. The running times are the total time needed for the 1000 queries, and values nocc/pattern and ndoc/pattern are the average output sizes.

Table 1. Running times and index sizes for inverted file. The results correspond to Occurrence Listing queries; the time needed for pruning the Document Listing result would be negligible. We used parameter values Block size = 64, q-gram size = 3, and list intersection limit = 10000 inside the inverted file implementation, see [11].

		$m = 3$		$m = 4$	
|text| (MB)	|index| (MB)	time (s)	nocc/pattern	time (s)	nocc/pattern
1	2.00	0.35	1334.5	0.13	267.4
25	49.12	8.61	29085.4	2.23	3077.1
50	98.11	17.46	57136.0	4.29	6428.9

The results are as expected: The running times of both structures depend almost linearly on their output sizes. When the input size grows, but the number of documents (i.e. the maximum output size for our structure) stays the same, our structure becomes faster than the inverted file. On short collections the running times are almost the same.

This result with short patterns, combined with the observation in [11] that compressed suffix arrays are faster than inverted files for Occurrence Listing

Table 2. Running times and index sizes for our structure, with varying k

			$m = 3$		$m = 4$	
|text| (MB)	k	|index| (MB)	time (s)	ndoc/pattern	time (s)	ndoc/pattern
1	1	2.12	0.0029	1	0.0036	1
1	50	3.13	0.35	40.7	0.21	27.9
1	100	3.31	0.68	72.9	0.35	44.0
1	150	3.41	0.95	100.0	0.45	55.7
1	200	3.47	1.21	123.4	0.54	65.0
1	250	3.52	1.41	144.3	0.61	72.8
25	200	84.76	3.1	180.1	2.1	132.2
50	200	169.29	3.7	185.1	2.7	148.6

queries on long patterns, gives reason to argue that an index based on compressed suffix arrays may be an attractive alternative to inverted files as a generic building block for flexible Information Retrieval tasks.

8 Future Work

Muthukrishnan [9] studied many other important Information Retrieval tasks such as *document mining* (finding the documents where a given pattern appears more often than a given threshold) and *proximity queries* (where the occurrences of patterns appearing near to each others are searched for). The solutions to these problems use the same kind of ideas as for the Document Listing problem, but are somewhat more complicated, and hence more difficult to make space-efficient. It is an interesting future challenge to derive space-efficient solutions to these problems such that they would become competitive with inverted file -based solutions in practice.

Acknowledgement

We wish to thank Gonzalo Navarro for bringing [9] to our attention and anonymous reviewers for bringing [8] and[12] to our attention.

References

1. Altschul, S.F., Gish, W., Miller, W., Myers, E.W., Lipman, D.J.: Basic local alignment search tool. Journal of Molecular Biology 215(3), 403–410 (1990)
2. Bast, H., Mortensen, C.W., Weber, I.: Output-sensitive autocompletion search. In: Crestani, F., Ferragina, P., Sanderson, M. (eds.) SPIRE 2006. LNCS, vol. 4209, pp. 150–162. Springer, Heidelberg (2006)
3. Crochemore, M., Rytter, W.: Jewels of Stringology. World Scientific (2002)
4. Farach-Colton, M., Bender, M.A.: The lca problem revisited. In: Proc. Latin American Theoretical Informatics (LATIN), pp. 88–94 (2000)

5. Ferragina, P., Manzini, G., Mäkinen, V., Navarro, G.: Compressed representation of sequences and full-text indexes. ACM Transactions on Algorithms (to appear)
6. Fischer, J., Heun, V.: A new succinct representation of rmq-information and improvements in the enhanced suffix array. In: Proceedings of the International Symposium on Combinatorics, Algorithms, Probabilistic and Experimental Methodologies (ESCAPE'07), LNCS. Springer (to appear)
7. Mäkinen, V., Navarro, G.: Dynamic entropy compressed sequences and full-text indexes. In: Lewenstein, M., Valiente, G. (eds.) CPM 2006. LNCS, vol. 4009, pp. 306–317. Springer, Heidelberg (2006)
8. Matias, Y., Muthukrishnan, S., Sahinalpk, S.C., Ziv, J.: Augmenting suffix trees with applications. In: Bilardi, G., Pietracaprina, A., Italiano, G.F., Pucci, G. (eds.) ESA 1998. LNCS, vol. 1461, pp. 67–78. Springer, Heidelberg (1998)
9. Muthukrishnan, S.: Efficient algorithms for document retrieval problems. In: Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms (SODA'02), pp. 657–666 (2002)
10. Navarro, G., Mäkinen, V.: Compressed full-text indexes (Article 2). ACM Computing Surveys 39(1) (2007)
11. Puglisi, S.J., Smyth, W.F., Turpin, A.: Inverted files versus suffix arrays for locating patterns in primary memory. In: Crestani, F., Ferragina, P., Sanderson, M. (eds.) SPIRE 2006. LNCS, vol. 4209, pp. 122–133. Springer, Heidelberg (2006)
12. Sadakane, K.: Space-efficient data structures for flexible text retrieval systems. Journal of Discrete Algorithms. ISAAC 2002 5(1), 12–22 (2002)
13. Witten, I.H., Moffat, A., Bell, T.C.: Managing gigabytes scompressing and indexing documents and images, 2nd edn. Morgan Kaufmann Publishers Inc, San Francisco, CA, USA (1999)

Compressed Text Indexes with Fast Locate

Rodrigo González[⋆] and Gonzalo Navarro[⋆⋆]

Dept. of Computer Science, University of Chile
{rgonzale,gnavarro}@dcc.uchile.cl

Abstract. Compressed text (self-)indexes have matured up to a point where they can replace a text by a data structure that requires less space and, in addition to giving access to arbitrary text passages, support indexed text searches. At this point those indexes are competitive with traditional text indexes (which are very large) for *counting* the number of occurrences of a pattern in the text. Yet, they are still hundreds to thousands of times slower when it comes to *locating* those occurrences in the text. In this paper we introduce a new compression scheme for suffix arrays which permits locating the occurrences extremely fast, while still being much smaller than classical indexes. In addition, our index permits a very efficient secondary memory implementation, where compression permits reducing the amount of I/O needed to answer queries.

1 Introduction and Related Work

Compressed text indexing has become a popular alternative to cope with the problem of giving indexed access to large text collections without using up too much space. Reducing space is important because it gives one the chance of maintaining the whole collection in main memory. The current trend in compressed indexing is *full-text compressed self-indexes* [13,1,4,14,12,2]. Such a self-index (for short) replaces the text by providing fast access to arbitrary text substrings, and in addition gives indexed access to the text by supporting fast search for the occurrences of arbitrary patterns. These indexes take little space, usually from 30% to 150% of the text size (note that this includes the text). This is to be compared with classical indexes such as suffix trees [15] and suffix arrays [10], which require at the very least 10 and 4 times, respectively, the space of the text, plus the text itself. In theoretical terms, to index a text $T = t_1 \dots t_n$ over an alphabet of size σ, the best self-indexes require $nH_k + o(n \log \sigma)$ bits for any $k \leq \alpha \log_\sigma n$ and any constant $0 < \alpha < 1$, where $H_k \leq \log \sigma$ is the k-th order empirical entropy of T [11,13][1]. Just the uncompressed text alone would need $n \log \sigma$ bits, and classical indexes require $O(n \log n)$ bits on top of it.

The search functionality is given via two operations. The first is, given a pattern $P = p_1 \dots p_m$, *count* the number of times P occurs in T. The second

⋆ Work supported by Mecesup Grant UCH 0109, Chile.

⋆⋆ This work was funded by a grant from Yahoo! Research Latin America.

[1] In this paper log stands for $\log_2$.

is to *locate* the occurrences, that is, to list their positions in T. Current self-indexes achieve a counting performance that is comparable in practice with that of classical indexes. In theoretical terms, for the best self-indexes the complexity is $O(m(1+\frac{\log\sigma}{\log\log n}))$ and even $O(1+\frac{m}{\log_\sigma n})$, compared to $O(m\log\sigma)$ of suffix trees and $O(m\log n)$ or $O(m+\log n)$ of suffix arrays. Locating, on the other hand, is far behind, hundreds to thousands of times slower than their classical counterparts. While classical indexes pay $O(occ)$ time to locate the *occ* occurrences, self-indexes pay $O(occ\log^\varepsilon n)$, where ε can in theory be any number larger than zero but is in practice larger than 1. Worse than that, the memory access patterns of self-indexes are highly non-local, which makes their potential secondary-memory versions rather unpromising. Extraction of arbitrary text portions is also quite slow and non-local compared to having the text directly available as in classical indexes. The only implemented self-index which has more local accesses and faster locate is the LZ-index [12], yet its counting time is not competitive.

In this paper we propose a suffix array compression technique that builds on well-known regularity properties that show up in suffix arrays when the text they index is compressible [13]. This regularity has been exploited in several ways in the past [7,14,8], but we present a completely novel technique to take advantage of it. We represent the suffix array using differential encoding, which converts the regularities into true repetitions. Those repetitions are then factored out using Re-Pair [6], a compression technique that builds a dictionary of phrases and permits fast local decompression using only the dictionary (whose size one can control at will, at the expense of losing some compression). We then introduce some novel techniques to further compress the Re-Pair dictionary, which can be of independent interest. We also use specific properties of suffix arrays to obtain a much faster compression losing only 1%–14% of compression.

As a result, for several text types, we reduce the suffix array to 20–70% of its original size, depending on its compressibility. This reduced index can still extract any portion of the suffix array very fast by adding a small set of sampled absolute values. We prove that the size of the result is $O(H_k\log(1/H_k)n\log n)$ bits for any $k \le \alpha\log_\sigma n$ and any constant $0 < \alpha < 1$. Note that this reduced suffix array is not yet a self-index as it cannot reproduce the text.

This structure can be used in two ways. One way is to attach it to a self-index able of counting, which in this process identifies as well the segment of the (virtual) suffix array where the occurrences lie. We can then locate the occurrences by decompressing that segment using our structure. The result is a self-index that needs 1–3 times the text size (that is, considerably larger than current self-indexes but also much smaller than classical indexes) and whose counting and locating times are competitive with those of classical indexes, far better for locating than current self-indexes. In theoretical terms, assuming for example the use of an alphabet-friendly FM-index [2] for counting, our index needs $O(H_k\log(1/H_k)n\log n+n)$ bits of space, counts in time $O(m(1+\frac{\log\sigma}{\log\log n}))$ and locates the *occ* occurrences of P in time $O(occ+\log n)$.

A second and simpler way to use the structure is, together with the plain text, as a replacement of the classical suffix array. In this case we must not only use it for locating the occurrences but also for binary searching. The binary search can be done over the samples first and then decompress the area between two consecutive samples to finish the search. This yields a very practical alternative requiring 0.8–2.4 times the text size (as opposed to 4) plus the text.

On the ther hand, if the text is very large, even a compressed index must reside on disk. Performing well on secondary memory with a compressed index has proved extremely difficult, because of their non-local access pattern. Thanks to its local decompression properties, our reduced suffix array performs very well on secondary memory. It needs the optimal $\lceil \frac{occ}{B} \rceil$ disk accesses for locating the *occ* occurrences, being B the disk block size measured in integers. On average, if the compression ratio (compressed divided by uncompressed suffix array size) is $0 \leq c \leq 1$, we perform $\lceil \frac{c \cdot occ}{B} \rceil$ accesses. That is, our index actually performs better, not worse (as it seems to be the norm), thanks to compression. We show how to upgrade this structure to an efficient secondary-memory self-index.

We experimentally explore the compression performance we achieve, the time for locating, and the simplified suffix array implementation, comparing against previous work. Our structure stands out as an excellent practical alternative.

2 Compressing the Suffix Array

Given a text $T = t_1 \ldots t_n$ over alphabet Σ of size σ, where for technical reasons we assume $t_n = \$$ is smaller than any other character in Σ and appears nowhere else in T, a *suffix array* $A[1,n]$ is a permutation of $[1,n]$ such that $T_{A[i],n} \prec T_{A[i+1],n}$ for all $1 \leq i < n$, being "$\prec$" the lexicographical order. By $T_{j,n}$ we denote the *suffix* of T that starts at position j. Since all the occurrences of a pattern $P = p_1 \ldots p_m$ in T are prefixes of some suffix, a couple of binary searches in A suffice to identify the segment in A of all the suffixes that start with P, that is, the segment pointing to all the occurrences of P. Thus the suffix array permits counting the occurrences of P in $O(m \log n)$ time and reporting the *occ* occurrences in $O(occ)$ time. With an additional array of integers, the counting time can be reduced to $O(m + \log n)$ [10].

Suffix arrays turn out to be compressible whenever T is. The k-th order empirical entropy of T, H_k [11], shows up in A in the form of large segments $A[i, i+\ell]$ that appear elsewhere in $A[j, j+\ell]$ with all the values shifted by one position, $A[j+s] = A[i+s]+1$ for $0 \leq s \leq \ell$. Actually, one can partition A into *runs* of maximal segments that appear repeated (shifted by 1) elsewhere, and the number of such runs is at most $nH_k + \sigma^k$ for any k [8,13].

This property has been used several times in the past to compress A. Mäkinen's Compact Suffix Array (CSA) [7] replaces runs with pointers to their definition elsewhere in A, so that the run can be recovered by (recursively) expanding the definition and shifting the values. Mäkinen and Navarro [8] use the connection with FM-indexes (runs in A are related to equal-letter runs in the Burrows-Wheeler transform of T, basic building block of FM-indexes) and run-length compression. Yet, the most successful technique to take advantage of those regularities has been the definition of function $\Psi(i) = A^{-1}[A[i]+1]$ (or

$A^{-1}[1]$ if $A[i] = n$). It can be seen that $\Psi(i) = \Psi(i-1) + 1$ within runs of A, and therefore a differential encoding of Ψ is highly compressible [14].

We present a completely different method to compress A. We first represent A in differential form: $A'[1] = A[1]$ and $A'[i] = A[i] - A[i-1]$ if $i > 1$. Take now a run of A of the form $A[j+s] = A[i+s] + 1$ for $0 \leq s \leq \ell$. It is easy to see that $A'[j+s] = A'[i+s]$ for $1 \leq s \leq \ell$. We have converted the runs of A into true repetitions in A'.

The next step is to take advantage of those repetitions in a way that permits fast local decompression of A'. We resort to Re-Pair [6], a dictionary-based compression method based on the following algorithm: (1) identify the most frequent pair $A'[i]A'[i+1]$ in A', let ab be such pair; (2) create a new integer symbol $s \geq n$ larger than all existing symbols in A' and add rule $s \rightarrow ab$ to a dictionary; (3) replace every occurrence of ab in A by s[2]; (4) iterate until every pair has frequency 1. The result of the compression is the table of rules (call it R) plus the sequence of (original and new) symbols into which A' has been compressed (call it C). Note that R can be easily stored as a vector of pairs, so that rule $s \rightarrow ab$ is represented by $R[s-n+1] = a : b$.

Any portion of C can be easily decompressed in optimal time and fast in practice. To decompress $C[i]$, we first check if $C[i] < n$. If it is, then it is an original symbol of A' and we are done. Otherwise, we obtain both symbols from $R[C[i] - n + 1]$, and expand them recursively (they can in turn be original or created symbols, and so on). We reproduce u cells of A' in $O(u)$ time, and the accesses pattern is local if R is small.

Since R grows by 2 integers (a, b) for every new pair, we can stop creating pairs when the most frequent one appears only twice. R can be further reduced by preempting this process, which trades its size for overall compression ratio.

A few more structures are necessary to recover the values of A: (1) a sampling of absolute values of A at regular intervals l; (2) a bitmap $L[1, n]$ marking the positions where each symbol of C (which could represent several symbols of A') starts in A'; (3) $o(n)$ further bits to answer *rank* queries on L in constant time [5,13]: $rank(L, i)$ is the number of 1's in $L[1, i]$. Thus, to retrieve $A[i, j]$ we: (1) see if there is a multiple of l in $[i, j]$, extending i to the left or j to the right to include such a multiple if necessary; (2) make sure we expand an integral number of symbols in C, extending i to the left and j to the right until $L[i] = 1$ and $L[j+1] = 1$; (3) use the mechanism described above to obtain $A'[i, j]$ by expanding $C[rank(L, i), rank(L, j)]$; (4) use any absolute sample of A included in $[i, j]$ to obtain, using the differences in $A'[i, j]$, the values $A[i, j]$; (5) return the values in the original interval $[i, j]$ requested.

The overall time complexity of this decompression is the output size plus what we have expanded the interval to include a multiple of l (i.e., $O(l)$) and to ensure an integral number of symbols in C. The latter can be controlled by limiting the length of the uncompressed version of the symbols we create.

[2] If $a = b$ it might be impossible to replace all occurrences, e.g. aa in aaa, but in such case one can at least replace each other occurrence in a row.

2.1 Faster Compression

A weak point in our scheme is compression speed. Re-Pair can be implemented in $O(n)$ time, but needs too much space [6]. We have used instead an $O(n \log n)$ time algorithm that requires less memory. We omit the details for lack of space.

We note that Ψ (which is easily built in $O(n)$ time from A) can be used to obtain a much faster compression algorithm, which in practice compresses only slightly less than the original Re-Pair. Recall that $\Psi(i)$ tells where in A is the value $A[i]+1$. The idea is that, if $A[i, i+\ell]$ is a run such that $A[j+s] = A[i+s]+1$ for $0 \leq s \leq \ell$ (and thus $A'[j+s] = A'[i+s]$ for $1 \leq s \leq \ell$), then $\Psi(i+s) = j+s$ for $0 \leq s \leq \ell$. Thus, by following permutation Ψ we have a good chance of finding repeated pairs in A' (although, as explained, Re-Pair does a slightly better job).

The algorithm is thus as follows. Let $i_1 = A^{-1}[1]$. We start at $i = i_1$ and see if $A'[i]A'[i+1] = A'[\Psi(i)]A'[\Psi(i)+1]$. If this does not hold, we move on to $i \leftarrow \Psi(i)$ and iterate. If the equality holds, we start a chain of replacements: We add a new pair $A'[i]A'[i+1]$ to R, make the replacements at i and $\Psi(i)$ and move on with $i \leftarrow \Psi(i)$, replacing until the pair changes. When the pair changes, that is $A'[i]A'[i+1] \neq A'[\Psi(i)]A'[\Psi(i)+1]$, we restart the process with $i \leftarrow \Psi(i)$, looking again for a new pair to create. When we traverse the whole A' without finding any pair to replace, we are done. With some care (omitted for lack of space) this algorithm runs in $O(n)$ time.

2.2 Analysis

We analyze the compression ratio of our data structure. Let N be the number of runs in Ψ. As shown in [8,13], $N \leq H_k n + \sigma^k$ for any $k \geq 0$. Except for the first cell of each run, we have that $A'[i] = A'[\Psi(i)]$ within the run. Thus, we cut off the first cell of each run, to obtain up to $2N$ runs now. Every pair $A'[i]A'[i+1]$ contained in such runs must be equal to $A'[\Psi(i)]A'[\Psi(i)+1]$, thus the only pairs of cells $A'[i]A'[i+1]$ that are not equal to the "next" pair are those where i is the last cell of its run. This shows that there are at most $2N$ different pairs in A', and thus the most frequent pair appears at least $\frac{n}{2N}$ times. Because of overlaps, it could be that only each other occurrence can be replaced, thus the total number of replacements in the first iteration is at least βn, for $\beta = \frac{1}{4N}$.

After we choose and replace the most frequent pair, we end up with at most $n - \beta n$ integers in A'. The number of runs has not varied, because a replacement cannot split a run. Thus, the same argument shows that the second time we remove at least $\beta(n - \beta n) = \beta n(1-\beta)$ cells. The third replacement removes at least $\beta(n - \beta n - \beta n(1-\beta)) = \beta n(1-\beta)^2$ cells. It is easy to see by induction that the i-th iteration removes $\beta n(1-\beta)^{i-1}$ cells.

After M iterations we have removed $\sum_{i=1}^{M} \beta n(1-\beta)^{i-1} = n - n(1-\beta)^M$ cells, and hence the length of C is $n(1-\beta)^M$ and the length of R is $2M$. The total size is optimized for $M^* = \frac{\ln n + \ln \ln \frac{1}{1-\beta} - \ln 2}{\ln \frac{1}{1-\beta}}$, where it is $\frac{2(\ln n + \ln \ln \frac{1}{1-\beta} - \ln 2 + 1)}{\ln \frac{1}{1-\beta}}$. Since $\ln \frac{1}{1-\beta} = \ln \frac{4N}{4N-1} = \frac{1}{4N}(1 + O(\frac{1}{N}))$, the total size is $8N \ln \frac{n}{4N} + O(N)$ integers.

Since $N \leq H_k n + \sigma^k$, if we stick to $k \leq \alpha \log_\sigma n$ for any constant $0 < \alpha < 1$, it holds $\sigma^k = O(n^\alpha)$ and the total space is $O(H_k \log \frac{1}{H_k} n \log n) + o(n)$ bits, as even after the M^* replacements the numbers need $O(\log n)$ bits.

Theorem 1. *Our structure representing A' using R and C needs $O(H_k \log \frac{1}{H_k} n \log n) + o(n)$ bits, for any $k \leq \alpha \log_\sigma n$ and any constant $0 < \alpha < 1$.*

As a comparison, Mäkinen's CSA [7] needs $O(H_k n \log n)$ bits [13], which is always better as a function of H_k. Yet, both tend to the same space as H_k goes to zero. Other self-indexes are usually smaller.

We can also show that the simplified replacement method of Section 2.1 reaches the same asymptotic space complexity (proof omitted for lack of space).

2.3 Compressing the Dictionary

We now develop some techniques to reduce the dictionary of rules R without affecting C. Those can be of independent interest to improve Re-Pair in general.

A first observation is that, if we have a rule $s \rightarrow ab$ and s is only mentioned in another rule $s' \rightarrow sc$, then we could perfectly remove rule $s \rightarrow ab$ and rewrite $s' \rightarrow abc$. This gives a net gain of one integer, but now we have rules of varying length. This is easy to manage, but we prefer to go further. We develop a technique that permits eliminating every rule definition that is used within R, once or more, and gain one integer for each rule eliminated. The key idea is to write down explicitly the binary tree formed by expanding the definitions (by doing a preorder traversal and writing 1 for internal nodes and 0 for leaves), so that not only the largest symbol (tree root) can be referenced later, but also any subtree.

For example, assume the rules $R = \{s \rightarrow ab,\ t \rightarrow sc,\ u \rightarrow ts\}$, and $C = tub$. We could first represent the rules by the bitmap $R_B = \texttt{100100100}$ (where s corresponds to position 1, t to 4, and u to 7) and the sequence $R_S = ab1c41$ (we are using letters for the original symbols of A', and the bitmap positions as the identifiers of created symbols[3]). We express C as $47b$. To expand, say, 4, we go to position 4 in R_B and compute $rank_0(R_B, 4) = 2$ (number of zeros up to position 4, $rank_0(i) = i - rank(i)$). Thus the corresponding symbols in R_S start at position 3. We extract one new symbol from R_S for each new zero we traverse in R_B, and stop when the number of zeros traversed exceeds the number of ones (this means we have completed the subtree traversal). This way we obtain the definition $1c$ for symbol 4.

Let us now reduce the dictionary by expanding the definition of s within t (even when s is used elsewhere). The new bitmap is $R_B = \texttt{11000100}$ (where $t = 1$, $s = 2$, and $u = 6$), the sequence is $R_S = abc12$, and $C = 16b$. We can now remove the definition of t by expanding it within u. This produces the new bitmap $R_B = \texttt{1110000}$ (where $u = 1$, $t = 2$, $s = 3$), the sequence $R_S = abc3$ and $C = 21b$. Further reduction is not possible because u's definition is only used

[3] In practice letters are numbers up to $n-1$ and the bitmap positions are distinguished by adding them $n-1$.

from C[4]. At the cost of storing at most $2|R|$ bits, we can reduce R by one integer for each definition that is used at least once within R.

The reduction can be easily implemented in linear time, avoiding the successive renamings of the example. We first count how many times each rule is used within R. Then we traverse R and only write down (the bits of R_B and the sequence R_S for) the entries with zero count. We recursively expand those entries, appending the resulting tree structure to R_B and leaf identifiers to R_S. Whenever we find a created symbol that does not yet have an identifier, we give it as identifier the current position in R_B and recursively expand it. Otherwise the expansion finishes and we write down a leaf (a "0") in R_B and the identifier in R_S. Then we rewrite C using the renamed identifiers.

3 Towards a Text Index

As explained in the Introduction, the reduced suffix array is not by itself a text index. We explore now different alternatives to upgrade it to full-text index.

3.1 A Main Memory Self-index

One possible choice is to add one of the many self-indexes able of counting the occurrences of P in little space [1,2,14,4]. Those indexes actually find out the area $[i, j]$ where the occurrences of P lie in A. Then locating the occurrences boils down to decompressing $A[i, j]$ from our structure.

To fix ideas, consider the alphabet-friendly FM-index [2]. It takes $nH_k + o(n \log \sigma)$ bits of space for any $k \le \alpha \log_\sigma n$ and constant $0 < \alpha < 1$, and can count in time $O(m(1 + \frac{\log \sigma}{\log \log n}))$. Our additional structure dominates the space complexity, requiring $O(H_k \log(1/H_k) n \log n) + o(n)$ bits for the representation of A'. To this we must add $O((n/l) \log n)$ bits for the absolute samples, and the extra cost to limit the formation of symbols that represent very long sequences. If we limit such lengths to l as well, we have an overhead of $O((n/l) \log n)$ bits, as this can be regarded as inserting a spurious symbol every l positions in A' to prevent the formation of longer symbols. By choosing $l = \log n$ we have $O(H_k \log(1/H_k) n \log n + n)$ bits of space, and time $O(occ + \log n)$ for locating the occurrences. Other tradeoffs are possible, for example having $n \log^{1-\varepsilon} n$ bits of extra space and $O(occ + \log^\varepsilon n)$ time, for any $0 < \varepsilon < 1$.

Extracting substrings can be done with the same FM-index, but the time to display ℓ text characters is, using $n \log^{1-\varepsilon} n$ additional bits of space, $O((\ell + \log^\varepsilon n)(1 + \frac{\log \sigma}{\log \log n}))$. By using the structure proposed in [3] we have other $nH_k + o(n \log \sigma)$ bits of space for $k = o(\log_\sigma n)$ (this space is asymptotically negligible) and can extract the characters in optimal time $O(1 + \frac{\ell}{\log_\sigma n})$.

Theorem 2. *There exists a self-index for text T of length n over an alphabet of size σ and k-th order entropy H_k, which requires $O(H_k \log(1/H_k) n \log n +$*

[4] It is tempting to replace u in C, as it appears only once, but our example is artificial: A symbol that is not mentioned in R must appear at least twice in C.

$n \log^{1-\varepsilon} n) + o(n \log \sigma)$ bits of space, for any $0 \le \varepsilon \le 1$. It can count the occurrences of a pattern of length m in time $O(m(1 + \frac{\log \sigma}{\log \log n}))$ and locate its occ occurrences in time $O(occ + \log^{\varepsilon} n)$. For $k = o(\log_\sigma n)$ it can display any text substring of length ℓ in time $O(1 + \frac{\ell}{\log_\sigma n})$. For larger $k \le \alpha \log_\sigma n$, for any constant $0 < \alpha < 1$, this time becomes $O((\ell + \log^{\varepsilon} n)(1 + \frac{\log \sigma}{\log \log n}))$.

3.2 A Smaller Classical Index

A simple and practical alternative is to use our reduced suffix array just like the classical suffix array, that is, not only for locating but also for searching, keeping the text in uncompressed form as well. This is not anymore a compressed index, but a practical alternative to a classical index.

The binary search of the interval that corresponds to P will start over the absolute samples of our data structure. Only when we have identified the interval between consecutive samples of A where the binary search must continue, we decompress the whole interval and finish the binary search. If the two binary searches finish in different intervals, we will also need to decompress the intervals in between for locating all the occurrences. For displaying, the text is at hand.

The cost of this search is $O(m \log n)$ plus the time needed to decompress the portion of A between two absolute samples. We can easily force the compressor to make sure that no symbol in C spans the limit between two such intervals, so that the complexity of this decompression can be controlled with the sampling rate l. For example, $l = O(\log n)$ guarantees a total search time of $O(m \log n + occ)$, just as the suffix array version that requires 4 times the text size (plus text).

Theorem 3. *There exists a full-text index for text T of length n over an alphabet of size σ and k-th order entropy H_k, which requires $O(H_k \log(1/H_k) n \log n + n)$ bits of space in addition to T, for any $k \le \alpha \log_\sigma n$ and any constant $0 < \alpha < 1$. It can count the occurrences of a pattern of length m in time $O(m \log n)$ and locate its occ occurrences in time $O(occ + \log n)$.*

3.3 A Secondary Memory Index

In [9], an index of size $nH_0 + O(n \log \log \sigma)$ bits is described, which can identify the area of A containing the occurrences of a pattern of length m (and thus count its occurrences) using at most $2m(1 + \lceil \log_B n \rceil)$ accesses to disk, where $B \log n$ is the number of bits in a disk block. However, this index is extremely slow to locate the occurrences: each locate needs $O(\log^{\varepsilon} n)$ random accesses to disk, where in practice $\varepsilon = 1$. This is achieved by storing the inverse of Ψ [14].

If, instead, we keep only the data structures for counting, and use our reduced suffix array, we can obtain $\lceil \frac{occ}{B} \rceil$ accesses to report the *occ* occurrences, which is worst-case optimal. Assume table R is small enough to fit in main memory (recall we can always force so, losing some compression). Then, we read the corresponding area of C from disk, and uncompress each cell in memory without any further disk access (the area of C to read can be obtained from an in-memory binary search over an array storing the absolute position of the first C cell of each disk block). On average, if we achieved compression ratio $c \leq 1$, we will need to read $c \cdot occ$ cells from C, at a cost of $\lceil \frac{c \cdot occ}{B} \rceil$. Therefore, we achieve for the first time a locating complexity that is *better* thanks to compression, not worse. Note that Mäkinen's CSA would not perform well at all under this scenario, as the decompression process is highly non-local.

To extract text passages of length ℓ we could use compressed sequence mechanisms like [3], which easily adapt to disk and have local decompression.

4 Experimental Results

We present three series of experiments in this section. The first one regards compression performance, the second the use of our technique as a plug-in for boosting the locating performance of a self-index, and the third the use of our technique as a classical index using reduced space. We use text collections obtained from the *PizzaChili* site, `http://pizzachili.dcc.uchile.cl`.

Compression performance. In Section 2.1 we mentioned that compression time of our scheme would be an issue and gave an approximate method based on Ψ which should be faster. Table 1 compares the performance of the exact Re-Pair compression algorithm (RP) and that of the Ψ-based approximation (RPΨ). We take absolute samples each 32 positions.

Table 1. Index size and build time using Re-Pair (RP) and its Ψ-based approximation (RPΨ). For the xml case, we also include a Re-Pair version (RPC) with rules up to length 256. Compression ratio compares with the $4n$ bytes needed by a suffix array.

Collection, size	Method	Index Size	Compr.	Re-Pai	Expected	Dict.	Main	Compr. with
(MB), H_3/H_0		(MB)	Ratio	Time (s)	decompr.	compr.	memory	5% in RAM
xml, 100,	RP	94.04	23.51%	25986	6939.99	57%	49%	34.29%
26.28%	RPΨ	102.76	25.69%	260	7570.49	57%	51%	81.85%
	RPC	99.82	24.96%	25129	134.99	58%	47%	35.86%
dna, 100,	RP	333.96	83.55%	11150	5.01	79%	19%	95.52%
97.02%	RPΨ	339.45	84.86%	546	4.73	78%	20%	101.4%
english, 100,	RP	221.31	55.33%	93421	238.31	59%	43%	87.98%
53.05%	RPΨ	241.33	60.33%	485	202.79	60%	44%	99.33%
pitches, 50,	RP	115.54	57.77%	15371	33.71	70%	21%	67.54%
61.37%	RPΨ	124.32	62.16%	180	26.78	67%	25%	85.36%
proteins, 100,	RP	286.66	71.67%	3143	58.97	80%	10%	79.58%
97.21%	RPΨ	295.15	73.78%	641	52.52	75%	13%	91.83%
sources, 100,	RP	151.81	37.95%	106173	2046.80	58%	48%	64.03%
40.74%	RPΨ	176.15	44.04%	377	1778.79	58%	50%	95.67%

The approximation runs 5 to 280 times faster and just loses 1%–14% in compression ratio. RP runs at 3 to 100 sec/MB, whereas RPΨ needs 0.26 to 0.65 sec/MB. Most of the indexing time is spent this compression; the rest adds up around 120 sec overall in all cases.

Compression ratio varies widely. On XML data we achieve 23.5% compression (the reduced suffix array is smaller than the text!), whereas compression is extremely poor on DNA. In many text types of interest we slash the suffix array to around half of its size. Below the name of each collection we wrote the percentage H_3/H_0, which gives an idea of the compressibility of the collection independent of its alphabet size (e.g. it is very easy to compress DNA to 25% because there are mainly 4 symbols but one chooses to spend a byte for each in the uncompressed text, otherwise DNA is almost incompressible).

Other statistics are available. In column 6 we measure the average length of a cell of C if we choose uniformly in A (longer cells are in addition more likely to be chosen for decompression). Those numbers explain the times obtained for the next series of experiments. Note that they are related to compressibility, but not as much as one could expect. Rather, the numbers obey to a more detailed structure of the suffix array: they are higher when the compression is not uniform across the array. In those cases, we can limit the maximum length of a C cell. To show how this impacts compression ratio and decompression speed, we include a so-called RPC method for xml (which has the largest C lengths). RPC forbids a rule to cross a 256-cell boundary. We can see that compression ratio is almost the same, worsening by 6.17% on xml (and less on others, not shown).

In column 7 we show the compression ratio achieved with the technique of Section 2.3, charging it the bitmap introduced as well. It can be seen that the technique is rather effective. Column 8 shows the percentage of the compressed structure (i.e., the compressed version of R) that should stay in RAM in order to be able to access C and the samples in secondary memory, as advocated in Section 3.3. Note that the percentage is not negligible when compression is good, and that 100 minus the percentage almost gives the percentage taken by C. The last column shows how much compression we would achieve if the structures that must reside on RAM were limited to 5% of the original suffix array size (this is measured before dictionary compression, so it would be around 3% after compression). We still obtain attractive compression performance on texts like XML, sources and pitches (recall that on secondary memory the compression ratio translates almost directly to decompression performance). As expected, RPΨ does a much poorer job here, as it does not choose the best pairs early.

A plugin for self-indexes. Section 3.1 considers using our reduced suffix array as a plugin to provide fast locate on existing self-indexes. In this experiment we plug our structure to the counting structures of the alphabet-friendly FM-index (AFI [2]), and compare the result against the original AFI, the Sadakane's CSA [14] and the SSA [2,8], all from *PizzaChili*. We increased the sampling rate of the locating structures of AFI, CSA and SSA, to match the size of our index (RPT). To save space we exclude DNA and pitches.

Fig. 1 shows the results. The experiment consists in choosing random ranges of the suffix array and obtaining the values. This simulates a locating query where we can control the amount of occurrences to locate. Our reduced suffix array has a constant time overhead (which is related to column 6 in Table 1 and the sample rate of absolute values) and from then on the cost per cell located is very low. As a consequence, it crosses sooner or later all the other indexes. For example, it becomes the fastest on XML after locating 4,000 occurrences, but it

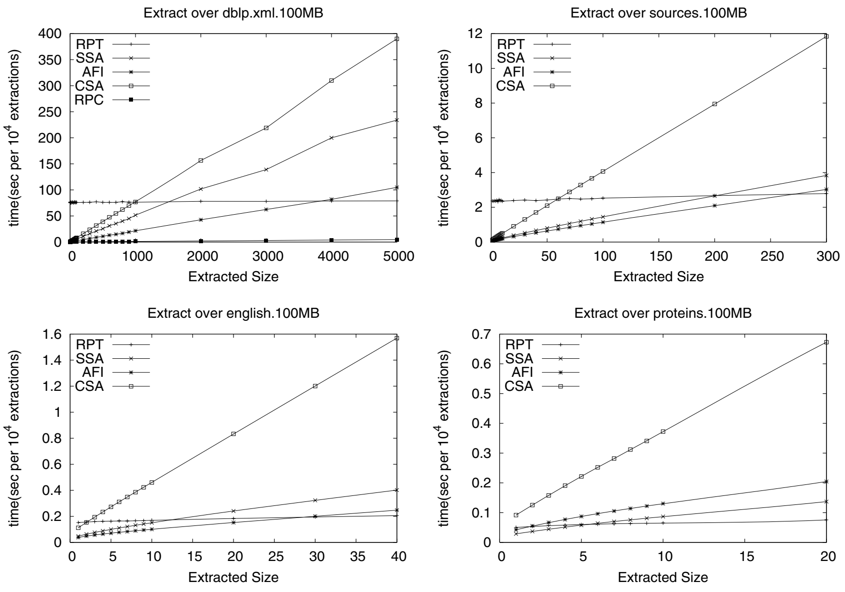


Fig. 1. Time to locate occurrences, as a function of the number of occurrences to locate. On xml, RPC becomes the fastest when extracting more than 2 results.

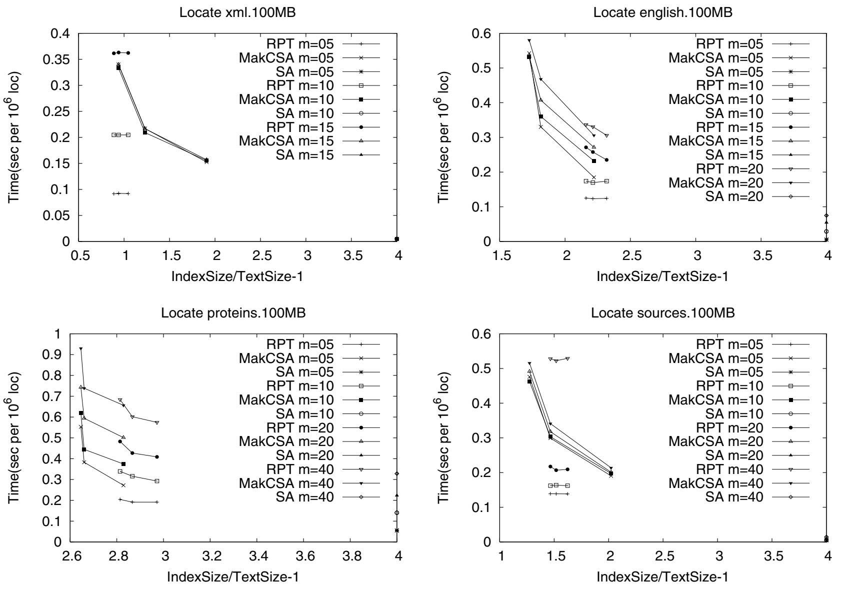


Fig. 2. Simulating a classical suffix array to binary search and locate the occurrences

needs just 6 occurrences to become the fastest on proteins. However, the RPC version shows an impressive (more than 500-fold) improvement on the cost per cell, standing as an excellent alternative when compression is so good.

A classical reduced index. Finally, we test our reduced suffix array as a replacement of the suffix array, that is, adding it the text and using it for binary searching, as explained in Section 3.2. We compare it with a plain suffix array (SA) and Mäkinen's CSA (MakCSA [7]), as the latter operates similarly.

Fig. 2 shows the result. The CSA offers space-time tradeoffs, whereas those of our index (sample rate for absolute values) did not significantly affect the time. Our structure stands out as a relevant space/time tradeoffs, especially when locating many occurrences (i.e. on short patterns).

References

1. Ferragina, P., Manzini, G.: Indexing compressed texts. J. of the ACM 52(4), 552–581 (2005)
2. Ferragina, P., Manzini, G., Mäkinen, V., Navarro, G.: Compressed representation of sequences and full-text indexes. ACM Transactions on Algorithms, 2006. TR 2004-05, Technische Fakultät, Univ. Bielefeld, Germany (to appear)
3. González, R., Navarro, G.: Statistical encoding of succinct data structures. In: Lewenstein, M., Valiente, G. (eds.) CPM 2006. LNCS, vol. 4009, pp. 295–306. Springer, Heidelberg (2006)
4. Grossi, R., Gupta, A., Vitter, J.: High-order entropy-compressed text indexes. In: Proc. 14th SODA, pp. 841–850 (2003)
5. Jacobson, G.: Space-efficient static trees and graphs. In: Proc. 30th FOCS, pp. 549–554 (1989)
6. Larsson, J., Moffat, A.: Off-line dictionary-based compression. Proc. IEEE 88(11), 1722–1732 (2000)
7. Mäkinen, V.: Compact suffix array — a space-efficient full-text index. Fundamenta Informaticae 56(1–2), 191–210 (2003)
8. Mäkinen, V., Navarro, G.: Succinct suffix arrays based on run-length encoding. Nordic J. of Computing 12(1), 40–66 (2005)
9. Mäkinen, V., Navarro, G., Sadakane, K.: Advantages of backward searching — efficient secondary memory and distributed implementation of compressed suffix arrays. In: Fleischer, R., Trippen, G. (eds.) ISAAC 2004. LNCS, vol. 3341, pp. 681–692. Springer, Heidelberg (2004)
10. Manber, U., Myers, G.: Suffix arrays: a new method for on-line string searches. SIAM J. Computing 22(5), 935–948 (1993)
11. Manzini, G.: An analysis of the Burrows-Wheeler transform. J. of the ACM 48(3), 407–430 (2001)
12. Navarro, G.: Indexing text using the Ziv-Lempel trie. J. of Discrete Algorithms 2(1), 87–114 (2004)
13. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Computing Surveys (to appear)
14. Sadakane, K.: New text indexing functionalities of the compressed suffix arrays. J. of Algorithms 48(2), 294–313 (2003)
15. Weiner, P.: Linear pattern matching algorithm. In: Proc. 14th IEEE Symp. on Switching and Automata Theory, pp. 1–11 (1973)

Processing Compressed Texts: A Tractability Border[⋆]

Yury Lifshits

Steklov Institute of Mathematics at St.Petersburg, Russia
yura@logic.pdmi.ras.ru

Abstract. What kind of operations can we perform effectively (without full unpacking) with compressed texts? In this paper we consider three fundamental problems: (1) check the equality of two compressed texts, (2) check whether one compressed text is a substring of another compressed text, and (3) compute the number of different symbols (Hamming distance) between two compressed texts of the same length.

We present an algorithm that solves the first problem in $O(n^3)$ time and the second problem in $O(n^2m)$ time. Here n is the size of compressed representation (we consider representations by straight-line programs) of the text and m is the size of compressed representation of the pattern. Next, we prove that the third problem is actually #P-complete. Thus, we indicate a pair of similar problems (equivalence checking, Hamming distance computation) that have radically different complexity on compressed texts. Our algorithmic technique used for problems (1) and (2) helps for computing minimal periods and covers of compressed texts.

1 Introduction

How can one minimize data storage space, without compromising too much on the query processing time? Here we address this problem using data compression perspective. Namely, what kind of problems can be solved in time polynomially depending on the size of *compressed* representation of texts?

Algorithms on compressed texts have applications in various areas of theoretical computer science. They were used for solving word equations in polynomial space [23]; for solving program equivalence within some specific class in polynomial time [14]; for verification of message sequence charts [9]. Fast search in compressed texts is also important for practical problems. Compression for indices of search engines is critical for web, media search, bioinformatics databases. Next, processing compressed objects has close relation to software/hardware verification. Usual verification task is to check some safety property on all possible system states. However, number of such states is so large that can not be verified by brute force approach. The only way is to store and process all states in some implicit (compressed) form.

⋆ Support by grants INTAS 04-77-7173 and NSh-8464.2006.1 is gratefully acknowledged.

Problem. *Straight-line program* (SLP) is now a widely accepted abstract model of compressed text. Actually it is just a specific class of context-free grammars that generate exactly one string. Rytter showed [24] that resulting encodings of most classical compression methods (LZ-family, RLE, dictionary methods) can be quickly translated to SLP. We give all details on SLPs in Section 2. Recently, an interesting variation of SLP was presented under the name of *collage systems* [13]. For any text problem on SLP-generated strings we ask two following questions: (1) Does a polynomial algorithm exist? (2) If yes, what is exact complexity of the problem? We can think about negative answer to the first question (say, NP-hardness) as an evidence that naive "generate-and-solve" is the best way for that particular problem.

Consider complexity of pattern matching problem on SLP-generated strings (sometimes called fully compressed pattern matching or FCPM). That is, given a SLPs generating a pattern P and a text T answer whether P is a substring of T and provide a succinct description of all occurrences. An important special case is *equivalence problem* for SLP-generated texts. Then we generalize compressed equivalence problem to compressed Hamming distance problem. Namely, given two SLP-generated texts of the same length, compute the number of positions where original texts differ from each other. Equality checking of compressed texts is a natural seems to be a natural problem related to checking changes in backup systems. Fully compressed pattern matching can be used in software verification and media search (audio/video pattern might be quite large and also require compression). Hamming distance is a simple form of approximate matching which is widely used in bioinformatics as well as in media search.

Compressed equality problem was solved for the first time in the paper [22] in 1994 in $O(n^4)$ time. The first solution for fully compressed pattern matching appeared a year later in the paper [12]. Next, a polynomial algorithm for computing combinatorial properties of SLP-generated text was presented in [8]. Finally, in 1997 Miyazaki, Shinohara and Takeda [18] constructed new $O(n^2m^2)$ algorithm for FCPM, where m and n are the sizes of SLPs that generate P and T, correspondingly. In 2000 for one quite special class of SLP the FCPM problem was solved in time $O(mn)$ [10]. Nevertheless, nothing was known about complexity of compressed Hamming distance problem.

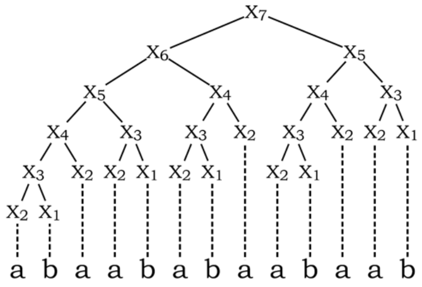
Our results. The key result of the paper is a new $O(n^2m)$ algorithm for pattern matching on SLP-generated texts. As before, m and n are sizes of SLPs generating P and T, correspondingly. This algorithm is not just an improvement over previous ones [8,10,12,18,22] but is also simpler than they are. Next, we prove #P-completeness of computing Hamming distance between compressed texts in Section 4. Recall that #P is a class of functions, a kind of extension for class of predicates NP. Here for the first time we have closely related problems (equivalence checking, Hamming distance computation) from different sides of the border between efficiently solvable problems on SLP-generated texts and intractable ones. Algorithmic technique from our main FCPM algorithm could be used for computing the shortest period/cover of compressed text. We show this application and state some questions for further research in Section 5.

Historical remarks. Algorithms for finding an explicitly given pattern in a compressed texts were the first results in the field [1,6]. Next, polynomial algorithms for regular expression matching [19], approximate pattern matching [11] and subsequence matching [5] were constructed for SLP-generated texts. Encouraging experimental results are reported in [20]. Some other problems turned out to be hard. Context-free language membership [17], two-dimensional pattern matching [4], fully compressed subsequence matching [16] are all at least NP-hard. The paper [25] surveys the field of processing compressed texts.

2 Compressed Strings Are Straight-Line Programs

A *Straight-line program* (SLP) is a context-free grammar generating exactly one string. Moreover, we allow only two types of productions: $X_i \to a$ and $X_i \to X_p X_q$ with $i > p, q$. The string represented by a given SLP is a unique text corresponding to the last nonterminal X_m. Although in previous papers X_i denotes only a nonterminal symbol while the corresponding text was denoted by $val(X_i)$ or $eval(X_i)$ we identify this notions and use X_i both as a nonterminal symbol and as the corresponding text. Hopefully, the right meaning is always clear from the context. We say that the size of SLP is equal to the number of productions.

Example. Consider string *abaababaabaab*. It could be generated by the following SLP: $X_7 \to X_6X_5,\ X_6 \to X_5X_4,\ X_5 \to X_4X_3,\ X_4 \to X_3X_2,\ X_3 \to X_2X_1,\ X_2 \to a,\ X_1 \to b.$



In fact, the notion of SLP describes only decompression operation. We do not care how such an SLP was obtained. Surprisingly, while the compression methods vary in many practical algorithms of Lempel-Ziv family and run-length encoding, the decompression goes in almost the same way. In 2003 Rytter [24] showed that given any LZ-encoding of string T we could efficiently get an SLP encoding for the same string which is at most $O(\log |T|)$ times longer than the original LZ-encoding. This translation allows us to construct algorithms only in the simplest SLP model. If we get a different encoding, we just translate it to SLP before applying our algorithm. Moreover, if we apply Rytter's translation to LZ77-encoding of a string T, then we get an $O(\log |T|)$-approximation of the *minimal* SLP generating T. The straight-line programs allow the exponential ratio between the size of SLP and the length of original text. For example $X_n \to X_{n-1}X_{n-1}, \dots X_2 \to X_1X_1, X_1 \to a$ has n rules and generates 2^{n-1}-long text.

We use both $\log |T|$ and n (number of rules in SLP) as parameters of algorithms' complexity. For example, we prefer $O(n \log |T|)$ bound to $O(n^2)$, since in practice the ratio between the size of SLP and the length of the text might be much smaller than exponential.

3 A New Algorithm for Fully Compressed Pattern Matching

Decision version of the fully compressed pattern matching problem (FCPM) is as follows:

INPUT: Two straight-line programs generating P and T
OUTPUT: Yes/No (whether P is a substring in T?)

Other variations are: to find the first occurrence, to count all occurrences, to check whether there is an occurrence from the given position and to compute a "compressed" representation of all occurrences. Our plan is to solve the last one, that is, to compute an auxiliary data structure that contains all necessary information for effective answering to the other questions.

We use a computational assumption which was implicitly used in all previous algorithms. In analysis of our algorithm we count arithmetical operations on positions in *original texts* as unit operations. In fact, text positions are integers with at most $\log |T|$ bits in binary form. Hence, in terms of bit operations the algorithm's complexity is larger than our $O(n^2 m)$ estimate up to some $\log |T|$-dependent factor.

Explanation of the algorithm goes in three steps. We introduce a special data structure (*AP-table*) and show how to solve pattern matching problem using this table in Subsection 3.1. Then we show how to compute AP-table using *local search* procedure in Subsection 3.2. Finally, we present an algorithm for local search in Subsection 3.3.

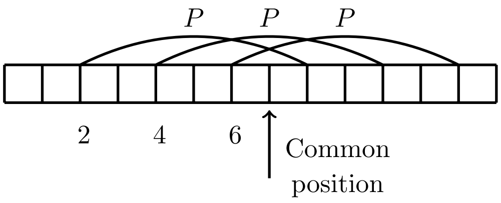
3.1 Pattern Matching Via Table of Arithmetical Progressions

We need some notation and terminology. We call a *position* in the text a point between two consequent letters. Hence, the text $a_1 \ldots a_n$ has positions $0, \ldots, n$ where first is in front of the first letter and the last one after the last letter. We say that some substring *touches* a given position if this position is either inside or on the border of that substring. We use the term *occurrence* both for a corresponding substring and for its starting position. Again, we hope that the right meaning is always clear from the context.

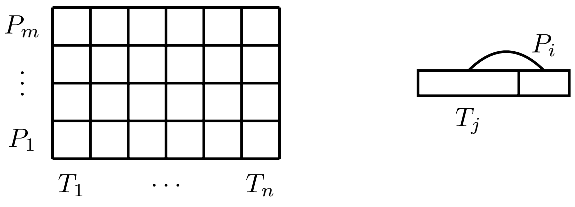
Let $P_1, \ldots, P_m$ and $T_1, \ldots, T_n$ be nonterminal symbols of SLPs generating P and T. For each of these texts we define a special *cut* position. It is a starting position for one-letter texts and merging position for $X_i = X_r X_s$. In the example above, the cut position for the intermediate text X_6 is between 5th and 6th letters: $abaab|aba$, since X_6 is obtained by concatenating $X_5 = abaab$ and $X_4 = aba$.

Our algorithm is based on the following theoretical fact (it was already used in [18], a similar technique was used also in [2]):

Lemma 1 (Basic Lemma). *All occurrences of P in T touching any given position form a single arithmetical progression (ar.pr.)*



The *AP-table* (table of arithmetical progressions) is defined as follows. For every $1 \leq i \leq m, 1 \leq j \leq n$ the value $AP[i,j]$ is a code of ar.pr. of occurrences of P_i in T_j that touch the cut of T_j. Note that any ar.pr. could be encoded by three integers: first position, difference, number of elements. If $|T_j| < |P_i|$ we define $AP[i,j] = \varnothing_1$, and if text is large enough but there are no occurrences we define $AP[i,j] = \varnothing_2$.



Claim 1: Using AP-table (actually, only top row is necessary) one can solve decision, count and checking versions of FCPM in time $O(n)$.

Claim 2: One can compute the whole AP-table by dynamic programming method in time $O(n^2m)$.

Proof of Claim 1. We get the answer for decision FCPM by the following rule: P occurs in T iff there is j such that $AP[m,j]$ is nonempty. Checking and counting are slightly more tricky. Recursive algorithm for checking: test whether the candidate occurrence touches the cut in the current text. If yes, use AP-table and check the membership in the corresponding ar.pr., otherwise call recursively this procedure either for the left or for the right part. We can inductively count the number of P-occurrences in all $T_1, \ldots, T_n$. To start, we just get the cardinality of the corresponding ar.pr. from AP-table. Inductive step: add results for the left part, for the right part and cardinality of central ar.pr. without "just-touching" occurrences.

3.2 Computing AP-Table

Sketch of the algorithm for computing AP-table:

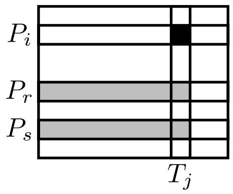
1. Preprocessing: compute lengths and cut positions for all intermediate texts;
2. Compute all rows and columns of AP-table that correspond to one-letter texts;
3. From the smallest pattern to the largest one, from the smallest text to the largest one consequently compute AP[i,j]:
   (a) Compute occurrences of the larger part of P_i in T_j around the cut of T_j;
   (b) Compute occurrences of the smaller part of P_i in T_j that start at ending positions of the larger part occurrences;
   (c) Intersect occurrences of smaller and larger part of P_i and merge all results to a single ar.pr.

Step 1: preprocessing. At the very beginning we inductively compute arrays of lengths, cut positions, first letter, last letter of the texts $P_1, \ldots, P_m, T_1, \ldots, T_n$ in time $O(n + m)$.

Step 2: computing AP-table elements in one-letter rows/columns. Case of $|P_i| = 1$: compare it with T_j if $|T_j| = 1$ or compare it with the last letter of the left part and the first one of the right part (we get this letters from precomputation stage). The resulting ar.pr. has at most two elements. Hence, just O(1) time used for computing every cell in the table. Case of $|T_j| = 1$: if $|P_i| > 1$ return $\varnothing_1$, else compare letters. Also $O(1)$ time is enough for every element.

Step 3: general routine for computing next element of AP-table. After one-letter rows and columns we fill AP-table in lexicographic order of pairs (i, j). Let $P_i = P_r P_s$. We use already obtained elements $AP[r, 1], \ldots, AP[r, j]$ and $AP[s, 1], \ldots, AP[s, j]$ for computing $AP[i, j]$. In other words, we only need information about occurrences of left/right part of the current pattern in the current and all previous texts.

Order of computation:
"grey" values are used for computing "black" one
(we assume $P_i = P_r P_s$)



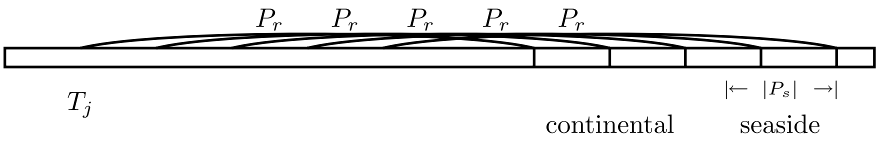
Let the cut position in T_j be γ (we get it from the preprocessing stage), and without loss of generality let $|P_r| \geq |P_s|$. The *intersection method* is (1) to compute all occurrences of P_r "around" cut of T_j, (2) to compute all occurrences of P_s "around" cut of T_j, and (3) shift the latter by $|P_r|$ and intersect.

Miyazaki et al. [18] construct a $O(mn)$ realization of intersection method. We use a different (more accurate) way and new technical tricks that require only $O(n)$ time for computing $AP[i, j]$. We use the same first step as in intersection method. But on the second one we look only for P_s *occurrences that start from* P_r *endings.*

We design a special auxiliary *local search* procedure that extracts useful information from already computed part of AP-table. Procedure $LocalSearch(i, j, [\alpha, \beta])$ returns occurrences of P_i in T_j inside the interval $[\alpha, \beta]$. Important properties: (1) Local search uses values AP[i,k] for $1 \leq k \leq j$, (2) It works properly only when $|\beta - \alpha| \leq 3|P_i|$, (3) It works in time $O(j)$, (4) The output of local search is a pair of ar.pr., all occurrences inside each ar.pr. have a common position, and all elements of the second are to the right of all elements of the first. We now show how to compute a new element using 5 local search calls.

Step 3a: finding occurrences of bigger part of the pattern. We apply local search for finding all occurrences of P_r in the interval $[\gamma - |P_i|, \gamma + |P_r|]$. Its length is $|P_i| + |P_r| \leq 3|P_r|$. As an answer we get two ar.pr. of all potential starts of P_r occurrences that touch the cut. Unfortunately, we are not able to do the same for P_s, since the length of interesting interval is not necessarily constant in terms of $|P_s|$. So we are going to find only occurrences of P_s that start from endings of two arithmetical progressions of P_r occurrences.

Step 3b: finding occurrences of smaller part of the pattern. We process each ar.pr. separately. We call an ending *continental* if it is at least $|P_s|$ far from the last ending in progression, otherwise we call it *seaside.*



Since we have an ar.pr. of P_r occurrences that have common position (property 4 of local search), all substrings of length $|P_s|$ starting from continental endings are identical. Hence, we need to check all seaside endings and *only one* continental position. For checking the seaside region we just apply local search for $|P_s|$-neighborhood of last endpoint and intersect the answer with ar.pr. of seaside ending positions. Intersecting two ar.pr. could be done in time $O(\log |T|)$. Indeed, the difference of resulting ar.pr. is equal to the least common multiple of differences of initial progressions. To find the first common point we should solve an equation of type $ax \equiv b \pmod{c}$. This kind of equations can be solved by technique similar to Euclid algorithm.

For checking continental region we apply local search for $|P_s|$ substring starting from the first continental ending.

Step 3c: simplifying answer to a single progression. Complete answer consists of all continental endings/or none of them, plus some sub-progression of seaside endings, plus something similar for the second ar.pr. Since all of these four parts are ar.pr. going one after another, we could simplify the answer to one ar.pr. (it must be one ar.pr. by Basic Lemma) in time $O(1)$.

Complexity analysis. We use one local search call for P_r, four local search calls for P_s, and twice compute the intersection of arithmetical progressions. Hence, we perform 7 steps of $O(n)$ complexity for computing a new element.

3.3 Realization of Local Search

Local search finds all P_i occurrences in the substring $T_j[\alpha,\beta]$. On the first step we run recursive **crawling procedure** with main parameters (i, j, α, β). After halting of all computation branches we get a sorted list of progressions representing P_i occurrences within $[\alpha,\beta]$ interval in T_j. Then we run **merging procedure** that simplifies all progressions to two ones.

Crawling procedure. Here we have three main parameters $(i, j, [\alpha,\beta])$ and two auxiliary ones: *global shift* and *pointer to output list.* Initially we start with main parameters of local search, zero shift and a pointer to empty list. In every call we take the ar.pr. of occurrences of P_i in T_j that touch the cut, leave only occurrences within the interval, and output this truncated ar.pr. (adding global shift) to the list (using current pointer). After that, we check whether the intersection of the interval $[\alpha,\beta]$ with left/right part of T_j is at least $|P_i|$ long. If so, we recursively call crawling procedure with the same i, the index of left/right part of T_j, and with this intersection interval. We also update global shift parameter for the right part and use new pointers to places just before/after inserted element.

Consider the set of all intervals we work with during the crawling procedure. Note that, by construction, any pair of them are either disjoint or embedded. Moreover, since the initial interval is at most $3|P_i|$, there are no four pairwise disjoint intervals in this set. If we consider a sequence of embedded intervals, then all intervals correspond to their own intermediate text from $T_1, \ldots, T_j$. Therefore, there were at most $3j$ recursive calls in crawling procedure and it works in time $O(j)$. At the end we get a sorted list of at most $3n$ arithmetical progressions. By "sorted" we mean that the last element of k-th progression is less than or equal to the first one of $k+1$-th progression. It follows from construction of crawling procedure, that output progressions could have only first/last elements in common.

Merging procedure. We go through the resulting list of progressions. Namely, we compare the distance between the last element of current progression and the first element of the next progression with the differences of these two progressions. If all three numbers are equal we merge the next progression with the current one. Otherwise we just announce a new progression. Applying Basic Lemma to $\delta_1 = \lfloor\frac{2\alpha+\beta}{3}\rfloor$ and $\delta_2 = \lfloor\frac{\alpha+2\beta}{3}\rfloor$ positions, we see that all occurrences of P_i in $[\alpha,\beta]$ interval form at most two (one after another) arithmetical progressions. Namely, those who touch δ_1 and those who don't touch but touch δ_2. Here we use that $\beta - \alpha \leq 3|P_i|$, and therefore any occurrence of P_i touches either δ_1 or δ_2. Hence, our merging procedure starts a new progression at most once.

3.4 Discussion on the Algorithm

Here we point out two possible improvements of the algorithm. Consider in details the "new element routine". Note that local search uses only $O(h)$ time, where h is the height of the SLP generating T, while intersection of arithmetical progressions uses even $O(\log|T|)$. Hence, if it is possible to "balance" any SLP up to $O(\log |T|)$ height, then the bound for working time of our algorithm becomes $O(nm \log |T|)$.

It is interesting to consider more rules for generating texts, since collage systems [13] and LZ77 [26] use concatenations and *truncations*. Indeed, as Rytter [24] showed, we could leave only concatenations expanding the archive just by factor $O(\log |T|)$. However, we hope that the presented technique works directly for the system of truncation/concatenation rules. We also claim that AP-table might be translated to a polynomial-sized SLP generating all occurrences of P in T.

4 Hardness Result

Hamming distance (denoted as $HD(S,T)$) between two strings of the same length is the number of characters which differ. *Compressed Hamming distance problem* (counting version): given two straight-line programs generating texts of the same length, compute Hamming distance between them.

A function belongs to class #P if there exists a nondeterministic Turing machine M such that the function value corresponding to input x is equal to the number of accepting branches of $M(x)$. In other words, there exists a polynomially-computable function $G(x,y)$ such that $f(x) = \#\{y | G(x,y) =$ "yes"$\}$. A function f has a [1]-Turing reduction to a function g, if there exist polynomially-computable functions E and D such that $f(x) = D(g(E(x)))$. We call a function to be #P-complete (under [1]-Turing reductions), if it belongs to the class #P and every other function from this class has a [1]-Turing reduction to it.

Theorem 1. *Compressed Hamming distance problem is #P-complete.*

Proof. Membership in #P. We can use a one-position-comparison as G function: $G(T,S;y) =$ "yes", if $T_y \neq S_y$. Then number of y giving answer "yes" is exactly equal to Hamming distance. Function G is polynomially computable. Indeed, knowing lengths of all intermediate texts we can walk through SLP decompression tree "from the top to the bottom" and compute value of T_y in linear time.

#P-hardness. It is enough to show a reduction from another complete problem. Recall the well-known #P-complete problem *subset sum* [7]: given integers $w_1, \dots, w_n, t$ in binary form, compute the number of sequences $x_1, \dots, x_n \in \{0,1\}$ such that $\sum_{i=1}^{n} x_i \cdot w_i = t$. In other words, how many subsets of $W = \{w_1, \dots, w_n\}$ have the sum of elements equal to t? We now construct a [1]-Turing reduction from subset sum to compressed Hamming distance. Let us fix input values for subset sum. We are going to construct two straight-line programs such that Hamming distance between texts generated by them can help to solve subset sum problem.

Our idea is the following. Let $s = w_1 + \cdots + w_n$. We construct two texts of length $(s+1)2^n$, describing them as a sequence of 2^n blocks of size $s+1$. The first text T is an encoding of t. All its blocks are the same. All symbols except one are "0", the only "1" is located at the $t+1$-th place. Blocks of the second text S correspond to all possible subsets of W. In every such block the only "1" is placed exactly after the place equal to the sum of elements of the corresponding subset. In a formal way, we can describe T and S by the following formulas:

$$T = (0^t 10^{s-t})^{2^n}, \quad S = \prod_{x=0}^{2^n-1} (0^{\bar{x}\cdot\bar{w}} 10^{s-\bar{x}\cdot\bar{w}}).$$

Here $\bar{x} \cdot \bar{w} = \sum x_i w_i$ and $\prod$ denotes concatenation.

The string S (let us call it *Lohrey string*) was used for the first time in the Markus Lohrey's paper [17], later it was reused in [16]. Lohrey proved in [17] that knowing input values for subset sum one can construct polynomial-size SLP that generates S and T in polynomial time. Notice that $HD(T, S)$ is exactly two times the number of subsets of W with elements' sum nonequal to t. Therefore, the subset sum answer can be computed as $2^n - \frac{1}{2}HD(T, S)$.

It turns out that #P-complete problems could be divided in subclasses that are not Karp-reducible to each other. E.g. recently [21] a new class TotP was presented. A function belongs to TotP, if there exists a nondeterministic machine M such that $f(x)$ is equal to the number of *all* branches of $M(x)$ minus one. Many problems with polynomially-easy "yes/no" version and #P-complete counting version belong to TotP.

We can show that compressed Hamming distance belongs to TotP. Indeed, we can test equality of substrings and split computation every time when *both* the left half and the right half of a substring in the first text are not equal to the corresponding left/right halves of the same interval in the second text. We should also add a dummy computing branch to every pair of nonequal compressed texts.

5 Consequences and Open Problems

A *period* of string T is a string W (and also an integer $|W|$) such that T is a prefix of W^k for some integer k. A *cover* (notion originated from [3]) of a string T is a string C such that any character in T is covered by some occurrence of C in T. Problem of *compressed periods/covers:* given a compressed string T, find the length of minimal period/cover and compute a "compressed" representation of all periods/covers.

Theorem 2. *Assume that AP-table can be computed in time $O((n+m)^k)$. Then compressed periods problem can be solved in time $O(n^k \log |T|)$, while compressed cover problem can be solved in time $O(n^k \log^2 |T|)$.*

Corollary. Our $O(n^2m)$ algorithm for AP-table provides $O(n^3 \log |T|)$ and $O(n^3$ $\log^2 |T|)$ complexity bounds for compressed periods and compressed covers, correspondingly.

For complete proof of Theorem 2 we refer to technical report version of this paper [15]. The compressed periods problem was introduced in 1996 in the extended abstract [8]. Unfortunately, the full version of the algorithm given in [8] (it works in $O(n^5 \log^3 |T|)$ time) was never published.

We conclude with some problems and questions for further research:

1. To speed up the presented $O(n^2 m)$ algorithm for fully compressed pattern matching. *Conjecture:* improvement to $O(nm \log |T|)$ is possible. More precisely, we believe that every element in AP-table can be computed in $O(\log |T|)$ time.
2. Is it possible to speed up computing of edit distance (Levenshtein distance) in the case when one text is highly compressible? Formally, is it possible to compute the edit distance in $O(nm)$ time, where n is the length of T_1, and m is the size of SLP generating T_2? This result leads to speedup of edit distance computing in case of "superlogarithmic" compression ratio. Recall that the classical algorithm has $O(\frac{n^2}{\log n})$ complexity.
3. Consider two SLP-generated texts. Is it possible to compute the length of the longest common substring for them in polynomial (from SLPs' size) time?

Compressed suffix tree. Does there exist a data structure for text representation such that (1) it allows pattern matching in time *linear to the pattern's length*, and (2) for some reasonable family of "regular" texts this structure requires less storing space than original text itself?

Application to verification. Algorithm for symbolic verification is one of the major results in model checking. It uses OBDD (ordered binary decision diagrams) representations for sets of states and transitions. It is easy to show that every OBDD representation can be translated to the SLP-representation of the same size. On the other hand there are sets for which SLP representation is logarithmically smaller. In order to replace OBDD representations by SLPs we have to answer the following question. Given two SLP-represented sets A and B, how to compute a close-to-minimal SLP representing $A \cap B$ and a close-to-minimal SLP representing $A \cup B$?

References

1. Amir, A., Benson, G., Farach, M.: Let sleeping files lie: Pattern matching in Z-compressed files. In: SODA'94 (1994)
2. Amir, A., Landau, G.M., Lewenstein, M., Sokol, D.: Dynamic text and static pattern matching. In: Dehne, F., Sack, J.-R., Smid, M. (eds.) WADS 2003. LNCS, vol. 2748, pp. 340–352. Springer, Heidelberg (2003)
3. Apostolico, A., Farach, M., Iliopoulos, C.S.: Optimal superprimitivity testing for strings. Inf. Process. Lett. 39(1), 17–20 (1991)
4. Berman, P., Karpinski, M., Larmore, L.L., Plandowski, W., Rytter, W.: On the complexity of pattern matching for highly compressed two-dimensional texts. Journal of Computer and Systems Science 65(2), 332–350 (2002)

5. Cegielski, P., Guessarian, I., Lifshits, Y., Matiyasevich, Y.: Window subsequence problems for compressed texts. In: Grigoriev, D., Harrison, J., Hirsch, E.A. (eds.) CSR 2006. LNCS, vol. 3967, Springer, Heidelberg (2006)
6. Farach, M., Thorup, M.: String matching in lempel-ziv compressed strings. In: STOC '95, pp. 703–712. ACM Press, New York (1995)
7. Garey, M., Johnson, D.: Computers and Intractability: a Guide to the Theory of NP-completeness. Freeman (1979)
8. Gąsieniec, L., Karpinski, M., Plandowski, W., Rytter, W.: Efficient algorithms for Lempel-Ziv encoding (extended abstract). In: Karlsson, R., Lingas, A. (eds.) SWAT 1996. LNCS, vol. 1097, pp. 392–403. Springer, Heidelberg (1996)
9. Genest, B., Muscholl, A.: Pattern matching and membership for hierarchical message sequence charts. In: Rajsbaum, S. (ed.) LATIN 2002. LNCS, vol. 2286, pp. 326–340. Springer, Heidelberg (2002)
10. Hirao, M., Shinohara, A., Takeda, M., Arikawa, S.: Fully compressed pattern matching algorithm for balanced straight-line programs. In: SPIRE'00, pp. 132–138. IEEE Computer Society Press, Los Alamitos (2000)
11. Kärkkäinen, J., Navarro, G., Ukkonen, E.: Approximate string matching over Ziv-Lempel compressed text. In: Giancarlo, R., Sankoff, D. (eds.) CPM 2000. LNCS, vol. 1848, pp. 195–209. Springer, Heidelberg (2000)
12. Karpinski, M., Rytter, W., Shinohara, A.: Pattern-matching for strings with short descriptions. In: Galil, Z., Ukkonen, E. (eds.) Combinatorial Pattern Matching. LNCS, vol. 937, pp. 205–214. Springer, Heidelberg (1995)
13. Kida, T., Matsumoto, T., Shibata, Y., Takeda, M., Shinohara, A., Arikawa, S.: Collage system: a unifying framework for compressed pattern matching. Theor. Comput. Sci. 298(1), 253–272 (2003)
14. Lasota, S., Rytter, W.: Faster algorithm for bisimulation equivalence of normed context-free processes. In: Královič, R., Urzyczyn, P. (eds.) MFCS 2006. LNCS, vol. 4162, pp. 646–657. Springer, Heidelberg (2006)
15. Lifshits, Y.: Algorithmic properties of compressed texts. Technical Report PDMI, 23/2006 (2006)
16. Lifshits, Y., Lohrey, M.: Quering and embedding compressed texts. In: Královič, R., Urzyczyn, P. (eds.) MFCS 2006. LNCS, vol. 4162, pp. 681–692. Springer, Heidelberg (2006)
17. Lohrey, M.: Word problems on compressed word. In: Díaz, J., Karhumäki, J., Lepistö, A., Sannella, D. (eds.) ICALP 2004. LNCS, vol. 3142, pp. 906–918. Springer, Heidelberg (2004)
18. Miyazaki, M., Shinohara, A., Takeda, M.: An improved pattern matching algorithm for strings in terms of straight line programs. In: Hein, J., Apostolico, A. (eds.) Combinatorial Pattern Matching. LNCS, vol. 1264, pp. 1–11. Springer, Heidelberg (1997)
19. Navarro, G.: Regular expression searching on compressed text. J. of Discrete Algorithms 1(5-6), 423–443 (2003)
20. Navarro, G., Raffinot, M.: A general practical approach to pattern matching over Ziv-Lempel compressed text. In: Crochemore, M., Paterson, M.S. (eds.) Combinatorial Pattern Matching. LNCS, vol. 1645, pp. 14–36. Springer, Heidelberg (1999)
21. Pagourtzis, A., Zachos, S.: The complexity of counting functions with easy decision version. In: Královič, R., Urzyczyn, P. (eds.) MFCS 2006. LNCS, vol. 4162, pp. 741–752. Springer, Heidelberg (2006)
22. Plandowski, W.: Testing equivalence of morphisms on context-free languages. In: van Leeuwen, J. (ed.) ESA 1994. LNCS, vol. 855, pp. 460–470. Springer, Heidelberg (1994)

23. Plandowski, W.: Satisfiability of word equations with constants is in PSPACE. J. ACM 51(3), 483–496 (2004)
24. Rytter, W.: Application of Lempel-Ziv factorization to the approximation of grammar-based compression. Theor. Comput. Sci. 302(1–3), 211–222 (2003)
25. Rytter, W.: Grammar compression, LZ-encodings, and string algorithms with implicit input. In: Díaz, J., Karhumäki, J., Lepistö, A., Sannella, D. (eds.) ICALP 2004. LNCS, vol. 3142, pp. 15–27. Springer, Heidelberg (2004)
26. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Transactions on Information Theory 23(3), 337–343 (1977)

Common Structured Patterns in Linear Graphs: Approximation and Combinatorics

Guillaume Fertin[1,⋆], Danny Hermelin[2,⋆⋆], Romeo Rizzi[3,⋆], and Stéphane Vialette[4,⋆]

[1] LINA, Univ. Nantes, 2 rue de la Houssinière, Nantes, France
Guillaume.Fertin@lina.univ-nantes.fr
[2] Dpt. Computer Science, Univ. Haifa, Mount Carmel, Haifa, Israel
danny@cri.haifa.ac.il
[3] DIMI, Univ. Udine, Udine, Italy
Romeo.Rizzi@dimi.uniud.it
[4] LRI, UMR 8623, Univ. Paris-Sud, Orsay, France
Stephane.Vialette@lri.fr

Abstract. A linear graph is a graph whose vertices are linearly ordered. This linear ordering allows pairs of disjoint edges to be either preceding ($<$), nesting ($\sqsubset$) or crossing ($\between$). Given a family of linear graphs, and a non-empty subset $\mathcal{R} \subseteq \{<, \sqsubset, \between\}$, we are interested in the MCSP problem: Find a maximum size edge-disjoint graph, with edge-pairs all comparable by one of the relations in $\mathcal{R}$, that occurs as a subgraph in each of the linear graphs of the family. In this paper, we generalize the framework of Davydov and Batzoglou by considering patterns comparable by all possible subsets $\mathcal{R} \subseteq \{<, \sqsubset, \between\}$. This is motivated by the fact that many biological applications require considering crossing structures, and by the fact that different combinations of the relations above give rise to different generalizations of natural combinatorial problems. Our results can be summarized as follows: We give tight hardness results for the MCSP problem for $\{<, \between\}$-structured patterns and $\{\sqsubset, \between\}$-structured patterns. Furthermore, we prove that the problem is approximable within ratios: (*i*) $2\mathscr{H}(k)$ for $\{<, \between\}$-structured patterns, (*ii*) $k^{1/2}$ for $\{\sqsubset, \between\}$-structured patterns, and (*iii*) $\mathcal{O}(\sqrt{k \lg k})$ for $\{<, \sqsubset, \between\}$-structured patterns, where k is the size of the optimal solution and $\mathscr{H}(k) = \sum_{i=1}^{k} 1/i$ is the k-th harmonic number.

1 Introduction

Many biological molecules such as RNA and proteins exhibit a three-dimensional structure that determines most of their functionality. This three dimensional structure can be modeled in two dimensions by an edge-disjoint linear graph, *i.e.*, a graph with linearly ordered vertices that are incident to exactly one edge. The

⋆ Partially supported by the French-Italian Galileo Project PAI 08484VH.

⋆⋆ Partially supported by the Caesarea Edmond Benjamin de Rothschild Foundation Institute (CRI).

corresponding structure-similarity or structure-prediction problems that arise in such contexts usually translate to finding common edge-disjoint subgraphs, or common *structured patterns*, that occur in a family of general linear graphs. Examples of such problems are Longest Common Subsequence [19,20], Maximum Common Ordered Tree Inclusion [2,8,21], Arc-Preserving Subsequence [4,14,17], and Maximum Contact Map Overlap [15]. In this paper, we study a general framework for such problems which we call Maximum Common Structured Pattern (MCSP).

The MCSP problem was introduced by Davydov and Batzoglou [10] in the context of (non-coding) RNA secondary structure prediction via multiple structural alignment. There, an RNA sequence of n nucleotides is represented by a linear graph with n vertices, and an edge connects two vertices if and only if their corresponding nucleotides are complementary. A family of linear graphs is then used to represent a family of functionally-related RNAs, and a common structured pattern in such a family is considered to be a probably common secondary structure element of the family. The ordering amongst the vertices of a linear graph allows a pair of disjoint edges in the graph to be either preceding ($<$), nesting ($\sqsubset$), or crossing ($\between$). Since most RNA secondary structures translate to linear graphs with non-crossing edges, Davydov and Batzoglou [10] focused on the variant of MCSP where the common structured pattern is required to be non-crossing. However, there are known RNAs which have secondary structures that translate to linear graphs with a few edge-crossings (pseudo-knotted RNA secondary structures). Also, when predicting proteins rather than RNA structures, the non-crossing restriction becomes an even bigger limitation since the folding structures of proteins are often more complex than those of RNAs. In [16], it is argued that many important protein secondary structure elements like alpha helices and anti-parallel beta sheets exhibit $\{<, \between\}$-structured patterns, *i.e.* patterns which are non-nesting rather than non-crossing.

In this sequel, we present a framework which extends the work of [10] by considering different types of common structured patterns. Following [31], we consider structured patterns that are allowed to have crossing edges, and which might also be restricted to be non-nesting or non-preceding. More specifically, the MCSP problem receives as input a family of linear graphs and a non-empty subset $\mathcal{R} \subseteq \{<, \sqsubset, \between\}$, and the goal is to find a maximum common $\mathcal{R}$-structured pattern. We study the combinatorics behind the structures of these different types of patterns, with a focus on approximation algorithms for MCSP.

The paper is organized as follows. In the remaining part of this section we briefly review related work and notations that will be used throughout the paper. In Section 2, we discuss simple structured patterns (*i.e.* R-structured patterns, where $R \in \{<, \sqsubset, \between\}$) and $\{<, \sqsubset\}$-structured patterns. Following this, we discuss the more complex $\{<, \between\}$-structured patterns and $\{\sqsubset, \between\}$-structured patterns in Section 3 and Section 4 respectively. In Section 5, we deal with general structured patterns, *i.e.* $\{<, \sqsubset, \between\}$-structured patterns. An overview of the paper, along with some open problems, is given in Section 6.

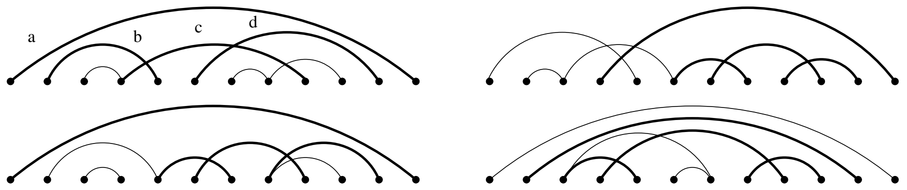


Fig. 1. Four linear graphs and a $\{<, \sqsubset, \between\}$-common structured pattern. The occurrence of the structured pattern in each graph is emphasized in bold. Edges **b**,**c**, and **d**, are nesting in edge **a**. Edge **b** precedes edge **d**, and they both cross edge **c**.

1.1 Related Work

There are many structural comparison problems that are closely related to MCSP. First, as mentioned previously, MCSP for $\{<, \sqsubset\}$-structured patterns has been studied by Davydov and Batzoglou in [10] under the name Maximum Common Nested Subgraph. Recently, new results concerning this problem appeared in [25]. We discuss the results of both these works in Section 2.

Closely related to MCSP are Arc-Preserving Subsequence [4,14,17], and Maximum Contact Map Overlap [15]. Both are concerned with finding maximum common subgraphs in a pair of linear graphs, except that in Arc-Preserving Subsequence the vertices of the linear graphs are assigned letters from some given alphabet, and an occurrence of a common subgraph in each of the linear graphs is required to preserve the letters, as well as their arc structure. Another closely related problem is Pattern Matching over 2-Interval Set [31], where one asks whether a structured pattern occurs in a given 2-interval set, which is a generalization of a linear graph. The 2-Interval Pattern problem [5,9,31] asks to find the maximum $\mathcal{R}$-structured pattern, for some given $\mathcal{R} \subseteq \{<, \sqsubset, \between\}$, in a single family of 2-interval sets.

There is a well-known bijective correspondence between $\{<, \sqsubset\}$-structured patterns and ordered forests – the nesting relation corresponds to the ancestor/predecessor relationship between the nodes, and the precedence relation corresponds to their order. Hence, MCSP for $\{<, \sqsubset\}$-structured patterns can be viewed as the problem of finding a tree which is included in all trees of a given tree family, the Maximum Common Ordered Tree Inclusion problem. Determining whether a tree is included in another is studied in [2,8,21]. Finding the maximum common tree included in a pair of trees can be done using the algorithms given in [22,29]. The MCSP problem for $\{<, \sqsubset\}$-structured patterns has been studied in [10,25]. We discuss the results there in Section 2.

Like $\{<, \sqsubset\}$-structured patterns, $\{\sqsubset, \between\}$-structured patterns also correspond to natural combinatorial objects, namely permutations (see Section 4). In [6], the authors studied the problem of determining whether a permutation-pattern occurs in a given permutation, the so called Pattern Matching for Permutations problem. This problem corresponds to determining whether a $\{\sqsubset, \between\}$-structured pattern is a subpattern of another $\{\sqsubset, \between\}$-structured pattern. Bose, Buss, and Lubiw proved that Pattern Matching for Permutations is **NP**-complete [6].

Determining whether a given $\{<, \between\}$-structured pattern occurs in a general linear graph has been studied in [16,26]. Gramm [16] gave a polynomial-time algorithm for this problem. Recently, Li and Li [26] proved that this algorithm was incorrect and showed the problem was in fact **NP**-complete. Prior to this, Blin *et al.* [5] proved that a generalization of this problem, where the linear graph is replaced by a 2-interval set, is **NP**-complete. Finally, probably the oldest and most famous problem related to MCSP is the Longest Common Subsequence (LCS) [19,20] problem, where one wishes to find the longest common subsequence in two or more sequences. Important developments of the initial algorithms of [19,20] can be found in [3,12,28]. Maier [27] proved that the LCS problem for multiple sequences is **NP**-hard.

1.2 Terminology and Basic Definitions

For a graph G, we denote $V(G)$ as the set of vertices and $E(G)$ as the set of edges. The *order* and the *size* of G stand for $|V(G)|$ and $|E(G)|$, respectively. A *linear graph* of order n is a vertex-labeled graph where each vertex is labeled by a distinct label from $\{1, 2, \ldots, n\}$. Thus, it can be viewed as a graph with vertices embedded on the integral line, yielding a total order amongst them. 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 linear graph are *disjoint* if they do not share a common vertex. A linear graph G is said to be *edge-disjoint* if it is composed of disjoint edges, *i.e.* if G is a matching. Of particular interest are the relations between pairs of disjoint edges [31]: Let $e = (i, j)$ and $e' = (i', j')$ be two disjoint edges in a linear graph G; we write (i) $e < e'$ (e *precedes* e') if $i < j < i' < j'$, (ii) $e \sqsubset e'$ (e is *nested* in e') if $i' < i < j < j'$ and (iii) $e \between e'$ (e and e' *cross*) if $i < i' < j < j'$.

Two edges e and e' are *R-comparable*, for some $R \in \{<, \sqsubset, \between\}$, if eRe' or $e'Re$. For a subset $\mathcal{R} \subseteq \{<, \sqsubset, \between\}$, $\mathcal{R} \neq \emptyset$, e and e' are said to be *$\mathcal{R}$-comparable* if e and e' are R-comparable for some $R \in \mathcal{R}$. A set of edges E (or a linear graph G with $E(G) = E$) is *$\mathcal{R}$-comparable* if any pair of distinct edges $e, e' \in E$ are $\mathcal{R}$-comparable. A *subgraph* of a linear graph G is a linear graph H which can be obtained from G by a series of vertex and edge deletions, where a deletion of vertex i results in removing vertex i and all edges incident to it from the graph, and then relabeling all vertices j with $j > i$ to $j - 1$. An edge-disjoint subgraph of a linear graph is called a *structured-pattern.* For a family of linear graphs $\mathcal{G} = G_1, \ldots, G_n$, a *common structured pattern* of $\mathcal{G}$ is an edge-disjoint linear graph H that is a subgraph of G_i, for all $1 \leq i \leq n$. Following the above notation, H is called an *$\mathcal{R}$-structured pattern*, for some non-empty $\mathcal{R} \subseteq \{<, \sqsubset, \between\}$, if $E(H)$ is *$\mathcal{R}$-comparable.*

Definition 1. *Given a family of linear graphs $\mathcal{G} = G_1, \ldots, G_n$ and a subset $\mathcal{R} \subseteq \{<, \sqsubset, \between\}$, $\mathcal{R} \neq \emptyset$, the* Maximum Common Structured Pattern (MCSP) *problem asks to find a maximum-size common $\mathcal{R}$-structured pattern of $\mathcal{G}$.*

We will use the following terminology to describe special edge-disjoint linear graphs. A linear graph is called a *sequence* if it is $\{<\}$-comparable, a *tower* if it is $\{\sqsubset\}$-comparable, and a *staircase* if it is $\{\between\}$-comparable. We define the *width* (resp. *height* and *depth*) of a linear graph to be the size of the maximum cardinality sequence (resp. tower and staircase) subgraph of the graph. A $\{<, \sqsubset\}$-comparable linear graph with the additional property that any two maximal towers in it do not share an edge is called a *sequence of towers*. Similarly, a $\{<, \between\}$-comparable linear graph is a *sequence of staircases* if any two maximal staircases do not share an edge. A *tower of staircases* is a $\{\sqsubset, \between\}$-comparable linear graph where any pair of maximal staircases do not share an edge, and a *staircase of towers* is a $\{\sqsubset, \between\}$-comparable linear graph where any pair of maximal towers do not share an edge. A sequence of towers (resp. sequence of staircases, tower of staircases, and staircase of towers) is *balanced* if all of its maximal towers (resp. staircases, staircases, and towers) are of equal size. Figure 2 illustrates an example of the above types of linear graphs.

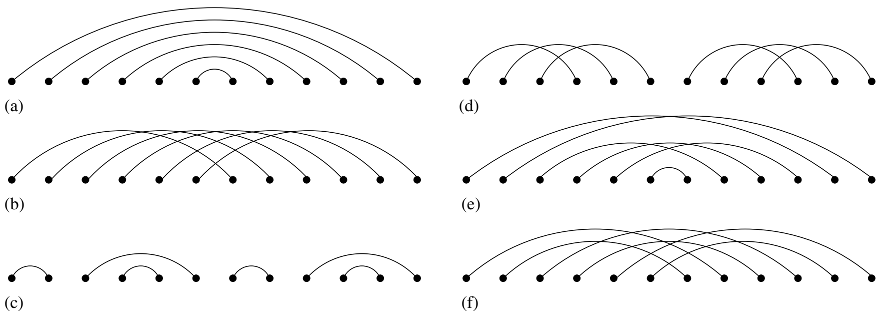


Fig. 2. Examples of restricted edge-disjoint linear graphs: (a) a tower of height 6, (b) a staircase of depth 6, (c) a sequence of towers of width 4 and height 2, (d) a balanced sequence of staircases of width 2 and depth 3, (e) a tower of staircases of height 3 and depth 3 and (f) a balanced staircase of towers of height 2 and depth 3

2 Simple and $\{<, \sqsubset\}$-Structured Patterns

A structured pattern is simple if it is an R-structured pattern for a single relation $R \in \{<, \sqsubset, \between\}$. We begin our study by considering the MCSP problem for simple structured patterns, and for $\{<, \sqsubset\}$-structured patterns. We first discuss the analogy between the relations we defined for disjoint edges in a linear graph, and well-studied relations defined for families of intervals. We show that known algorithms on interval families can be used to solve MCSP for simple structured patterns in polynomial-time. Following this, we discuss results presented in [10,25] for MCSP for $\{<, \sqsubset\}$-structured patterns.

For a given linear graph G of size m, let $\mathcal{I}(G) = \{[i, j] \mid (i, j) \in E(G)\}$ be the family of intervals obtained by considering each edge of G as an interval of the line, closed between both its endpoints. A pair of $\{<\}$-comparable edges in

$E(G)$ correspond to a pair of disjoint intervals in $\mathcal{I}(G)$, a pair of $\{\sqsubset\}$-comparable edges correspond to a pair of nesting intervals, and a pair of $\{\between\}$-comparable edges correspond to a pair of overlapping intervals. Note that this correspondence is bi-directional only if G is edge-disjoint, since a pair of edges sharing a vertex can correspond to a pair of nesting or overlapping intervals. Nevertheless, we can always modify $\mathcal{I}(G)$ in such a way, so that all intervals have unique endpoints, and so that any pair of intervals who shared an endpoint now become non-nesting (resp. non-overlapping). A maximum pairwise disjoint subset of intervals can be computed in linear time using standard dynamic-programming, assuming the interval family is given in a sorted manner [18] (which we can provide in linear time in our case using bucket sorting). A maximum pairwise nesting subset can be computed in $\mathcal{O}(m \lg\lg m)$ in an interval family of m intervals (see for example the algorithm in [7]), and a maximum pairwise overlapping subset in $\mathcal{O}(m^{1.5})$ time [30].

Lemma 1. *Let G be a linear graph of size m. Then there exists a $\mathcal{O}(m)$ (resp. $\mathcal{O}(m \lg\lg m)$ and $\mathcal{O}(m^{1.5})$) time algorithm for finding the largest $\{<\}$-comparable (resp. $\{\sqsubset\}$-comparable and $\{\between\}$-comparable) subgraph of G.*

Theorem 1. *The* MCSP *problem for $\{<\}$-structured patterns (resp. $\{\sqsubset\}$-structured patterns and $\{\between\}$-structured patterns) is solvable in $\mathcal{O}(nm)$ (resp. $\mathcal{O}(nm \lg\lg m)$ and $\mathcal{O}(nm^{1.5})$) time, where $n = |\mathcal{G}|$ and $m = \max_{G \in \mathcal{G}} |E(G)|$.*

We next consider $\{<, \sqsubset\}$-structured patterns. The MCSP problem for this type of patterns was considered by [10,25], in the context of multiple RNA structural alignment.

Theorem 2 ([25]). *The* MCSP *problem for $\{<, \sqsubset\}$-structured patterns is* **NP***-hard even if each input linear graph is a sequence of towers of height at most* 2.

Note, however, that the problem MCSP is polynomial-time solvable in case the number of input linear graphs is a constant [25]. Also, it is proven in [25] that MCSP for $\{<, \sqsubset\}$-structured patterns is approximable with ratio $\lg k + 1$, where k is the size of the optimal solution.

Theorem 3 ([25]). *The* MCSP *problem for $\{<, \sqsubset\}$-structured patterns is approximable within ratio $\mathcal{O}(\lg k)$ in $\mathcal{O}(nm^2)$ time, where k is the size of an optimal solution, $n = |\mathcal{G}|$, and m is the maximum size of any graph in $\mathcal{G}$.*

3 $\{<, \between\}$-Structured Patterns

We now turn to consider MCSP for $\{<, \between\}$-structured patterns. We begin by proving a tight hardness result for the problem. Following this, we present an approximation algorithm for the problem which achieves a ratio of $2\mathscr{H}(k)$ in $\mathcal{O}(nm^3 \log^2 m)$ time, where k is the size of an optimal solution, $\mathscr{H}(k) = \sum_{i=1}^{k} 1/i$, $n = |\mathcal{G}|$, and m is the maximum size of any graph in $\mathcal{G}$.

Theorem 4. *The* MCSP *problem for* $\{<, \between\}$*-structured patterns is* **NP***-hard even if each input linear graph is a sequence of staircases of depth at most* 2*.*

A recent result [26] implies that MCSP for $\{<, \between\}$-structured patterns is hard even if $\mathcal{G}$ consists of only two graphs of unlimited structure. We next show that one can approximate the maximum common $\{<, \between\}$-structured pattern of $\mathcal{G}$ within ratio $2\mathscr{H}(k)$. The first ingredient of our proof is to observe that every $\{<, \between\}$-structured pattern contains a sequence of staircases of substantial size. The second ingredient consists in showing that any sequence of staircases contains a balanced subgraph of substantial size.

Lemma 2. *Let H be a $\{<, \between\}$-comparable linear graph. There exists a partition $E(H) = E_{RED} \cup E_{BLUE}$ such that both $H[E_{RED}]$ and $H[E_{BLUE}]$ are sequences of staircases.*

Lemma 3. *Let H be a sequence of staircases of size k. Then H contains a balanced sequence of staircases with at least $\frac{k}{\mathscr{H}(k)}$ edges.*

As a direct corollary of Lemmas 2 and 3, we obtain:

Corollary 1. *Any $\{<, \between\}$-comparable linear graph of size k contains as a subgraph a balanced sequence of staircases of size at least $\frac{k}{2\mathscr{H}(k)}$.*

What is left is to show that, given a set of linear graphs, one can find in polynomial-time the size of the largest balanced sequence of staircases that occurs in each input linear graph. For this particular purpose, we present Algorithm Bal-Seq-Staircase in Figure 3.

```
Algorithm Bal-Seq-Staircase($G, w, d$).
Data   : A linear graph $G$ of size $m$, and two positive integers $d$ and $w$.
Result : true iff $G$ contains a balanced sequence of staircases of width $w$ and
         depth $d$.
begin
    1. $E' \leftarrow \emptyset$
    2. for $i = 1, 2, \ldots, m-1$ do
        (a) Let $j$ be the smallest integer such that $G[i, \ldots, j]$ contains as a
        subgraph a staircase of size $d$ (set $j = \infty$ if no such integer exists).
        (b) if $j \neq \infty$ then $E' \leftarrow E' \cup \{(i, j)\}$.
    end
    3. Compute $H$, the maximum $\{<\}$-comparable subgraph of $G' = (V(G), E')$.
    4. if $|E(H)| \geq w$ then return true else return false.
end
```

Fig. 3. Algorithm Bal-Seq-Staircase for finding a balanced sequence of staircases of width w and depth d in a linear graph. For a linear graph $G \in \mathcal{G}$, and two integers i and j with $1 \leq i < j \leq |V(G)|$, $G[i, \ldots, j]$ stands for the subgraph of G obtained by deleting all vertices labeled k with $k < i$ or $j < k$.

Lemma 4. *Algorithm* Bal-Seq-Staircase(G, w, d) *runs in* $\mathcal{O}(m^{2.5} \log m)$ *time and returns* ***true*** *if and only if* G *contains a balanced sequence of staircases of width* w *and depth* d.

Theorem 5. *The* MCSP *problem for* $\{<, \between\}$*-structured patterns is approximable within ratio* $2\mathscr{H}(k)$ *in* $\mathcal{O}(nm^{2.5} \log^2 m)$ *time, where* k *is the size of an optimal solution,* $n = |\mathcal{G}|$*, and* m *is the maximum size of any graph in* $\mathcal{G}$.

4 $\{\sqsubset, \between\}$-Structured Patterns

We next consider $\{\sqsubset, \between\}$-structured patterns. We begin by proving a hardness result, analogous to Theorem 4, which states that MCSP for $\{\sqsubset, \between\}$-structured patterns is **NP**-hard even if the input consists of towers of staircases of depth at most 2. However, unlike the approach we used for $\{<, \between\}$-structured patterns, we cannot use towers of staircases to obtain very good approximations of maximum common $\{\sqsubset, \between\}$-structured patterns. We show that there exists a $\{\sqsubset, \between\}$-comparable linear graph of size k which does not contain a tower of staircases of size $\varepsilon\sqrt{k}$ for some constant ε. On the other hand, such a graph must contain either a tower or a staircase with at least $\sqrt{k}$ edges.

Theorem 6. *The* MCSP *problem for* $\{\sqsubset, \between\}$*-structured patterns is* **NP***-hard even if each input linear graph is a tower of staircases of depth at most* 2.

We now turn to approximating MCSP for $\{\sqsubset, \between\}$-structured patterns. First, let us observe the one-to-one correspondence between $\{\sqsubset, \between\}$-structured patterns and permutations. Let H be a $\{\sqsubset, \between\}$-comparable linear graph of size k. Then the vertices in H which are left endpoints of edges are labeled $\{1, \ldots, k\}$ and the right endpoints are labeled $\{k+1, \ldots, 2k\}$. The permutation π_H corresponding to H is defined by $\pi_H(i) = j - k \iff (i, j) \in E(H)$. Clearly, all $\{\sqsubset, \between\}$-comparable linear graphs have corresponding permutations, and vice versa. It follows from this bijective correspondence, that the number of different $\{\sqsubset, \between\}$-comparable linear graphs of size k is exactly $k!$. Moreover, notice that increasing subsequences in π_H correspond to $\{\between\}$-comparable subgraphs of H, while decreasing subsequences correspond to $\{\sqsubset\}$-comparable subgraphs. The well known Erdős-Szekeres Theorem [13] states that any permutation on $1, \ldots, k$ contains either an increasing or a decreasing subsequence of size at least $\sqrt{k}$ (see also Lemma 6). Hence, using the algorithms in Lemma 1 for finding the maximum common $\{\sqsubset\}$-structured $\{\between\}$-structured patterns, we obtain the following theorem:

Theorem 7. *The* MCSP *problem for model* $\mathcal{M} = \{\sqsubset, \between\}$ *is approximable within ratio* $k^{1/2}$ *in* $\mathcal{O}(nm^{1.5})$ *time, where* k *is the size of an optimal solution* $n = |\mathcal{G}|$*, and* $m = \max_{G \in \mathcal{G}} |E(G)|$.

Alon [1] recently showed that towers of staircases cannot be used to obtain a much better approximation algorithm than the one proposed above. To see this, let us count the number of different towers of staircases with k edges. Note that the number of towers of staircases of size k and of height h, is exactly the number of different partitions of $\{1, \ldots, k\}$ into h consecutive intervals, *i.e.* $\binom{k}{h-1}$. Hence the total number of towers of staircases of size k equals $\sum_{h=1}^{k} \binom{k}{h-1} = 2^k - 1 < 2^k$. Using this simple observation, the following lemma can be proved.

Lemma 5 ([1]). *There exists a $\{\sqsubset, \between\}$-comparable linear graph of size $K = \Omega(k^2)$ which does not contain a tower of staircases of size k.*

5 General Structured Patterns

In this section we consider MCSP for general, *i.e.*, $\{<, \sqsubset, \between\}$, structured patterns. Since $\{<, \sqsubset, \between\}$-structured patterns generalize all other types of patterns, all hardness results presented in previous sections apply for general structured patterns as well. We present three approximation algorithms with increasing time complexities and decreasing approximation ratios.

Observe that both $<$ and $\sqsubset$ induce partial orders on the edges of a given linear graph. Recall now that a *chain* (resp. *anti-chain*) in a partial order is a subset of pairwise comparable (resp. incomparable) elements. Dilworth's Theorem [11] states that in any partial order, the size of the maximum chain equals the size of the minimum anti-chain partitioning. Therefore, in any partial order on k elements, the size of the maximum chain multiplied by the size of the maximum anti-chain is at least k. The following lemma states this property in our terms.

Lemma 6. *Let H be a $\{<, \sqsubset, \between\}$-comparable linear graph of size k, width $w(H)$, and height $h(H)$. Also, let $hd(H)$ and $wd(H)$ be the sizes of the maximum $\{\sqsubset, \between\}$-comparable and $\{<, \between\}$-comparable subsets of $E(H)$. Then $k \leq w(H) \cdot hd(H)$ and $k \leq h(H) \cdot wd(H)$.*

An immediate consequence of Lemma 6 is as follows.

Lemma 7. *Let H be a $\{<, \sqsubset, \between\}$-comparable linear graph of size k. Then H contains a simple structured pattern of size at least $k^{1/3}$.*

Combining the lemma above with the fact that a maximum common simple structured pattern of $\mathcal{G}$ can be found in $\mathcal{O}(nm^{1.5})$ time (Theorem 1), we obtain our first approximation algorithm for general structured patterns.

Theorem 8. *The* MCSP *problem for $\{<, \sqsubset, \between\}$-structured patterns is approximable within ratio $\mathcal{O}(k^{2/3})$ in $\mathcal{O}(nm^{1.5})$ time, where k is the size of an optimal solution, $n = |\mathcal{G}|$, and $m = \max_{G \in \mathcal{G}} |E(G)|$.*

It is easily seen that Lemma 7 is tight. One way to obtain an extremal example of this is as follows: Take $k^{1/3}$ balanced towers of staircases, each one of depth $k^{1/3}$ and height $k^{1/3}$, and concatenate them one next to the other into one supergraph of size k, reassigning labels accordingly.

Lemma 8. *Let k be an integer such that $k^{1/3}$ is also integer. Then there exists an $\{<, \sqsubset, \between\}$-comparable linear graph of size k that does not contain a simple structured pattern of size $\varepsilon\, k^{1/3}$ for any $\varepsilon > 1$.*

Dilworth's theorem does not apply on the crossing relation since it is not transitive. However, an analogous result proven in [23] (see also [24]) implies that for any $\{<, \sqsubset, \between\}$-comparable linear graph H, $|E(H)| = \mathcal{O}(d \cdot wh \lg wh)$, where d and wh are sizes of the maximum $\{\between\}$-comparable and $\{<, \sqsubset\}$-comparable subsets of $E(H)$. This yields the following analogous of Lemma 6.

Lemma 9. *Let H be a $\{<, \sqsubset, \between\}$-comparable linear graph of size k. Then H contains a subgraph of size $\Omega(\sqrt{k/\lg k})$ which is either $\{<, \sqsubset\}$-comparable or $\{\between\}$-comparable.*

Using Lemma 9, the algorithm for finding a maximum structured pattern given in Theorem 1, and the $\mathcal{O}(\lg k)$-approximation algorithm for $\{<, \sqsubset\}$-structured patterns given in Theorem 3, we obtain our second approximation algorithm.

Theorem 9. *The* MCSP *problem for $\{<, \sqsubset, \between\}$-structured patterns is approximable within ratio $\mathcal{O}(\sqrt{k \lg^3 k})$ in $\mathcal{O}(nm^2)$ time.*

For our third algorithm, we show that any $\{<, \sqsubset, \between\}$-comparable linear graph contains a subgraph of sufficient size that is either a tower or a balanced sequence of staircases.

Lemma 10. *Let H be a $\{<, \sqsubset, \between\}$-comparable linear graph of size k. Then H contains either a tower or a balanced sequence of staircases of size $\Omega(\sqrt{k/\lg k})$.*

Applying Lemma 3 and the algorithms for finding the maximum common tower and balanced sequence of staircases in $\mathcal{G}$ given in Theorems 1 and 5, respectively, we obtain the following theorem.

Theorem 10. *The* MCSP *problem for $\{<, \sqsubset, \between\}$-structured patterns is approximable within ratio $\mathcal{O}(\sqrt{k \lg k})$ in $\mathcal{O}(nm^{2.5} \lg^2 m)$ time.*

We next consider subgraphs of $\{<, \sqsubset, \between\}$-comparable linear graphs that are comparable by pairs of relations. We show that any $\{<, \sqsubset, \between\}$-comparable linear graph of size k contains such a subgraph of size at least $m^{2/3}$, and that this lower bound is relatively tight. Unfortunately, this result can not be applied for approximation purposes (approximating MCSP for $\{\sqsubset, \between\}$-patterns remains the bottleneck). Nevertheless, we present this result on account of independent interest.

Lemma 11. *Let H be a $\{<, \sqsubset, \between\}$-comparable graph of size k. Then H has a subgraph of size $\varepsilon\, k^{2/3}$, where $\varepsilon = \frac{\sqrt{17}-1}{8}$, which is either $\{<, \sqsubset\}$-comparable, $\{<, \between\}$-comparable, or $\{\sqsubset, \between\}$-comparable.*

We believe the bound of Lemma 11 to be not the best possible. However, combining Lemmas 6 and 8, we show that the above lemma is relatively tight.

Lemma 12. *Let k be an integer such that $k^{1/3}$ is integer. Then there exists a $\{<, \sqsubset, \between\}$-comparable linear graph of size k that contains neither a $\{<, \sqsubset\}$-comparable subgraph, nor a $\{<, \between\}$-comparable subgraph, nor a $\{\sqsubset, \between\}$-comparable subgraph of size least $\varepsilon\, k^{2/3}$ for any $\varepsilon > 1$.*

6 Discussion and Open Problems

In this paper we introduced MCSP as a general framework for many structure-comparison and structure-prediction problems, that occur mainly in computational molecular biology. Our framework followed the approach in [31] by analyzing all types of $\mathcal{R}$-structured patterns, $\mathcal{R} \subseteq \{<, \sqsubset, \between\}$. We gave tight hardness results for finding maximum common $\{<, \between\}$-structured patterns and maximum common $\{<, \between\}$-structured patterns. We also proved that MCSP is approximable within ratio: (i) $2\mathscr{H}(k)$ for $\{<, \between\}$-structured patterns, (ii) $k^{1/2}$ for $\{\sqsubset, \between\}$-structured patterns, and (iii) $\mathcal{O}(\sqrt{k \lg k})$ for $\{<, \sqsubset, \between\}$-structured patterns.

There are many questions left open by our study. Below we list some of them. According to Lemma 11, we could improve in terms of approximation ratio on all the algorithms suggested for general structured patterns, if we had a better approximation algorithm for $\{\sqsubset, \between\}$-structured patterns. Is there an approximation algorithm which achieves a better ratio then the simple $\sqrt{k}$ algorithm? On the same note, can lower bounds on the approximation factor of MCSP for $\{<, \sqsubset, \between\}$-structured patterns or $\{\sqsubset, \between\}$-structured patterns be proven? How about $\{<, \sqsubset\}$-structured patterns or $\{<, \between\}$-structured patterns?

References

1. Alon, N.: Private communication (2006)
2. Alonso, L., Schott, R.: On the tree inclusion problem. In: Borzyszkowski, A.M., Sokolowski, S. (eds.) MFCS 1993. LNCS, vol. 711, pp. 211–221. Springer, Heidelberg (1993)
3. Apostolico, A., Guerra, C.: The longest common subsequence problem revisited. Algorithmica 2, 315–336 (1987)
4. Blin, G., Fertin, G., Rizzi, R., Vialette, S.: What makes the arc-preserving subsequence problem hard ? In: Sunderam, V.S., van Albada, G.D., Sloot, P.M.A., Dongarra, J.J. (eds.) ICCS 2005. LNCS, vol. 3515, pp. 860–868. Springer, Heidelberg (2005)
5. Blin, G., Fertin, G., Vialette, S.: New results for the 2-interval pattern problem. In: Sahinalp, S.C., Muthukrishnan, S.M., Dogrusoz, U. (eds.) CPM 2004. LNCS, vol. 3109, Springer, Heidelberg (2004)
6. Bose, P., Buss, J.F., Lubiw, A.: Pattern matching for permutations. IPL 65(5), 277–283 (1998)
7. Chang, M.-S., Wang, F.-G.: Efficient algorithms for the maximum weight clique and maximum weight independent set problems on permutation graphs. IPL 43(6), 293–295 (1992)
8. Chen, W.: More efficient algorithm for ordered tree inclusion. J. Algorithms 26(2), 370–385 (1998)
9. Crochemore, M., Hermelin, D., Landau, G.M., Vialette, S.: Approximating the 2-interval pattern problem. In: Brodal, G.S., Leonardi, S. (eds.) ESA 2005. LNCS, vol. 3669, pp. 426–437. Springer, Heidelberg (2005)
10. Davydov, E., Batzoglou, S.: A computational model for RNA multiple structural alignment. In: Sahinalp, S.C., Muthukrishnan, S.M., Dogrusoz, U. (eds.) CPM 2004. LNCS, vol. 3109, pp. 254–269. Springer, Heidelberg (2004)

11. Dilworth, R.P.: A decomposition theorem for partially ordered sets. Annals of Mathematics Series 2 51, 161–166 (1950)
12. Eppstein, D., Galil, Z., Giancarlo, R., Italiano, G.F.: Sparse dynamic programming I: Linear cost functions. J. ACM 39(3), 519–545 (1992)
13. Erdős, P., Szekeres, G.: A combinatorial problem in geometry. Compositio Mathematica 2, 463–470 (1935)
14. Evans, P.A.: Algorithms and complexity for annotated sequence analysis. PhD thesis, University of Alberta (1999)
15. Goldman, D., Istrail, S., Papadimitriou, C.H.: Algorithmic aspects of protein structure similarity. In: Proc. 40th Foundations of Computer Science (FOCS), pp. 512–522 (1999)
16. Gramm, J.: A polynomial-time algorithm for the matching of crossing contact-map patterns. IEEE/ACM Trans. Comp. Biol. and Bioinfo. 1(4), 171–180 (2004)
17. Gramm, J., Guo, J., Niedermeier, R.: Pattern matching for arc-annotated sequences. In: Agrawal, M., Seth, A.K. (eds.) FST TCS 2002: Foundations of Software Technology and Theoretical Computer Science. LNCS, vol. 2556, pp. 182–193. Springer, Heidelberg (2002)
18. Gupta, U.I., Lee, D.T., Leung, J.Y-T.: Efficient algorithms for interval graph and circular-arc graphs. Networks 12, 459–467 (1982)
19. Hirschberg, D.S.: Algorithms for the longest common subsequence problem. J. ACM 24(4), 664–675 (1977)
20. Hunt, J.W., Szymanski, T.G.: A fast algorithm for computing longest common subsequences. Communications of the ACM 20, 350–353 (1977)
21. Kilpeläinen, P., Mannila, H.: Ordered and unordered tree inclusion. SIAM J. Comp. 24(2), 340–356 (1995)
22. Klein, P.N.: Computing the edit-distance between unrooted ordered trees. In: Bilardi, G., Pietracaprina, A., Italiano, G.F., Pucci, G. (eds.) ESA 1998. LNCS, vol. 1461, pp. 91–102. Springer, Heidelberg (1998)
23. Kostochka, A.: On upper bounds on the chromatic numbers of graphs. Transactions of the Institute of Mathematics (Siberian Branch of the Academy of Sciences in USSR) 10, 204–226 (1988)
24. Kostochka, A., Kratochvil, J.: Covering and coloring polygon-circle graphs. Discrete Mathematics 163, 299–305 (1997)
25. Kubica, M., Rizzi, R., Vialette, S., Waleń, T.: Approximation of RNA multiple structural alignment. In: Lewenstein, M., Valiente, G. (eds.) CPM 2006. LNCS, vol. 4009, pp. 211–222. Springer, Heidelberg (2006)
26. Li, S.C., Li, M.: On the complexity of the crossing contact map pattern matching problem. In: Bücher, P., Moret, B.M.E. (eds.) WABI 2006. LNCS (LNBI), vol. 4175, pp. 231–241. Springer, Heidelberg (2006)
27. Maier, D.: The complexity of some problems on subsequences and supersequences. J. ACM 25(2), 322–336 (1978)
28. Masek, W.J., Paterson, M.S.: A faster algorithm computing string edit distances. J. Comp. and Syst. Sc. 20(1), 18–31 (1980)
29. Shasha, D., Zhang, K.: Simple fast algorithms for the editing distance between trees and related problems. SIAM J. Comp. 18(6), 1245–1262 (1989)
30. Tiskin, A.: Longest common subsequences in permutations and maximum cliques in circle graphs. In: Lewenstein, M., Valiente, G. (eds.) CPM 2006. LNCS, vol. 4009, pp. 270–281. Springer, Heidelberg (2006)
31. Vialette, S.: On the computational complexity of 2-interval pattern matching problems. Theoretical Computer Science 312(2-3), 223–249 (2004)

Identification of Distinguishing Motifs

WangSen Feng[1], Zhanyong Wang[2], and Lusheng Wang[2]

[1] Department of Computer Science
Peking University, People's Republic of China
[2] Department of Computer Science, City University of Hong Kong, Hong Kong
zhyong@cs.cityu.edu.hk, cswangl@cityu.edu.hk

Abstract. Motivation: Motif identification for sequences has many important applications in biological studies, e.g., diagnostic probe design, locating binding sites and regulatory signals, and potential drug target identification. There are two versions.

1. **Single Group:** Given a group of n sequences, find a length-l motif that appears in each of the given sequences and those occurrences of the motif are *similar*.
2. **Two Groups:** Given two groups of sequences B and G, find a length-l (distinguishing) motif that appears in every sequence in B and does not appear in anywhere of the sequences in G.

Here the occurrences of the motif in the given sequences have errors. Currently, most of existing programs can only handle the case of single group. Moreover, it is very difficult to use edit distance (allowing indels and replacements) for motif detection.

Results: (1) We propose a randomized algorithm for the one group problem that can handle indels in the occurrences of the motif. (2) We give an algorithm for the two groups problem. (3) Extensive simulations have been done to evaluate the algorithms.

Keywords: motif detection, EM Algorithms, and two groups.

1 Introduction

Motif identification for DNA sequences has many important applications in biological studies, e.g., diagnostic probe design, locating binding sites and regulatory signals, and potential drug target identification [6,17,19]. The most general problem is as follows:

Two Groups: Given two groups of sequences B and G, find a length-l (distinguishing) motif that appears in every sequence in B and does not appear in anywhere of the sequences in G.

The motif we want to find is called the *distinguishing* motif that can differentiate the two groups. If no error is allowed, the problem is easy. However, in practice, the occurrence of the motif in each of the given sequences has errors (mutations and indels). The problem becomes extremely hard when errors appear. Many mathematic models have been proposed.

The following definition was first stated in [12].

Distinguishing substring selection: given a set $B = \{s_1, s_2, \ldots, s_{n_1}\}$ of n_1 (bad) strings of length at least l, and a set $G = \{g_1, g_2, \ldots, g_{n_2}\}$ of n_2 (good) strings of length at least l, the problem is to find a string s of length l such that for each string $s_i \in B$ there exists a length-l substring t_i of s_i with $d_H(s, t_i) \leq d_b$ and for any length-l substring u_i of $g_i \in G$, $d_H(s, u_i) \geq d_g$. Here d_b and d_g are two parameters, called the *radius* of the two groups. We want d_b to be small and d_g to be large. $d_H(,)$ represents the Hamming distance between two strings.

When indels are allowed in the occurrences of the motif, the problem is called the *distinguishing subsequence selection* problem, where the input is the same as the previous problem and the objective here is to find a string s of length l such that for each string $s_i \in B$ there exists a substring t_i of s_i with $d(s, t_i) \leq d_b$ and for any substring u_i of $g_i \in G$, $d(s, u_i) \geq d_g$. Here $d(,)$ represents the edit distance.

A special case, where only one group is involved is as follows:

Single Group: Given a group of n given sequences, find a motif that appears in each of the given sequences and those occurrences of the motif are *similar*.

There are several ways to define the similarity of the motif occurrences, e.g., the *consensus score*, *general consensus score* and SP-score [8,14]. Here we use the following mathematical definitions.

The Closest Substring Problem: Given n sequences $\{s_1, s_2, \ldots, s_n\}$, each is of length m ($m \geq l$), the problem is to find a center string s of length l and a substring t_i of length l in s_i such that

$$d = \max_{i=1}^{n} d_H(s, t_i)$$

is minimized, where $d_H(,)$ is the Hamming distance between the two strings of the same length.

When indels are allowed, the problem is called the *closest subsequence problem*, where the input is the same as the closest substring problem and the objective here is to find a center string s of length l and a substring t_i in s_i such that

$$d = \max_{i=1}^{n} d(s, t_i)$$

is minimized, where $d(,)$ is the edit distance between the two sequences.

The closest substring problem was proved to be NP-hard in [12]. Thus, the more general problem, the distinguishing substring selection problem, is also NP-hard. Some polynomial time approximation schemes have been designed for both the closest substring problem and the distinguishing substring selection problem [16,5]. However, those polynomial time approximation schemes are too slow and do not work very well in practice.

Several programs have been developed for the single group problem based on various mathematical models. Bailey and Elkan designed MEME that uses the EM algorithm for motif identification [1]. The algorithm allows the motif to be absent in some of the given sequences. Waterman introduced the extended sample-driven approach in [25]. Keich and Pevzner proposed a variant of the extended sample-driven approach in [10]. Keich and Pevzner also developed another motif detection program in [11]. Buhler and Tompa develop a software, PROJECTION, that combines the EM algorithm and a random projection approach [3]. PROJECTION outperforms the programs in [10,11]. Recently, Price, Ramabhadran and Pevzner designed a new algorithm that uses branching from sample strings [21]. Their program, PatternBranching, is much faster than the previously best known program, PROJECTION. All the algorithms and programs mentioned here do not allow indels in the occurrences of the motif and can only handle the one group problem. As pointed out by Buhler and Tompa, using edit distasnce for motif detection seems extremely difficult for the random projection and EM approach [3]. Along this line, some algorithms can handle one gap (a segment of consecutive spaces) [9,13,4].

In this paper, we design an algorithm that allows indels in the occurrences of the motif. Our algorithm is an extension of the EM approach. We also propose an algorithm to solve the two groups problem.

2 Representations of Motifs

In our algorithms, we use two representations of motifs, *consensus pattern*, and *profile* [7]. Let $t_1, t_2, \cdots, t_n$ be n strings of length l. Each t_i is an occurrence of a motif. The *consensus pattern* of the n occurrences is obtained by choosing the letter that appears the most in each of the l columns. Here the alphabet is $\{A, C, G, T\}$. The *profile* of the n occurrences is a $4 \times l$ matrix W, each cell $W(i, j)$ is a number indicating the occurrence rate of letter i in column j. Figure 1 gives an example. In this paper, we will use the two representations to design our algorithm. We find that both representations are useful. We actually use the profile representation in the early stage of the EM algorithm. The consensus pattern representation is used after the EM refining procedure to improve the accuracy.

3 Computing the Closest Subsequence

In this section, we describe the algorithm for computing the closest subsequence problem and show the simulation results. The general frame is similar as in [24].

3.1 The General Frame

From mathematical point of view, the basic ideas of the algorithm we are going to describe are from [15,16]. In [15,16], a random sampling technique is used to select substrings from the given sequences to form a set of linear inequalities. Here we use the technique to directly give the starting matrix for the EM approach.

The idea is to randomly choose k positions among the l positions of the motif. Then we can guess the true motif at the k selected positions by trying 4^k possible strings of length k. The partial motif (with k guessed letters) is used to search all the given sequences s_1, s_2, …, s_n to find a t_i from each s_i that is closest to the partial motif, i.e., the number of mismatches between t_i and the partial motif at those selected k positions is minimized. Let K be the set of k selected positions, and t'_i and s two strings of length l. We use $d(t'_i, s|K)$ to denote the number of mismatches between t'_i and s at the positions in K. The algorithm is given in Figure 2.

```
                       caaccca
                       caacccc
                       catcccg
                       catccct
                       caccccca
                 --------------------
consensus pattern      caaccca
another con. pattern   catccca
                  (a)
```

a	0	1	0.4	0	0	0	0.4
c	1	0	0.2	1	1	1	0.2
g	0	0	0.0	0	0	0	0.2
t	0	0	0.4	0	0	0	0.2

(b)

Fig. 1. (a) The 5 occurrences of the motif and the consensus patterns. (b) The profile matrix.

In Step (3), when using a partial string s_p to search a given string s_i, we try to find a substring of length l such that the number of mismatches at those k selected positions is minimized. Steps (1)-(4) generate a candidate of the motif. Since this is a randomized algorithm, different executions generate different results. Thus, in the algorithm, we repeat Steps (1)-(4) several times to enhance the quality of the output. The following theorem is from [24].

Theorem 1. [24] Let s^* be the optimal center string and t_i the length l substring of s_i in the optimal solution. Set $k = \lceil \frac{4}{\epsilon^2} \log(nm) \rceil$. With probability $1 - 2(nm)^{-\frac{1}{3}}$, the algorithm finds a string s of length l and a length l substring t'_i for each given string s_i such that $\sum_{i=1}^{n} d(s, t'_i) \leq \sum_{i=1}^{n}(d(s^*, t_i) + 2\epsilon l)$. The running time of the algorithm is $O(4^k nml)$.

In practice, k has to be a relatively big number in order to get satisfactory results. The speed is far below that of PROJECTION. PROJECTION uses a random projection method to find seeds and uses an EM method to do local search. In [24], a combined approach was proposed. It uses the EM algorithm to replace Steps (3) and (4) in Figure 2.

3.2 EM Method

Lawrence and Reilly were the first to introduce the Expectation Maximization (EM) algorithm in motif finding problems [13]. Bailey and Elkan used it in multiple motif finding [2]. Buhler and Tompa adopted the EM method in their PROJECTION algorithm in the motif refining step [3]. The following description of the EM algorithm is based on [2].

```
1. randomly choose $k$ positions from the $l$ positions of the motif;
2. try all $4^k$ possible (partial) strings;
3. use the partial consensus strings $s_p$ to search all the $n$ given strings and find a
   substring that is closest to the partial string from each of the $n$ given strings;
4. reconstruct the center string $s$ based on the $n$ selected substrings of length $l$;
5. repeat 1–4 several times and choose the best result.
```

Fig. 2. A randomized algorithm for consensus pattern

In the EM algorithm, a motif of length-l is represented by a $4 \times l$ matrix. Let a $4 \times l$ matrix W be the initial guess of the motif. $s_i(j)$ denotes the j-th letter in sequence s_i. Here is the standard EM algorithm to refine the motif:

1. For each position j in each sequence s_i, $s_{ij} = s_i(j)s_i(j+1)\dots s_i(j+l-1)$ denotes the l-mer (substring of length l) starting at $s_i(j)$ and ending at $s_i(j+l-1)$. Calculate the likelihood that s_{ij} $(1 \le i \le n,\ 1 \le j \le m-l+1)$ is the occurrence of the motif as follows:
$$P(i,j) = \prod_{x=1}^{l} W(s_{ij}(x), x),$$
where $s_{ij}(x) = s_i(j+x-1)$ is the x-th base in l-mer s_{ij}. In order to avoid zero weights, a fixed small number δ (we use $\delta = 0.1$) is added to every element of W before calculating the likelihoods.
2. For each l-mer s_{ij}, we get a normalized probability from the likelihood.
$$P'(i,j) = \frac{P(i,j)}{\sum_{j=1}^{m-l+1} P(i,j)}.$$
Replace $P(i,j)$ with $P'(i,j)$.
(The normalization guarantees that $\sum_{j=1}^{m-l+1} P'(i,j) = 1$, reflecting the fact that there is exactly one motif occurrence in each sequence.)
3. Re-estimate the (motif) matrix W from all the l-mers as follows:
$$W = \sum_{i=1}^{n} \sum_{j=1}^{m-l+1} W^{ij},$$
where W^{ij} is also a $4 \times l$ matrix, constructed from s_{ij}:
$$W^{ij}(b,x) = \begin{cases} P(i,j) & : \text{ if } b = s_{ij}(x) \\ 0 & : \text{ otherwise.} \end{cases}$$
4. A normalization is applied to W to ensure that the sum of each column in W is 1, i.e.,
$$W'(b,x) = \frac{W(b,x)}{\sum_{b=A,C,G,T} W(b,x)}.$$
Replace W with W'.

5. Steps 1–4 are called a *cycle*. Let W_{q-1} and W_q be the two consecutive matrices produced in cycles $q-1$ and q. If

$$\max |W_q(b,x) - W_{q-1}(b,x)| < \epsilon, \tag{1}$$

then EM stops. Otherwise, goto step 1 and start next cycle.

In step 5, ϵ is a parameter given by the user. We use a relatively large value $\epsilon = 0.05$ such that on average the EM algorithm stops within very few cycles.

Improved EM Algorithm
The algorithm in Section3.1 directly finds a substring that is closest to the guessed motif, whereas the standard EM algorithm in Section 3.2 considers every substring when constructing W. In [24], it was proposed that one can use $\frac{1}{m-l+1}$, the average $P(i,j)$ for $j = 1, 2, \ldots, m-l+1$, as the threshold when constructing W. All terms with value less than $\frac{1}{m-l+1}$ are ignored when constructing W in Step 3. Another improvement is the shifting technique. For the occurrences of motifs with high score, we try to shift the occurrences of the motifs to left and right by one or two positions. It was shown that for the improved EM algorithm, both speed and accuracy are improved significantly [24]. The software MotifDetector in [24] can outperform PROJECTION for long motifs. However, PatternBranching in [21] is much faster than both PROJECTION and MotifDetector in all cases. All the methods mentioned in this subsection cannot handle indels in the occurrences of the motif.

The main reason that we still work on the EM approach is that we have ways to extend the EM algorithms to handle indels and to solve the two groups case.

3.3 Incorporating Indels

In this section, we introduce a method that extend the EM algorithms to handle insertions and deletions.

Recall that the EM algorithm needs to compute W^{ij} for every substring of length l when no indel is allowed.Let k be the maximum total number of insertions and deletions allowed in an occurrence of a motif. When indels are allowed, the matrix for EM becomes $5 \times l$, where we add the space as a letter. We have to consider all length $l+h$ substrings in all the given sequences, where $h = 0, 1, -1, 2, -2, ...k, -k$ is the number of insertions (positive number) and deletions (negative number). For each length $l+h$ substring, we will align it with the $5 \times l$ matrix.

Aligning a length $l+h$ string with a $5 \times l$ matrix
Let $d[i,j]$ be the score of aligning the first i columns in W with the first j letters in the string. $d[i,j]$ can be computed as follows:

$$\begin{aligned} d[i,j] = max\{&d[i-1,j-1] \times w[x,i], \\ &d[i-1,j] \times w[\Delta,i], d[i,j-1] \times \delta\}, \end{aligned} \tag{2}$$

where x is the j-th letter in the string, Δ represents the space (a new letter), and $\delta = 0.1$ is the number that is used to avoid 0-weight in the EM algorithm .

We can compute all the $d[i, j]$'s in a bottom up order. A standard backtracking procedure will give an optimal alignment.

The new EM algorithm allowing indels. The new EM algorithm also contains the 5 steps in Section 3.2. We consider all length-$l + h$ ($h = 0, 1, -1, 2, -2, \ldots, k, -k$) substrings in the given sequences. For a fixed starting position, say, position q, there are $2k+1$ substrings starting at position q and their lengths are $l, l+1, l-1, l+2, l-2, \ldots, l+k$ and $l-k$. For each starting position q, we align the $2k + 1$ substrings with the matrix W and treat the strings in the alignments as new strings of length l with some possibly inserted spaces in the new strings and with some letters in the old strings deleted if they correspond to inserted empty columns in W. We then choose the substring with the biggest score to do Steps 1-5 for the EM algorithm described in Section 3.2. We also apply the threshold and shifting techniques as in the improved EM algorithm here.

Note that, in equation (2), we use the score (product) for EM approach. In our algorithm, we use the profile representation (matrix) for the EM iterations. After EM refining procedure, we convert the matrix into a string (by choosing the letter with the biggest number in each column) and use Hamming distance to choose the substring (from a given sequence) that is closest to the computed motif. The result is better than that of always using the EM score (product of elements in W as in Step 1 of the Em algorithm in Section3.2). A refinement of the Hamming distance is the score $\sum_{i=1}^{l} W(a_i, i)$, where $a_1, a_2, \ldots, a_l$ is a string of length l and W is the matrix for the motif. We found that the result using refinement (of Hamming distance) score to choose the substrings that are closest to W is better than that of using Hamming distance. The complete algorithm is given in Figure 3. The number of times that Steps 1-3 are repeated in the algorithm is denoted as Maxtrials.

```
1. randomly choose $k$ positions from the $l$ positions of the motif;
2. For each of the $4^k$ possible strings of length $k$, a matrix $W$ is formed as
   follows: for column $x$, if position $x$ is among the $k$ selected positions, then set
   $W(b, x) = 1$, where $b$ is the letter at position $x$, and set the other 3 elements
   in column $x$ to be 0; otherwise, set $W(b, x) = 0.25$ for $b = A, C, G, T$.
3. Use the new EM algorithm allowing indels (with $W$ as initial value) to find
   motifs.
4. repeat 1–3 several (Maxtrials) times and choose the best result.
```

Fig. 3. The complete algorithm for the new algorithm allowing indels

3.4 Experiment Results

In this section, we do simulations to illustrate the quality of our algorithm. (All cases are run on an IBM T42 1.5GHz notebook computer.)

Evaluation method

To evaluate the programs, Pevzner and Sze proposed a challenge problem [18], which has been studied by Keich and Pevzner [10,11]. We randomly generate n ($n = 20$) sequences of length m($m = 600$). Given a center string s of length l, for each of the n random sequences, we randomly choose d positions for s, randomly mutate the d letters from s and implant the mutated copy of s into the random sequence. The problem here is to find the implanted pattern. The pattern thus implanted is called an (l, d)-*pattern*. We will use this model to do simulations and test our algorithms. To incorporate indels, d here is simply the total number of insertions, deletions, and mutations.

Simulation Results

Table 1 shows the results, where the motifs for 15 sequences have no insertion and deletion, and the motifs of the other 5 sequences have one deletion and $d-1$ mutations. We can see that the accuracy (the probability that the algorithm finds the implanted patterns) for each case is very close to that of the algorithm in [24], where no insertion and deletion is involved.

Table 2 shows the results, where the motifs for 10 sequences have no insertion and deletion, the motifs of 5 sequences have one deletion and $d-1$ mutations, and the motifs of the other 5 sequences have one insertion and $d-1$ mutations. We can see that the running time increases significantly and accuracy in many cases is slightly worse than that in Table 1.

Table 1. The results for one deletion

Problem	Maxtrials	Accuracy	Time(s)
(11,2)	1	100%	12
(13,3)	4	90%	66
(13,3)	6	95%	101
(15,4)	8	90%	216
(17,4)	1	100%	12
(20,5)	1	100%	22
(25,5)	1	100%	19

Table 2. The mixed case, where the motifs in 5 sequences have one insertion and the motifs in the other 5 sequences have one deletion

Problem	Maxtrials	Accuracy	Time(s)
(11,2)	1	90%	28
(13,3)	1	90%	58
(15,3)	1	100%	17
(15,4)	1	60%	122
(17,4)	1	100%	20
(20,5)	1	100%	54
(25,5)	1	100%	37

Table 3 shows the results for the case, where the motifs in 5 sequences have one deletion, the motifs in the other 5 sequences have two deletions and the motifs in the remaining 10 sequences have no indel. Table 4 shows the results for the case, where the motifs in 5 sequences have one insertion, the motifs in the other 5 sequences have two insertions and the motifs in the remaining 10 sequences have no indel. The results in Table 4 are slightly better than those in Table 3. The reason might be that the case in Table 4 needs to insert two columns in the matrix for the motif, whereas the case in Table 3 needs to insert two spaces in the motif sequences. Table 5 describes the mixed case, where there

Table 3. The motifs in 5 sequences have one deletion. The motifs in the other 5 sequences have two deletions.

Problem	Maxtrials	Accuracy	Time(s)
(11,2)	1	90%	77
(15,3)	1	100%	18
(15,4)	1	50%	210
(17,4)	1	100%	93
(20,5)	1	100%	20
(25,5)	1	100%	27

Table 4. The motifs in 5 sequences have one insertion, the motifs in the other 5 sequences have two insertions

Problem	Maxtrials	Accuracy	Time(s)
(11,2)	1	100%	15
(15,3)	1	100%	36
(15,4)	1	70%	185
(17,4)	1	100%	37
(20,5)	1	100%	49
(25,5)	1	100%	62

are motifs with two indels. We use the following method to randomly generate the motifs. Among the 20 implanted motifs, the probabilities that a motif has one insertion, one deletion, two insertions , two deletions , one insertion and one deletion are all 1/8. Thus, the probability that a implanted motif has no indel is 3/8. Overall, the results for this mixed case are still reasonable.

Table 5. 1/8 motifs have one insertion, 1/8 motifs have one deletion, 1/8 motifs have two insertions, 1/8 motifs have two deletions and 1/8 motif have one insertion and one deletion

Problem	Maxtrials	Accuracy	Time(s)
(11,2)	1	90%	186
(13,3)	1	50%	624
(15,3)	1	100%	50
(17,3)	1	100%	62
(17,4)	1	90%	212
(20,4)	1	100%	102
(25,4)	1	90%	401

4 Computing the Distinguishing Substring Selection

In this section, we extend the EM approach to work for the distinguishing substring selection problem, where there are two groups of sequences.

4.1 The Extended EM Algorithm for Two Groups

Let $B = \{s_1, s_2, \dots, s_{n_1}\}$ and $G = \{g_1, g_2, \dots, g_{n_2}\}$ be two groups of sequences. The problem here is that we want to find a motif s of length l such that for each string $s_i \in B$ there exists a length-l substring t_i of s_i with $d(s, t_i) \leq d_b$ and for any length-l substring u_i of $g_i \in G$, $d_H(s, u_i) \geq d_g$. As a first strick, we do not consider indels.

We still follow the 5 basic Steps of the EM algorithm in Section 3.2.

For each length-l substring of s_i (g_k), we compute the likelihood $P(i,j)$ ($P(k,j)$) as in Step 1 of Section 3.2 and compute the normalized likelihood $P'(i,j)$ ($P'(k,j)$) as in Step 2 of Section 3.2. In Step 3, the $4 \times l$ matrix W_{ij} is computed as usual.

$$W^{ij}(b,x) = \begin{cases} P(i,j) & : \text{ if } b = s_{ij}(x) \\ 0 & : \text{ otherwise.} \end{cases}$$

The main difference is that we re-estimate the (motif) matrix W from all the l-mers in both groups as follows:

$$W = \sum_{s_i \in B} \sum_{j=1}^{m-l+1} W^{ij} - \rho \sum_{g_i \in G} \sum_{j \text{ s.t. } P'(i,j) > ave} W^{ij}, \tag{3}$$

where every l-mer from B contributes positively, every l-mer from G with $P'(i,j) > ave$ contributes negatively, ρ is a parameter to control the intensity of the negative contributions and ave is the average $P'(i,j)$ over all l-mers from group B. In fact, we do not consider all the l-mers from group G. We only use those $W^{i,j}$ with $P'(i,j) > ave$ for every $g_i \in G$.

The main idea is that when the motif represented by the matrix W is too close to some l-mer from group G ($P'(i,j) > ave$), we scoop the pattern from the matrix by subtracting the corresponding matrix W^{ij}.

Steps 4 and 5 of the EM algorithm remain the same.

The choice of ρ: The choice of ρ has influence on the performance of the program. We carry experiments to determine its value. We find that when ρ is between 0.1 and 0.4, the program has better performance. So we set ρ to be 0.2 in our program.

The choice of ave: Here ave is set to be the average $P'(i,j)$ over all l-mers from group B. This setting allow us to make $d_g - d_b$ to be large. Considering l-mers in group G with $P'(i,j) > 2ave$ or $P'(i,j) > 3ave$ makes the program faster. Here the users can try to set this parameter to control $d_g - d_b$ and the running time.

4.2 Experiment Results

We have done experiments based on simulation data. In many applications, the length of the motif should be from 18 to 30 [22,23].

Simulation data

We first arbitrarily choose a sequence c_1 of length 600 as the center for group B. To get the center c_2 for group G, we randomly choose 200 positions (allowing repeats) from c_1 and randomly set the letters at those positions to one of the four letters. (Thus, it is possible that some of the letters at those selected positions may remain the same.) After obtaining the two centers c_1 and c_2, we generate two groups of sequences. Each group contains 10 sequences. The sequences in each group is generated as follows: randomly choose 200 positions (allowing repeats) from the center and randomly set the letters at those positions to one of the four letters. The two centers are not included in the two groups of sequences.

AveDisG1 is the average Hamming distance between two sequences in group B. AveDisG2 is the average Hamming distance between two sequences in group G. AveDisG12 is the average Hamming distance between any pair of sequences in different groups. l is the length of the motif. For two groups of sequences, the case is *successful* if our program finds a motif such that for each string $s_i \in B$ there exists a length-l substring t_i of s_i with $d_H(s, t_i) \leq d$ and for any length-l substring u_i of $g_i \in G$, $d_H(s, u_i) > d$. The accuracy is defined as the percentage of successful cases. From Table 6, we can see that it is easy to find a motif that can distinguish the two groups when l is large. Intuitively, it is easy to find a motif that can distinguish the two groups when the distance between the two groups (or the distance between the two centers) is large. Table 7 shows the results when 300 random positions are selected to generate c_2.

Table 6. The results for the two groups, where c_2 is generated by randomly choosing 200 positions (allowing repeats) from c_1 and randomly setting the letters at those positions to one of the four letters. The average Hamming distance between c_1 and c_2 is about 128

AveDisG1	AveDisG2	AveDisG12	l	accuracy
215	215	285	12	95%
215	215	285	13	98%
215	215	285	14	99%
215	215	285	15	100%
215	215	285	20	100%
215	215	285	25	100%
215	215	285	30	100%

Table 7. The results for the two groups, where c_2 is generated by randomly choosing 300 positions (allowing repeats) from c_1 and randomly setting the letters at those positions to one of the four letters. The average Hamming distance between c_1 and c_2 is about 175.

AveDisG1	AveDisG2	AveDisG12	l	accuracy
215	215	310	12	98%
215	215	310	13	100%
215	215	310	14	100%
215	215	310	15	100%
215	215	310	20	100%
215	215	310	25	100%
215	215	310	30	100%

Acknowledgement

The paper is supported by a grant from the Research Grants Council of the Hong Kong Special Administrative Region, China [Project No. CityU 1196/03E].

References

1. Bailey, T., Elkan, C.: Fitting a mixture model by expectation maximization to discover motifs in biopolymers. In: Proceedings of the Second International Conference on Intelligent Systems for Molecular Biology (ISMB-94), pp. 28–36. AAAI Press, Menlo PArk (1994)
2. Bailey, T., Elkan, C.: Unsupervised learning of multiple motifs in biopolymers using expectation maximization. Machine Learning 21, 51–80 (1995)
3. Buhler, J., Tompa, M.: Finding motifs using random projections. Journal of Computational Biology 9, 225–242 (2002)
4. Cardon, L.R., Stormo, G.D.: Expectation maximization algorithm for identifying protein-binding sites with variable lengths from unaligned DNA fragments. J. Mol. Biol. 223, 159–170 (1992)

5. Deng, X., Li, G., Li, Z., Ma, B., Wang, L.: Generic Drug Design without Side Effect. SIAM J on Computing 32(4), 1073–1090 (2003)
6. Dopazo, J., Rodríguez, A., Sáiz, J.C., Sobrino, F.: Design of primers for PCR amplification of highly variable genomes. CABIOS 9, 123–125 (1993)
7. Gusfield, D.: Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology. Cambridge University Press, Cambridge (1997)
8. Hertz, G., Stormo, G.: Identification of consensus patterns in unaligned DNA and protein sequences: a large-deviation statistical basis for penalizing gaps. In: Proc. 3rd Intl Conf. Bioinformatics and Genome Research, pp. 201–216 (1995)
9. Hu, Y.-J.H: Finding subtle motifs with variable gaps in unaligned DNA sequences. Computer Methods and Programs in Biomedicine 70, 11–20 (2003)
10. Keich, U., Pevzner, P.: Finding motifs in the twilight zone. Bioinformatics 18, 1374–1381 (2002a)
11. Keich, U., Pevzner, P.: Subtle motifs: defining the limits of motif finding algorithms. Bioinformatics 18, 1382–1390 (2002b)
12. Lanctot, K., Li, M., Ma, B., Wang, S., Zhang, L.: Distinguishing string selection problems. In: Proc. 10th ACM-SIAM Symp. on Discrete Algorithms, pp. 633–642. Also to appear in Information and Computation
13. Lawrence, C., Reilly, A.: An expectation maximization (EM) algorithm for the identification and characterization of common sites in unaligned biopolymer sequences. Proteins 7, 41–51 (1990)
14. Li, M., Ma, B., Wang, L.: Finding Similar Regions in Many Strings. In: Proceedings of the Thirty-first Annual ACM Symposium on Theory of Computing, Atlanta, pp. 473–482 (1999)
15. Li, M., Ma, B., Wang, L.: Finding Similar Regions in Many Sequences (special issue for Thirty-first Annual ACM Symposium on Theory of Computing). J. Comput. Syst. Sci. 65, 73–96 (2002a)
16. Li, M., Ma, B., Wang, L.: On the closest string and substring problems. JACM 49(2), 157–171 (2002b)
17. Lucas, K., Busch, M., Mössinger, S., Thompson, J.A.: An improved microcomputer program for finding gene- or gene family-specific oligonucleotides suitable as primers for polymerase chain reactions or as probes, CABIOS,vol. 7, pp. 525–529 (1991)
18. Pevzner, P., Sze, S.: Combinatorial approaches to finding subtle signals in DNA sequences. In: Proceedings of the 8th International Conference on Intelligent Systems for Molecular Biology. pp. 269–278 (2000)
19. Proutski, V., Holme, E.C.: Primer Master: a new program for the design and analysis of PCR primers. CABIOS 12, 253–255 (1996)
20. Stormo, G.: Consensus patterns in DNA. In: Doolittle, R.F.(ed.) Molecular evolution: computer analysis of protein and nucleic acid sequences, Methods in Enzymology, vol. 183, pp. 211–221 (1990)
21. Price, A., Ramabhadran, S., Pevzner, P.: Finding Subtle Motifs by Branching from Sample Strings, Bioinformatics 19, 149–155 (2003)
22. Keller, G.H., Manak, M.M.: DNA Probes, Stockton Press, p. 12 (1989)
23. McPearson, M.J., Quirke, M.J., Taylor, G.R: PCR A Practical Approach, p. 8. Oxford University Press, New York (1991)
24. Wang, L., Dong, L., Fan, H.: Randomized Algorithms for Motif Detection. In: Fleischer, R., Trippen, G. (eds.) ISAAC 2004. LNCS, vol. 3341, pp. 884–895. Springer, Heidelberg (2004)
25. Waterman, M., Arratia, R., Galas, E.: Pattern recognition in several sequences:consenus and alignment. Bull. Math. Biol. 46, 515–527 (1984)

Algorithms for Computing the Longest Parameterized Common Subsequence

Costas S. Iliopoulos[1,⋆], Marcin Kubica[2,⋆⋆], M. Sohel Rahman[1,⋆⋆⋆,†], and Tomasz Waleń[2,⋆⋆]

[1] Algorithm Design Group
Department of Computer Science, Kings College London,
Strand, London WC2R 2LS, England
{csi,sohel}@dcs.kcl.ac.uk
http://www.dcs.kcl.ac.uk/adg

[2] Institute of Informatics, Warsaw University
Banacha 2, 02-097 Warszawa, Poland
{kubica,walen}@mimuw.edu.pl

Abstract. In this paper, we revisit the classic and well-studied *longest common subsequence* (LCS) problem and study some new variants, first introduced and studied by Rahman and Iliopoulos [Algorithms for Computing Variants of the Longest Common Subsequence Problem, ISAAC 2006]. Here we define a generalization of these variants, the *longest parameterized common subsequence* (LPCS) problem, and show how to solve it in $O(n^2)$ and $O(n+\mathcal{R}\log n)$ time. Furthermore, we show how to compute two variants of LCS, RELAG and RIFIG in $O(n+\mathcal{R})$ time.

1 Introduction

This paper deals with some new interesting variants of the classic and well-studied *longest common subsequence* (LCS) problem. The longest common subsequence between strings can be defined as the maximum number of common (identical) symbols between them, while preserving the order of those symbols. Therefore, the LCS problem, can be seen as an investigation for the *"closeness"* among strings. Apart from being interesting from pure theoretical point of view, the LCS problem has extensive applications in diverse areas of computer science and bioinformatics.

The LCS problem for $k > 2$ strings was first shown to be NP-hard [13] and later proved to be hard to be approximated [11]. In fact, Jiang and Li, in [11], showed that there exists a constant $\delta > 0$, such that, if LCS problem for more

⋆ Supported by EPSRC and Royal Society grants.

⋆⋆ Partially supported by the Polish Ministry of Science and Higher Education under grant N20600432/0806.

⋆⋆⋆ Supported by the Commonwealth Scholarship Commission in the UK under the Commonwealth Scholarship and Fellowship Plan (CSFP).

† On Leave from Department of CSE, BUET, Dhaka-1000, Bangladesh.

than 2 strings has a polynomial time approximate algorithm with performance ratio n^δ, then $P = NP$. The restricted but probably the more studied problem that deals with two strings has been studied extensively [7,8,9,14,16,15,17,19]. The classic dynamic programming solution to LCS problem (for two strings), invented by Wagner and Fischer [19], has $O(n^2)$ worst case running time, where each given string is of length n. Masek and Paterson [14] improved this algorithm using the "Four-Russians" technique [1] to reduce the worst case running time[1] to $O(n^2/\log n)$. Since then, not much improvement in terms of n can be found in the literature. However, several algorithms exist with complexities depending on other parameters. For example, Myers in [16] and Nakatsu et al. in [17] presented an $O(nD)$ algorithm where the parameter D is the simple Levenshtein distance between the two given strings [12]. Another interesting and perhaps more relevant parameter for this problem is $\mathcal{R}$, where $\mathcal{R}$ is the total number of ordered pairs of positions at which the two strings match. Hunt and Szymanski [9] presented an algorithm running in $O((\mathcal{R}+n)\log n)$. They have also cited applications where $\mathcal{R} \sim n$ and thereby claimed that for these applications the algorithm would run in $O(n \log n)$ time. For a comprehensive comparison of the well-known algorithms for LCS problem and study of their behaviour in various application environments the readers are referred to [4].

Very recently, Rahman and Iliopoulos [18,10] introduced the notion of gap-constraints in LCS and presented efficient algorithms to solve the resulting variants. The motivations and applications of their work basically come from Computational Molecular Biology and are discussed in [10]. In this paper, we revisit those variants of LCS and present improved algorithms to solve them. The results we present in this paper are summarized in the following table.

PROBLEM	INPUT	Results in [18,10]	Our Results
LPCS	X, Y, K_1, K_2 and D	–	$O(\min(n^2, n+\mathcal{R}\log n))$
FIG	X, Y and K	$O(n^2 + \mathcal{R}\log\log n)$	
ELAG	X, Y, K_1 and K_2	$O(n^2 + \mathcal{R}\log\log n)$	
RIFIG	X, Y and K	$O(n^2)$	$O(n+\mathcal{R})$
RELAG	X, Y, K_1 and K_2	$O(n^2 + \mathcal{R}(K_2 - K_1))$	

The rest of the paper is organized as follows. In Section 2, we present all the definitions and notations required to present the new algorithms. In Sections 3 to 5, we present new improved algorithms for all the variants discussed in this paper. Finally, we briefly conclude in Section 6.

2 Preliminaries

Suppose we are given two sequences $X[1] \dots X[n]$ and $Y[1] \dots Y[n]$. A subsequence $S[1..r] = S[1]\,S[2] \dots S[r]$ of X is obtained by deleting $[0, n-r]$ symbols from X. A common subsequence of two strings X and Y, denoted $CS(X, Y)$, is

[1] Employing different techniques, the same worst case bound was achieved in [6]. In particular, for most texts, the achieved time complexity in [6] is $O(hn^2/\log n)$, where $h \le 1$ is the entropy of the text.

a subsequence common to both X and Y. The longest common subsequence of X and Y, denoted $LCS(X, Y)$, is a common subsequence of maximum length. In LCS problem, given two sequences, X and Y, we want to find out a longest common subsequence of X and Y.

In [18,10], Rahman and Iliopoulos introduced a number of new variants of the classical LCS problem, namely FIG, ELAG, RIFIG and RELAG problems. These new variants were due to the introduction of the notion of gap constraints in LCS problem. In this section we set up a new 'parameterized' model for the LCS problem, giving us a more general way to incorporate all the variants of it. In the rest of this section we define this new notion of parameterized common subsequence and define the variants of LCS mentioned above in light of the new framework. We remark that both the definitions of [18,10] and this paper are equivalent.

Let X and Y be sequences of length n. We will say, that the sequence C is the *parameterized common subsequence* $PCS(X, Y, K_1, K_2, D)$ (for $1 \leq K_1 \leq K_2 \leq n$, $0 \leq D \leq n$) if there exist such sequences P and Q, that:

– $|C| = |P| = |Q|$; we will denote the length of these sequences by l,
– P and Q are increasing sequences of indices from 1 to n, that is: $1 \leq P[i], Q[i] \leq n$ (for $1 \leq i \leq l$), and $P[i] < P[i+1]$ and $Q[i] < Q[i+1]$ (for $1 \leq i < l$),
– the sequence of elements from X indexed by P and the sequence of elements from Y indexed by Q are both equal C, that is: $C[i] = X[P[i]] = Y[Q[i]]$ (for $1 \leq i \leq l$),
– additionally, P and Q satisfy the following two constraints:
  - $K_1 \leq P[i+1] - P[i], Q[i+1] - Q[i] \leq K_2$, and
  - $|(P[i+1] - P[i]) - (Q[i+1] - Q[i])| \leq D$, for $1 \leq i < l$.

By $LPCS(X, Y, K_1, K_2, D)$ (*longest parameterized common subsequence*) we will denote the problem of finding the maximum length of the common subsequence C of X and Y[2]. Now we can define the problems introduced in [18,10] using our new framework as follows.

– $FIG(X, Y, K)$ (LCS problem with fixed gap) denotes the problem $LPCS(X, Y, 1, K, n)$,
– $ELAG(X, Y, K_1, K_2)$ (LCS problem with elastic gap) denotes the problem $LPCS(X, Y, K_1, K_2, n)$,
– $RIFIG(X, Y, K)$ (LCS problem with rigid fixed gap) denotes the problem $LPCS(X, Y, 1, K, 0)$,
– $RELAG(X, Y, K_1, K_2)$ (LCS problem with rigid elastic gap) denotes the problem $LPCS(X, Y, K_1, K_2, 0)$.

Let us denote by $\mathcal{R}$ the total number of ordered pairs of positions at which X and Y match, that is the size of the set $M = \{(i, j) : X[i] = Y[j], 1 \leq i, j \leq n\}$.

[2] The parameterization presented here should not be mistaken with one that can be found in the parameterized edit distance problem [2,3].

3 An $O(n^2)$ Algorithm for LPCS

The *LPCS*(X, Y, K_1, K_2, D) problem can be solved in polynomial time using dynamic programming. Let us denote by $T[i,j]$ maximum length of such a *PCS*$(X[1,\ldots,i], Y[1,\ldots,j], K_1, K_2, D)$, that ends at $X[i] = Y[j]$. Using the problem definition, we can formulate the following equation:

$$T[i,j] = \begin{cases} 0 & \text{if } X[i] \neq Y[j] \\ 1 + \max(\{0\} \cup \{T[x,y] : (x,y) \in Z_{i-K_1,j-K_1}\}) & \text{if } X[i] = Y[j] \end{cases}$$

where $Z_{i,j}$ denotes the set:

$$Z_{i,j} = \{(x,y) : 0 \leq i-x, j-y \leq K_2 - K_1, |(i-x)-(j-y)| \leq D\}$$

We will show, how to compute array T in $O(n^2)$ time using dynamic programming. But first we have to introduce an auxiliary data-structure.

3.1 Max-Queue

Max-queue is a kind of priority queue that provides the maximum of the last L elements put into the queue (for a fixed L). It provides the following operations:

– `init`(Q, L) initializes Q as the empty queue and fixes the parameter L,
– `insert`(Q, x) inserts x into Q,
– `max`(Q) is the maximum from the last L elements put into Q (assuming, that Q is not empty).

Max-queue is implemented as a pair $Q = (q, c)$, where q is a two-linked queue of pairs, and c is a counter indexing consecutive insertions. Each element x inserted into the queue is represented by pair (i, x), where i is its index. The q contains only pairs containing these elements, that (at some moment) can be returned as answer to `max` query. These elements form a decreasing sequence. The empty queue is represented by $(\emptyset, 0)$. Insertion can be implemented as shown in Algorithm 1.

Algorithm 1. `insert`$(Q = (q, c), x)$

```
  /* Remove such pairs $(i, val)$, that $val \leq x$.                */
1 while not $empty(q)$ and $q.tail.val \leq x$ do RemoveLast($q$);
2 $c++$;
3 Enqueue($q$, $\{index = c, val = x\}$);
  /* Remove such pairs $(i, val)$, that $i \leq c - L$.              */
4 while $q.head.index \leq c - L$ do RemoveFirst($q$);
```

The amortized running time of `insert` is $O(1)$. The `max` query simply returns $q.head.val$ (or 0 if the q is empty).

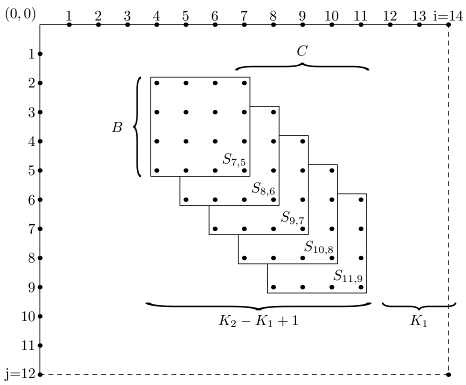


Fig. 1. Set $Z_{i-K_1,j-K_1}$, for $i = 14$, $j = 12$, $K_1 = 3$, $K_2 = 10$, and $D = 3$

3.2 The Algorithm

The set $Z_{i,j}$ has a complicated shape. It is easier to view it as a sum of squares. Let $B = \min(K_2 - K_1, D) + 1$, $C = K_2 - K_1 - B + 2$, and $S_{i,j} = \{(i - x, i - y) : 0 \le x, y < B\}$. Then, we can define $Z_{i,j}$ as:

$$Z_{i,j} = \bigcup_{0 \le k < C} S_{i-k,j-k}$$

To compute T, we will use three auxiliary arrays:

– $R[i, j] = \max_{k=0,\ldots,B-1} T[i - k, j]$,
– $S[i, j] = \max_{k=0,\ldots,B-1} R[i, j - k] = \max_{(x,y)\in S_{i,j}} T[x, y]$,
– $P[i, j] = \max_{k=0,\ldots,C-1} S[i - k, j - k] = \max_{(x,y)\in Z_{i,j}} T[x, y]$.

Now, $T[i, j]$ can be expressed as:

$$T[i,j] = \begin{cases} 0 & \text{if } X[i] \neq Y[j] \\ 1 + P[i - K_1, j - K_1] & \text{if } X[i] = Y[j] \end{cases}$$

We will compute all the arrays using dynamic programming, filling them row by row. We will also use max-queues to compute respective maxima — while computing elements of these arrays indexed by i and j:

– Q_R is a max-queue containing information about $T[i - B + 1 \ldots i, j]$,
– $Q_S[i]$ is a max-queue containing information about $R[i, j - B + 1 \ldots j]$,
– $Q_P[i - j]$ is a max-queue containing information about $S[i, j], \ldots, S[i - C + 1, j - C + 1]$.

The value $LPCS(X, Y, K_1, K_2, D)$ is computed in the *GlobalMax* variable. Please note, that arrays R, S and P are introduced for the clarity of the algorithm and can be removed.

The actual longest parameterized common subsequence can be reconstructed in $O(n)$ time. Since the operations on max-queues run in $O(1)$ amortized time, total time complexity of the above algorithm is $O(n^2)$.

Algorithm 2. AlgLPCS-1

```
Initialize $R[i,j] = S[i,j] = GlobalMax = 0$;
for $i = 1$ to $n$ do Init($Q_S[i], B$);
for $i = -n+1$ to $n-1$ do Init($Q_P[i], C$);
for $j = 1$ to $n$ do
    Init($Q_R, B$);
    for $i = 1$ to $n$ do
        if $X[i] = Y[j]$ then
            $T[i,j] = P[i-K_1, j-K_1] + 1$;
            $GlobalMax = \max(GlobalMax, T[i,j])$;
        else
            $T[i,j] = 0$;
        insert($Q_R, T[i,j]$);   $R[i,j] =$ max($Q_R$);
        insert($Q_S[i], R[i,j]$);   $S[i,j] =$ max($Q_S[i]$);
        insert($Q_P[i-j], S[i,j]$);   $P[i,j] =$ max($Q_P[i-j]$);
```

4 An $O(n + \mathcal{R} \log n)$ Algorithm for FIG and ELAG

For special cases, where $\mathcal{R} = o(n^2/\log n)$, we can solve ELAG (and FIG) problems more efficiently, namely in $O(n + \mathcal{R}\log n)$ running time. In order to do it, instead of computing the whole array T, we should compute only these entries that correspond to matches from the set M. For $(i,j) \notin M$ we have $T[i,j] = 0$, and for $(i,j) \in M$ we have:

$$T[i,j] = 1 + \max\left(\{0\} \cup \left\{ \begin{array}{r} T[x,y] : (x,y) \in M, i-K_2 \le x \le i-K_1, \\ j-K_2 \le y \le j-K_1 \end{array} \right\}\right)$$

We will require data structures D and Q providing the following operations:

– *Insert*(i,j,p) — inserts element (i,j) with priority p,
– *Remove*(i,j) — removes element (i,j),
– *Priority*(i,j) — returns priority of the element (i,j), or 0 if it is not present,
– *Max*(l,r) — returns maximum priority among such elements (i,j), that $l \le i \le r$ (or 0 if there are no such elements).

We can implement the above operations in $O(\log n)$ time, using balanced search trees (such, as AVL or Red-Black trees [5]) and enriching each node with a maximum priority in the corresponding subtree.

Let $M' = \{(i-K_1, j-K_1) : (i,j) \in M\}$ and $B = K_2 - K_1 + 1$. The algorithm scans the consecutive rows of M and M'. While scanning, we keep in D information about elements from the last B rows of $T[i,j]$. Hence, when processing row j, we have:

$$D.Max(i-B+1, i) = \max_{\substack{(x,y)\in M, \\ i-K_2+K_1 \le x \le i, \\ j-K_2+K_1 \le y \le j}} T[x,y]$$

Algorithm 3. AlgELAG

```
1  Compute sets $M$ and $M' = \{(i - K_1, j - K_1) : (i, j) \in M\}$;
2  Initialize $D = \emptyset$, $Q = \emptyset$, $GlobalMax = 0$, $B = K_2 - K_1 + 1$;
3  for $j$=1 to $n$ do
4      // Remove row $j - B$ from $D$.
5      for $(x, y = j - B) \in M$ do  $D.Remove(x, y)$;
6      // Insert row $j$ into $D$
7      for $(x, j) \in M$ do
8          $Len = 1 + Q.Priority(x - K_1, j - K_1)$;
9          $D.Insert(x, j, Len)$;
10         $GlobalMax = \max(GlobalMax, Len)$;
11     for $(x, j) \in M'$ do
12         $Q.Insert((x, j),\ D.Max(x - B + 1, x))$;
```

However, instead of storing values $T[i, j]$ in an array, we store in Q pairs (i, j) (for $(i, j) \in M'$) with priorities $\max\{T[x, y] : 0 \le i - x, j - x < B\}$.

The value $ELAG(X, Y, K_1, K_2)$ is computed in the *GlobalMax* variable. The actual longest common subsequence with elastic gap can be reconstructed in $O(n)$ time. Clearly, the overall time complexity of the above algorithm is $O(n + \mathcal{R} \log n)$.

The above algorithm can be extended to solve the LPCS problem in $O(n + \mathcal{R} \log n)$ running time.

5 An $O(n + \mathcal{R})$ Algorithm for RIFIG and RELAG

To solve the RELAG and RIFIG problems, we need to observe, that they can be reduced to $O(n)$ independent 1-dimensional problems. Since RIFIG is a special case of RELAG, for $K_1 = 1$, we will focus on the latter one. Please recall, that RELAG is equivalent to $LPCS(X, Y, K_1, K_2, 0)$.

Let $T[i, j]$ denote the maximum length of such a $PCS(X[1, \ldots, i], Y[1, \ldots, j], K_1, K_2, 0)$, that includes $X[i]$ and $Y[j]$. $T[i, j]$ can be computed using the following formula:

$$T[i, j] = \begin{cases} 0 & \text{if } X[i] \ne Y[j] \\ 1 + \max\{T[i - p, j - p] : K_1 \le p \le K_2\} & \text{if } X[i] = Y[j] \end{cases}$$

Let $M' = \{(i - K_1, j - K_1) : (i, j) \in M\}$, and $R[i, j] = \max\{T[i - p, j - p] : 0 \le p \le K_2 - K_1\}$. It is enough to calculate values $T[i, j]$ only for $(i, j) \in M$, and they can be expressed as: $T[i, j] = 1 + R[i - K_1, j - K_1]$. Hence, it is enough to calculate values $R[i, j]$ only for $(i, j) \in M'$.

We will use a slightly extended version of max-queue (cf. Section 3.1). Since we process only indices $(i, j) \in M \cup M'$, we must be able to insert elements with specified indices. Let $Q = (q, c)$ be a max-queue. Operation `insert-ind`(Q, x, i) first sets the counter c to $i - 1$, and then calls `insert`(Q, x). The amortized running time of such an operation is still constant, since each element is inserted and removed once.

We will process each diagonal separately. For each $d = 1, \ldots, 2n-1$ we scan points $(i,j) \in M \cup M'$ laying on the d-th diagonal (i.e. such that $n+i-j=d$), in order of increasing i. We will use a max-queue Q to compute values $R[i,j]$, but we will store them in a one dimensional vector $P[i]$, $P[i] = R[i, n+i-d]$. When processing $(i,j) \in M$, we can compute $T[i,j]$, as $T[i,j] = R[i-K_1, j-K_1]+1 = P[i-K_1]+1$. When processing $(i,j) \in M'$ we can compute $P[i]$, as $P[i] = R[i,j] = \mathtt{max}(Q)$. The details are shown in Algorithm 4: AlgRELAG.

Algorithm 4. AlgRELAG

```
1  Compute sets $M$ and $M' = \{(i-K_1, j-K_1) : (i,j) \in M\}$;
2  Initialize $GlobalMax = 0$, $P[i] = 0$, for $1 \le i \le n$;
3  for $d=1$ to $2n-1$ do
4      init($Q, K_2 - K_1 + 1$) ;                    /* extended Max-Queue */
5      foreach $(i,j) \in M \cup M'$ and $n+i-j=d$ (in order of increasing $i$) do
6          if $(i,j) \in M$ then
7              $Len = P[i-K_1] + 1$;
8              insert-ind($Q, Len, i$);
9              $GlobalMax = \max(GlobalMax, Len)$;
10         if $(i,j) \in M'$ then
11             insert-ind($Q, 0, i$) ;   /* phony insert, to clean up the $Q$ */
12             $P[i] = \mathtt{max}(Q)$;
13     // Clean modified cells of array $P$  foreach $(i,j) \in M'$ and $n+i-j=d$
       do  $P[i] = 0$;
```

Sets M and M' can be computed and sorted in $O(n+\mathcal{R})$ time (assuming, that the alphabet is composed of polynomially bounded integer numbers). While scanning the diagonals, we have to process $|M \cup M'|$ positions, each requiring constant amortized time. Hence, the overall time complexity of the AlgRELAG is $O(n+\mathcal{R})$.

6 Conclusions

We have studied variants of the well-known LCS problem: FIG, ELAG, RIFIG and RELAG, presented in [18,10]. These problems can be seen as special cases of the more general LPCS problem, introduced here. We presented an algorithm for solving the LPCS problem in $O(n^2)$ time, that improves the previously known algorithms for FIG, ELAG and RELAG. For special cases, when $\mathcal{R} = o(n^2)$, we have also presented algorithms for RELAG and RIFIG problems running in $O(n+\mathcal{R})$ time, and for FIG and ELAG problems running in $O(n+\mathcal{R}\log n)$ time. The latter one can be extended to solve LPCS problem, without changing its running time.

References

1. Arlazarov, V.L., Dinic, E.A., Kronrod, M.A., Faradzev, I.A.: On economic construction of the transitive closure of a directed graph (english translation). Soviet Math. Dokl. 11, 1209–1210 (1975)
2. Brenda, S.: Baker, B.S.: Parameterized diff. In: Symposium of Discrete Algorithms (SODA), pp. 854–855 (1999)
3. Baker, B.S., Giancarlo, R.: Sparse dynamic programming for longest common subsequence from fragments. Journal of Algorithms 42(2), 231–254 (2002)
4. Bergroth, L., Hakonen, H., Raita, T.: A survey of longest common subsequence algorithms. In: String Processing and Information Retrieval (SPIRE), pp. 39–48. IEEE Computer Society, Los Alamitos (2000)
5. Cormen, T.H., Leiserson, C.E., Rivest, R.L.: Introduction to algorithms. MIT Press, McGraw Hill, Cambridge (1992)
6. Crochemore, M., Landau, G.M., Ziv-Ukelson, M.: A sub-quadratic sequence alignment algorithm for unrestricted cost matrices. In: Symposium of Discrete Algorithms (SODA), pp. 679–688 (2002)
7. Hadlock, F.: Minimum detour methods for string or sequence comparison. Congressus Numerantium 61, 263–274 (1988)
8. Hirschberg, D.S.: Algorithms for the longest common subsequence problem. Journal of ACM 24(4), 664–675 (1977)
9. Hunt, J.W., Szymanski, T.G.: A fast algorithm for computing longest subsequences. Commun. ACM 20(5), 350–353 (1977)
10. Iliopoulos, C.S., Rahman, M.S.: Algorithms for computing variants of the longest common subsequence problem. Theoretical Computer Science (to Appear)
11. Jiang, T., Li, M.: On the approximation of shortest common supersequences and longest common subsequences. SIAM Journal of Computing 24(5), 1122–1139 (1995)
12. Levenshtein, V.I.: Binary codes capable of correcting deletions, insertions, and reversals. Problems in Information Transmission 1, 8–17 (1965)
13. Maier, D.: The complexity of some problems on subsequences and supersequences. Journal of the ACM 25(2), 322–336 (1978)
14. Masek, W.J., Paterson, M.: A faster algorithm computing string edit distances. J. Comput. Syst. Sci. 20(1), 18–31 (1980)
15. Mäkinen, V., Navarro, G., Ukkonen, E.: Transposition invariant string matching. Journal of Algorithms 56, 124–153 (2005)
16. Myers, E.W.: An O(ND) difference algorithm and its variations. Algorithmica 1(2), 251–266 (1986)
17. Nakatsu, N., Kambayashi, Y., Yajima, S.: A longest common subsequence algorithm suitable for similar text strings. Acta Inf. 18, 171–179 (1982)
18. Rahman, M.S., Iliopoulos, C.S.: Algorithms for computing variants of the longest common subsequence problem. In: Asano, T. (ed.) ISAAC 2006. LNCS, vol. 4288, pp. 399–408. Springer, Heidelberg (2006)
19. Wagner, R.A., Fischer, M.J.: The string-to-string correction problem. Journal of ACM 21(1), 168–173 (1974)

Fixed-Parameter Tractability of the Maximum Agreement Supertree Problem*

Extended Abstract

Sylvain Guillemot and Vincent Berry

Equipe *Méthodes et Algorithmes pour la Bioinformatique*, LIRMM,
Univ. Montpellier – CNRS
{sguillem,vberry}@lirmm.fr

Abstract. Given a ground set L of labels and a collection of trees whose leaves are bijectively labelled by some elements of L, the Maximum Agreement Supertree problem (SMAST) is the following: find a tree T on a largest label set $L' \subseteq L$ that homeomorphically contains every input tree restricted to L'. The problem finds applications in several fields, e.g. phylogenetics. In this paper we focus on the parameterized complexity of this NP-hard problem. We consider different combinations of parameters for SMAST as well as particular cases, providing both FPT algorithms and intractability results.

1 Introduction

Motivation. Supertree construction consists in building trees on a large set of labels from smaller trees covering parts of the label set. This task finds application in bioinformatics where trees represent phylogenies, but also in other fields such as databases [1] and data mining [2]. In phylogenetics, the labels are bijectively associated with the leaves of the trees and represent current organisms, while internal nodes represent hypothetical ancestors. The topological information in the input trees consists in the groupings of labels induced by internal nodes, representing related sets of organisms such as species, orders, families, etc. The goal is to build a supertree complying as much as possible with the topological information of the source trees. The task is relatively easy when the input trees agree on the relative positions of the labels. In this case, it is possible to find in polynomial time a supertree that contains any input tree as an induced subtree, hence that incorporates all topological information provided by the data [1]. However, in practice several input trees usually disagree on the position of some leaves with respect to other leaves.

Related work. Some methods aim at producing supertrees incorporating as much input information as possible under the constraint that they do not

* This paper was supported by the *Action incitative* BIOSTIC-LR.

contradict any input tree: they avoid disagreements between the input trees by collapsing some of their edges [3,4] or by excluding some of their leaves, i.e. labels. The Maximum Agreement Supertree (SMAST) method [5,6,7] is apparented to the latter kind. Given a collection $\mathcal{T}$ of k trees of maximum degree d with labels taken in a ground set L of size n, an agreement supertree for $\mathcal{T}$ is a tree T on a subset $L' \subseteq L$ such that each tree of $\mathcal{T}$ restricted to L' is included in T. The SMAST problem consists in finding an agreement supertree containing the maximum number of labels from L. Note that this problem is called MASP in [5].

This problem is NP-hard in general as it generalizes the MAST problem [8]. SMAST remains NP-hard when d is unrestricted for $k \geq 3$ input trees [5] and for trees of degree $d \geq 2$ when k is unrestricted [6]. Moreover, [5,6] have also considered the complement problem, which is a minimization problem where the measure is the number p of labels missing in an agreement supertree. This complement problem can not be approximated in polynomial time within a constant factor, unless P = NP [6]. The corresponding decision problem parameterized in p is W[2]-hard [6].

For $k = 2$ [5,6] showed that SMAST can be solved in polynomial time, by reduction to MAST. For the particular case of $d = 2$, [5] gave an $O(n^{3k^2})$ algorithm for SMAST.

Our results. In this paper, we focus on the particular case where $d = 2$. Note that in phylogenetics, the input trees of SMAST will often be binary as a result of the optimization algorithms used to analyze raw molecular data. We improve on previous results in several ways.

First, we give an algorithm that solves SMAST on k rooted binary trees on a label set of size n in $O((2k)^p kn^2)$ time. This algorithm is only exponential in p, that roughly represents the extent to which the input trees disagree. Thus, the algorithm will be reasonably fast when dealing with collections of trees obtained for genes displaying a low level of conflict. Then, we provide an $O((8n)^k)$ algorithm, independent of p, and significantly improving on the $O(n^{3k^2})$ algorithm of [5]. This algorithm shows that SMAST is tractable for a small number of trees, extending in some sense the previously known results for $k = 2$ trees [5,6]. We also obtain some fixed-parameter intractability results for various combinations of parameters of SMAST.

We then consider SMAST on collections of rooted triples (binary trees on 3 leaves), focusing on the complexity of this variant parameterized in p. Since this problem is equivalent to SMAST in its general setting [6], it is W[2]-hard. However, we show here that an FPT algorithm can be achieved for *complete* collections of rooted triples, i.e., when there is at least one rooted triple for each set of 3 labels in L. This results from the fact that conflicts between the input trees can be circumvented to small sets of labels, leading to $O(4^p n^3)$ and $O(3.12^p + n^4)$ algorithms.

2 Definitions

We consider rooted trees which are bijectively leaf-labelled. Let T be such a tree, we identify its leaf set with its label set, denoted by $L(T)$. The *size* of T is $|T| := |L(T)|$. The node set of T is denoted by $N(T)$, and $r(T)$ stands for the root of T. We use a parenthesized notation for trees: if ℓ is a label, then ℓ also denotes the corresponding leaf-tree; if $T_1, ..., T_k$ are trees, then $(T_1, ..., T_k)$ stands for the tree whose root is connected to the child subtrees $T_1, ..., T_k$.

If x is a node of T, $T(x)$ stands for the subtree of T rooted at x, and $L(x)$ stands for the label set of this subtree. If x, y are two nodes of T, then $x <_T y$ means that x is a descendant of y in T; we denote by $\leq_T$ the non-strict counterpart of the relation $<_T$. The upper bound of two nodes x, y of T w.r.t. $\leq_T$ is called the lowest common ancestor of x, y, and is denoted by $\mathsf{lca}_T(x, y)$. If x, y are two nodes of T s.t. $x <_T y$, denote by $\mathsf{child}_T(x, y)$ the child of y along the path joining y to x in T. If x is an internal node of T, the set of children of x in T is denoted by $\mathsf{children}_T(x)$.

Given a tree T and a label set L, the *restriction* of T to L, denoted by $T|L$, is the tree homeomorphic to the smallest subtree of T connecting leaves of L. Let T, T' be two trees. We say that T *embeds* in T' iff $T = T'|L(T)$. We say that T *agrees* with T' iff $T|L(T') = T'|L(T)$. A *collection* is a family $\mathcal{T} = \{T_1, ..., T_k\}$ of trees, the label set of the collection is $L(\mathcal{T}) = \cup_{i=1}^{k} L(T_i)$. Given a label set L, the *restriction* of $\mathcal{T}$ to L is the collection $\mathcal{T}|L = \{T_1|L, ..., T_k|L\}$. See Figure 1 for an example of a collection. Note that here, names are also given to some nodes for the purpose of referencing them in the text.

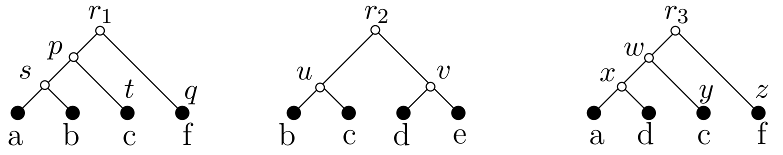


Fig. 1. A collection $\mathcal{T}$ of 3 input trees on the label set $L = \{a, b, c, d, e, f\}$

An *agreement supertree* for $\mathcal{T}$ is a tree S with $L(S) \subseteq L(\mathcal{T})$ and s.t. for each $i \in [k]$, S agrees with T_i. We say that S is a *total agreement supertree* for $\mathcal{T}$ if additionnally $L(S) = L(\mathcal{T})$. The collection $\mathcal{T}$ is *compatible* iff there exists a total agreement supertree for $\mathcal{T}$. A *conflict* between $\mathcal{T}$ is a set $C \subseteq L(\mathcal{T})$ s.t. $\mathcal{T}|C$ is incompatible. For instance, $T = (((a, b), c), (e, f))$ is an agreement supertree for the collection $\mathcal{T}$ of Figure 1, and $C = \{a, b, c, d\}$ is a conflict between $\mathcal{T}$.

Given a collection $\mathcal{T}$, we define $SMAST(\mathcal{T})$ as the set of agreement supertrees for $\mathcal{T}$. The Maximum Agreement Supertree problem (Smast) asks: given a collection $\mathcal{T}$, find an agreement supertree for $\mathcal{T}$ with the largest possible size. Equivalently, it amounts to seek a largest set $L \subseteq L(\mathcal{T})$ s.t. $\mathcal{T}|L$ is compatible. The size of such an *optimal* solution is denoted by $\#SMAST(\mathcal{T})$. We also denote by P-Smast the parameterized version of Smast, which asks: given a collection $\mathcal{T}$ and a parameter p, can $\mathcal{T}$ be made compatible by removing at most p labels?

3 Solving SMAST on Binary Trees

Throughout this section, we consider a fixed collection $\mathcal{T} = \{T_1, ..., T_k\}$ of binary trees, we let n denote the size of the label set $L(\mathcal{T})$ and k the number of trees in $\mathcal{T}$.

If T is a tree, we define $N^{\perp}(T) := N(T) \cup \{\perp\}$. We extend the notation $T(u)$ to $u \in N^{\perp}(T)$, s.t. if $u = \perp$ then $T(u)$ is the empty tree. We extend the relation $\leq_T$ to $N^{\perp}(T)$ s.t. $\perp \leq_T x$ for each $x \in N^{\perp}(T)$.

A *position* in $\mathcal{T}$ is a tuple $\pi = (u_1, ..., u_k)$, where each $u_i \in N^{\perp}(T_i)$. For $i \in [k]$, the ith component of the tuple π is denoted by $\pi[i]$. We set $I(\pi) = \{i \in [k] : \pi[i] \text{ is an internal node of } T_i\}$. We define the *initial position* $\pi_{\top} = (r(T_1), ..., r(T_k))$ and the *final position* $\pi_{\perp} = (\perp, ..., \perp)$.

3.1 Solving SMAST in $O((2k)^p \times kn^2)$ Time

In this section, we describe an algorithm deciding the compatibility of a collection in $O(kn^2)$ time, *and* returning a conflict of size $\leq 2k$ in case of incompatibility. This yields an FPT algorithm for P-SMAST with $O((2k)^p \times kn^2)$ running time.

The compatibility of a collection can be decided by the well-known BUILD algorithm [1,9]. However, in case of incompatibility, this algorithm doesn't provide a conflict, which is required here for the purpose of a bounded search FPT algorithm. Like BUILD, the algorithm presented here builds the supertree using a recursive top-down approach. Each step constructs a graph where the connected components correspond to subtrees of the supertree. Here, we replace the graph used in BUILD with a graph that when connected yields a conflict of size $\leq 2k$, identified thanks to a spanning tree.

We begin with some additional definitions. A position is *reduced* iff no component is a leaf (i.e. each component is either $\perp$ or an internal node). To any position π, we associate a reduced position $\pi\!\downarrow$ by replacing by $\perp$ any component of π that is a leaf. In the following, we will assume that π is a reduced position in $\mathcal{T}$. We set $\mathcal{T}(\pi) := \{T_1(u_1), ..., T_k(u_k)\}$.

We define the graph $G(\mathcal{T}, \pi)$ as follows: (i) its vertex set is $V = \cup_{i \in I(\pi)} \mathsf{children}_{T_i}(\pi[i])$; (ii) two vertices $x, y \in V$ are adjacent iff $L(x) \cap L(y) \neq \emptyset$. In other terms, $G(\mathcal{T}, \pi)$ is the intersection graph of the set system $\{L(x) : x \in V\}$. Given $V' \subseteq V$, we define the successor of π w.r.t. V', denoted by $Succ_{V'}(\pi)$, as the position π' s.t.

- if $\pi[i] = \perp$, then $\pi'[i] = \perp$;
- if $\pi[i]$ is an internal node of T_i, with children v_i, v_i', then either:

$$\begin{cases} \text{if } v_i \in V' \text{ and } v_i' \notin V' \text{ then } \pi'[i] = v_i, \\ \text{if } v_i \notin V' \text{ and } v_i' \in V' \text{ then } \pi'[i] = v_i', \\ \text{if } v_i \in V' \text{ and } v_i' \in V' \text{ then } \pi'[i] = u_i, \\ \text{if } v_i \notin V' \text{ and } v_i' \notin V' \text{ then } \pi'[i] = \perp . \end{cases}$$

We set $succ_{V'}(\pi) = Succ_{V'}(\pi)\!\downarrow$.

To decide the compatibility of $\mathcal{T}$, we use a recursive top-down approach. At a given step of the recursion, we consider a reduced position π in $\mathcal{T}$, and we try to identify two successors π_1, π_2, which are reduced positions corresponding to the child subtrees of an hypothetical agreement supertree for $\mathcal{T}(\pi)$. We then recurse on these successor positions π_1, π_2, until $\pi_\perp$ is reached.

The recursion step proceeds as follows. It constructs the graph $G(\mathcal{T}, \pi)$, and performs a connexity test on this graph. If the graph is not connected, then the connexity test yields a partition of V in two disconnected sets V_1, V_2 (which themselves might contain several connected components); then the successor positions are π_1, π_2, where $\pi_i = succ_{V_i}(\pi)$. The correctness of this step is precisely stated in Lemma 3. If the graph is connected, then the connexity test yields a spanning tree of $G(\mathcal{T}, \pi)$, and we obtain a conflict by choosing, for each edge of the tree, a label present in the intersection of the two corresponding subtrees (Lemma 4).

See Figure 2 for an illustration of this process.

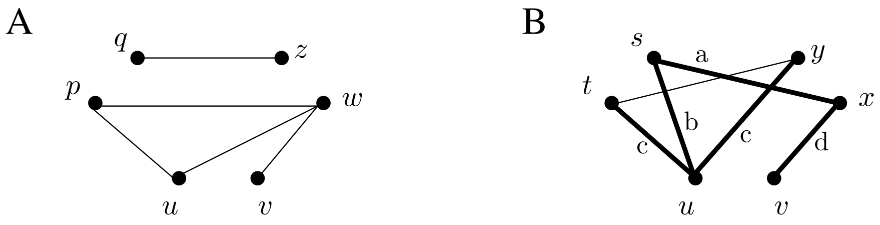


Fig. 2. A. The graph $G(\mathcal{T}, \pi_\top)$ of the position $\pi_\top = (r_1, r_2, r_3)$ for the collection of trees of Figure 1. This graph is disconnected, the two connected components indicate the two successor positions $\pi_1 = (p, r_2, w)$ and $\pi_2 = (q, \perp, z)$ of $\pi_\top$. B. The graph $G(\mathcal{T}, \pi_1)$ is connected. Choosing a spanning tree (bold edges) of the graph and an arbitrary label shared by the two subtrees corresponding to the extremities of each edge of this tree identifies a conflict $C = \{a, b, c, d\}$.

We will need the following notations. Given $V' \subseteq V$, set $L(V') = \cup_{x \in V'} L(x)$. Set $L(\pi) = L(\mathcal{T}(\pi))$. Given $V_1, V_2 \subseteq V$, we say that V_1, V_2 are *connected* iff $G(\mathcal{T}, \pi)$ contains an edge $\{x, y\}$ with $x \in V_1, y \in V_2$; otherwise, V_1, V_2 are said to be *disconnected.*

We will repeatedly use the following simple observations:

Lemma 1. *Let $V' \subseteq V$. Then: $L(Succ_{V'}(\pi)) \subseteq L(V')$.*

Lemma 2. *Two sets $V_1, V_2 \subseteq V$ are connected in $G(\mathcal{T}, \pi)$ iff $L(V_1) \cap L(V_2) \neq \emptyset$.*

Given a position π in $\mathcal{T}$, we say that π is *compatible* iff $\mathcal{T}(\pi)$ is compatible. Note that $\pi_\perp$ is compatible. Observe also that: π is compatible iff $\pi\!\downarrow$ is compatible. Let us now consider a reduced position $\pi \neq \pi_\perp$. Then $|L(\pi)| \geq 2$. We have the following recursive characterization of compatibility for this case:

Lemma 3. *Suppose that $|L(\pi)| \geq 2$. The following are equivalent:*

- *π is compatible;*
- *there exists a partition V_1, V_2 of V s.t. (i) V_1, V_2 are disconnected in $G(\mathcal{T}, \pi)$, (ii) $succ_{V_1}(\pi), succ_{V_2}(\pi)$ are compatible.*

Moreover, if the graph $G(\mathcal{T}, \pi)$ turns out to be connected, a spanning tree of this graph yields a small conflict between $\mathcal{T}$:

Lemma 4. *Suppose that $G(\mathcal{T}, \pi)$ is connected, and let $T = (V, F)$ be a spanning tree of $G(\mathcal{T}, \pi)$. For each edge $e = \{u, v\} \in F$, choose $\ell_e \in L(u) \cap L(v)$. Then $C = \{\ell_e : e \in F\}$ is a conflict between $\mathcal{T}$.*

Lemmas 3 and 4 give rise to an algorithm for deciding the compatibility of a collection, and obtaining a conflict of small size in case of incompatibility:

Theorem 1. *There is an algorithm which, in $O(kn^2)$ time, decides if $\mathcal{T}$ is compatible, or returns a conflict of size $\leq 2k$.*

Proof. We define a procedure IsCompatible(π) which takes as input a reduced position, decides if π is compatible, or returns a conflict of size $\leq 2k$ in case of incompatibility. The procedure is as follows: (i) if $\pi = \pi_\perp$, answer ("*yes*"); (ii) if $\pi \neq \pi_\perp$, test whether $G(\mathcal{T}, \pi)$ is connected:

- If the graph is connected, then let $T = (V, F)$ be a spanning tree of $G(\mathcal{T}, \pi)$, choose $\ell_e \in L(u) \cap L(v)$ for each edge $e = \{u, v\} \in F$, construct $C = \{\ell_e : e \in F\}$, and return ("*no*", C).
- If the graph is not connected, then let V_1, V_2 be any partition of V in two disconnected sets, and construct the positions π_1, π_2 where $\pi_i = succ_{V_i}(\pi)$. Call IsCompatible(π_1), let R_1 be its result; if R_1 = ("*yes*") then call IsCompatible(π_2) and return its result R_2, else return R_1.

To decide if $\mathcal{T}$ is compatible, we simply call IsCompatible($\pi_{\mathcal{T}}\downarrow$).

We now justify the correctness and the running time of the algorithm. The correctness of the procedure IsCompatible follows from lemmas 3 and 4. For the running time, we rely on the fact that using appropriate data structures, we can ensure that a call to IsCompatible takes $O(kn)$ time (see [10] for details). By lemmas 1 and 2, the total number of calls to IsCompatible is $O(n)$, therefore the total running time of the algorithm is $O(kn^2)$. □

The algorithm of Theorem 1 yields a simple FPT algorithm for P-Smast using the bounded search tree technique:

Theorem 2. *The* P-Smast *problem can be solved in $O((2k)^p \times kn^2)$ time.*

Proof. The algorithm constructs a search tree of height $\leq p$, where a node of the search tree at depth i is labelled by a set of labels $X \subseteq L$ s.t. $|X| = i$. At a given node u labelled by a set X, the algorithm determines in $O(kn^2)$ time if $\mathcal{T}|(L \backslash X)$ is compatible, using the procedure of Theorem 1. If the answer is positive, the node is labelled by "success". Otherwise, the algorithm proceeds as follows: if the node is at depth p, then it is labelled by "failure"; if it is at depth

$< p$, then the procedure of Theorem 1 has returned a conflict C of size $\leq 2k$, and for each $x \in C$ a child node of u is added, with label $X \cup \{x\}$. The running time follows easily, since the search tree has height $\leq p$, degree $\leq 2k$, and since each node is processed in $O(kn^2)$ time. $\square$

3.2 Solving Smast in $O((8n)^k)$ Time

In this section, we describe an algorithm which solves Smast in $O((8n)^k)$ time. The algorithm uses dynamic programming, and is somewhat similar in spirit to the algorithm described in [8] for solving Mast on two trees.

For the needs of this section, it is convenient to characterize the agreement relation on trees in terms of *partial embeddings*. Let T, T' be two trees, say that a partial embedding of T into T' is a function $\phi : N(T) \rightarrow N(T') \cup \{\perp\}$ such that:

– for any x leaf of T, we have $\phi(x) = \perp$ if $x \notin L(T')$, or $\phi(x) = x$ otherwise,
– for any x internal node of T with children $u_1, ..., u_p$, let $V = \{j : \phi(u_j) \neq \perp\}$, then (i) either $V = \emptyset$, and $\phi(x) = \perp$, (ii) either $V = \{i\}$ and $\phi(x) = \phi(u_i)$, (iii) or $|V| \geq 2$ and $\phi(u_i) <_T \phi(x)$ for each $i \in V$, and the nodes $\{\mathsf{child}_T(\phi(u_i), \phi(x)) : i \in V\}$ are pairwise distinct.

Then T agrees with T' iff there exists a partial embedding of T into T'.

Let $\mathcal{T}$ be a collection and π a position in $\mathcal{T}$. Let $SMAST(\pi)$ denote the set of trees T s.t. (i) T is an agreement supertree for $\mathcal{T}$, (ii) for each i, the partial embedding $\phi_i : T \rightarrow T_i$ is such that $\phi_i(r(T)) \leq_{T_i} \pi[i]$. We denote by $\#SMAST(\pi)$ the size of a largest tree of $SMAST(\pi)$.

The algorithm computes values $\#SMAST(\pi)$ for each position π using a recurrence relation whose base case is stated in Lemma 6 and general case is stated in Lemma 8. The recurrence relation relies on a partial order $\leq_{\mathcal{T}}$ on positions, which is defined below. Given a position π, $\#SMAST(\pi)$ will be computed from values $\#SMAST(\pi')$ with $\pi' <_{\mathcal{T}} \pi$. At the end of the algorithm, $\#SMAST(\mathcal{T})$ is obtained as $\#SMAST(\pi_\top)$.

We define the relation $\leq_{\mathcal{T}}$ on positions in $\mathcal{T}$ by: $\pi \leq_{\mathcal{T}} \pi'$ iff for each $i \in [k]$, $\pi[i] \leq_{T_i} \pi'[i]$. We denote by $<_{\mathcal{T}}$ its strict counterpart, where $\pi <_{\mathcal{T}} \pi'$ iff for each $i \in [k]$, $\pi[i] \leq_{T_i} \pi'[i]$, and one of these relations is strict. Observe that:

Lemma 5. *If $\pi' \leq_{\mathcal{T}} \pi$, then $SMAST(\pi') \subseteq SMAST(\pi)$.*

The base case of the recurrence corresponds to *terminal* positions: a position π is *terminal* if for each $i \in [k]$, $\pi[i]$ is a leaf or $\perp$. Given $x \in L(\mathcal{T})$, let $Ind(x) = \{i \in [k] : x \in L(T_i)\}$. Given a terminal position π, and given $x \in L(\pi)$, define $Ind(x, \pi) = \{i \in [k] : \pi[i] = x\}$; observe that $Ind(x, \pi) \subseteq Ind(x)$. Say that an element $x \in L(\pi)$ is *maximally present* iff $Ind(x, \pi) = Ind(x)$, and let $P(\pi)$ denote the set of maximally present elements of $L(\pi)$. Then:

Lemma 6. *Suppose that π is terminal. Then: $\#SMAST(\pi) = \big|P(\pi)\big|$.*

We now describe the general case of the recurrence relation, corresponding to nonterminal positions. If π is nonterminal, then $\#SMAST(\pi)$ is computed from two values $\#SMAST_1(\pi), \#SMAST_2(\pi)$.

We first define $\#SMAST_1(\pi)$. Say that a position π' is a *successor* of π iff there exists $i \in [k]$ s.t. $\pi'[i]$ is a child of $\pi[i]$ and $\pi'[j] = \pi[j]$ for each $j \neq i$. Let $S(\pi)$ denote the set of successors of π. Then define:

$$\#SMAST_1(\pi) = \max_{\pi' \in S(\pi)} \#SMAST(\pi'). \tag{1}$$

We now define $\#SMAST_2(\pi)$. Say that a pair (π_1, π_2) of positions is a *decomposition* of π iff (i) $\pi_1 \neq \pi, \pi_2 \neq \pi$ and (ii) for each $i \in [k]$, the following holds:

– either $\pi[i] = \perp$, in which case $\pi_1[i] = \pi_2[i] = \perp$;
– either $\pi[i]$ is a leaf x, in which case we have $\{\pi_1[i], \pi_2[i]\} = \{\perp, x\}$;
– either $\pi[i]$ is an internal node u with two children v, v', in which case we have either $\{\pi_1[i], \pi_2[i]\} = \{\perp, u\}$ or $\{\pi_1[i], \pi_2[i]\} = \{v, v'\}$.

Let $D(\pi)$ denote the set of decompositions of π. Then define:

$$\#SMAST_2(\pi) = \max_{(\pi_1,\pi_2) \in D(\pi)} (\#SMAST(\pi_1) + \#SMAST(\pi_2)). \tag{2}$$

Note that computing the values $\#SMAST_1(\pi)$ and $\#SMAST_2(\pi)$ only involves values $\#SMAST(\pi')$ with $\pi' <_{\mathcal{T}} \pi$, by the following lemma:

Lemma 7

(i) If $\pi' \in S(\pi)$, then $\pi' <_{\mathcal{T}} \pi$;
(ii) If $(\pi_1, \pi_2) \in D(\pi)$ then $\pi_1 <_{\mathcal{T}} \pi$ and $\pi_2 <_{\mathcal{T}} \pi$.

We are now ready to state the relation for nonterminal positions:

Lemma 8. *Suppose that π is not terminal. Then:*
$\#SMAST(\pi) = \max(\#SMAST_1(\pi), \#SMAST_2(\pi))$.

Proof. We first prove that $\#SMAST_1(\pi) \leq \#SMAST(\pi)$. Let $S \in SMAST(\pi')$ for some $\pi' \in S(\pi)$, s.t. $|S|$ is maximal. Since $\pi' <_{\mathcal{T}} \pi$ by Lemma 7, we have $S \in SMAST(\pi)$ by Lemma 5, and the result follows.

We now prove that $\#SMAST_2(\pi) \leq \#SMAST(\pi)$. Let $(\pi_1, \pi_2) \in D(\pi)$, and let S_1, S_2 s.t. $S_j \in SMAST(\pi_i)$, $|S_j|$ maximal. If one of the S_j's is empty, say S_1, then $\#SMAST(\pi_1) = 0$, and we obtain $\#SMAST_2(\pi) = |S_2| = \#SMAST(\pi_2) \leq \#SMAST(\pi)$ by lemmas 5 and 7. Suppose now that S_1, S_2 are not empty. For $j \in \{1, 2\}$, since $S_j \in SMAST(\pi_j)$, there exists partial embeddings $\phi_{j,i} : S_j \to T_i$ s.t. $\phi_{j,i}(r(S_i)) \leq_{T_i} \pi_j[i]$ for each $i \in [k]$. Let $S = (S_1, S_2)$, we claim that $S \in SMAST(\pi)$. Indeed, define $\phi_i : S \to T_i$ as follows. Set $\phi_i(x) = \phi_{j,i}(x)$ if x is a node of S_j, and $\phi_i(x) = \mathsf{lca}_{T_i}(\phi_{1,i}(r(S_1)), \phi_{2,i}(r(S_2))$ if x is the root of S. Then: (i) $L(S_1) \cap L(S_2) = \emptyset$, hence S is well-defined, (ii)

ϕ_i is a partial embedding of S into T_i, (iii) $\phi_i(r(S)) \leq_{T_i} \pi[i]$. We conclude that $\#SMAST_2(\pi) = |S_1| + |S_2| = |S| \leq \#SMAST(\pi)$.

Finally, we show that $\#SMAST(\pi) \leq \max(\#SMAST_1(\pi), \#SMAST_2(\pi))$. Let $S \in SMAST(\pi)$ s.t. $|S|$ is maximal. Then there exists partial embeddings $\phi_i : S \to T_i$ s.t. $\phi_i(r(S)) \leq_{T_i} \pi[i]$ for each $i \in [k]$. Let $u_i = \phi_i(r(S))$ for each i. We consider two cases.

First case: there exists $i \in [k]$ s.t. $u_i <_{T_i} \pi[i]$. This case holds in particular if $|S| \leq 1$. Define π' from π by setting the ith component to $\mathsf{child}_{T_i}(u_i, \pi[i])$, then $\pi' \in S(\pi)$. We verify that $S \in SMAST(\pi')$: indeed, ϕ_i is a partial embedding of S into T_i s.t. $\phi_i(r(S)) \leq_{T_j} \pi'[j]$ for each j. We conclude that $|S| = \#SMAST(\pi) \leq \#SMAST(\pi') \leq \#SMAST_1(\pi)$.

Second case: $u_i = \pi[i]$ for each $i \in [k]$. In this case, we have $|S| \geq 2$, hence $S = (S_1, S_2)$. Let u be the root of S, let v_i be the root of S_i in S, then $\pi = (\phi_1(u), ..., \phi_k(u))$. For $j \in \{1, 2\}$, define π_j as follows: given $i \in [k]$, (i) if $\phi_i(v_j) = \phi_i(u)$, set $\pi_j[i] = \phi_i(u)$, (ii) if $\phi_i(v_j) = \perp$, set $\pi_j[i] = \perp$, (iii) if $\phi_i(v_j) <_{T_i} \phi_i(u)$, set $\pi_j[i] = \mathsf{child}_{T_i}(\phi_i(v_j), \phi_i(u))$. Then $(\pi_1, \pi_2) \in D(\pi)$. We now show that $S_j \in SMAST(\pi_j)$: indeed, ϕ_i is a partial embedding of S_j into T_i, and by definition of π_j we have $\phi_i(r(S_j)) \leq_{T_i} \pi_j[i]$ for each $i \in [k]$. We conclude that $|S| = \#SMAST(\pi) = |S_1| + |S_2| \leq \#SMAST(\pi_1) + \#SMAST(\pi_2) \leq \#SMAST_2(\pi)$. □

Lemmas 6 and 8 yield an algorithm for computing $\#SMAST(\mathcal{T})$:

Theorem 3. *$\#SMAST(\mathcal{T})$ can be computed in $O((8n)^k)$ time.*

Proof. Using dynamic programming, the algorithm computes the values $\#SMAST(\pi)$ for each position π, using the recurrence relations stated in lemmas 6 and 8. The correctness of the algorithm follows from the lemmas, and the termination of the algorithm is ensured by Lemma 7 and the fact that $<_\mathcal{T}$ is an order relation on positions in $\mathcal{T}$.

We now consider the space and time requirements for the algorithm. First observe that the number of positions π in $\mathcal{T}$ is $\leq (2n)^k$: indeed, a component $\pi[i]$ has $\leq 2n$ possible values (one of the $\leq 2n-1$ nodes of T_i, or the value $\perp$). It follows that the space complexity is $O((2n)^k)$. We claim that the time complexity is $O((8n)^k)$. Indeed, consider the time required to compute $\#SMAST(\pi)$, assuming that the values $\#SMAST(\pi')$ for $\pi' <_\mathcal{T} \pi$ are available. Testing if π is terminal requires $O(k)$ time. If π is terminal, computing $\left|P(\pi)\right|$ takes $O(k)$ time. If π is nonterminal, then we need to compute $\#SMAST_1(\pi)$ and $\#SMAST_2(\pi)$, which respectively require $O(k)$ and $O(4^k)$ time. Thus, $\#SMAST(\pi)$ is computed in $O(4^k)$ time, hence the total running time of the algorithm is $O((8n)^k)$. □

3.3 Hardness Results

The parameterized complexity of the SMAST problem on binary trees is considered w.r.t. the following parameters: k denotes the number of input trees, l denotes an upper bound on the maximum size of the input trees, p (resp. q) denotes an upper (resp. lower) bound on the number of labels to remove (resp.

conserve) in order to obtain compatibility of the collection. Our complexity results for several combinations of the parameters are summarized in Theorem 4:

Theorem 4. *We have the following hardness results for* SMAST*:*

Parameters	*Complexity of* SMAST
q	W[1]*-complete (even for* $l = 3$*)*
q, k	W[1]*-complete*
p	W[2]*-hard (even for* $l = 3$*)*
k, p	*FPT by a* $O((2k)^p \times kn^2)$ *time algorithm*
k	XNL*-hard, solvable in* $O((8n)^k)$ *time*

The proof of the third result can be found in [6]. Other results are proved in [10].

4 Solving Smast on Complete Collection of Triples

A *rooted triple* (or *triple* for short) is a binary tree T s.t. $|L(T)| = 3$. A *collection of triples* is a collection $\mathcal{R} = \{t_1, ..., t_k\}$ where each t_i is a triple. $\mathcal{R}$ is *complete* iff each set of three labels in $L(\mathcal{R})$ is present in at least one t_i. To a binary tree T, we associate a complete collection of triples $rt(T)$ formed by the triples t_i which embed in T. For a complete collection $\mathcal{R}$, we say that $\mathcal{R}$ is *treelike* iff there exists a tree T s.t. $\mathcal{R} = rt(T)$; then we say that $\mathcal{R}$ *displays* T.

Let P-SMAST-CR denote the restriction of P-SMAST to complete collections of triples. We can show that non-treelike collections have conflicts of size ≤ 4, a result similar to that known on quartets [11]. This allows to solve P-SMAST-CR in $O(n^4 + 3.12^p)$ time by reduction to 4-HITTING SET (using a $O^*(3.12^p)$ algorithm for 4-HITTING SET described in [12]), and also in $O(4^p n^4)$ time by bounded search (see, e.g. [13]). In the following, we describe a faster algorithm with $O(4^p n^3)$ running time. We first present an algorithm to decide treelikeness in linear $O(n^3)$ time (Proposition 1 and Theorem 5).

Proposition 1. *There is an algorithm* INSERT-LABEL-OR-FIND-CONFLICT $(\mathcal{R}, X, x, T)$ *which takes as input a complete collection of triples* $\mathcal{R}$*, a set* $X \subseteq L(\mathcal{R})$*, an element* $x \in L(\mathcal{R}) \backslash X$ *and a tree* T *s.t.* $\mathcal{R}|X$ *displays* T*, and in* $O(n^2)$ *time decides if* $\mathcal{R}' = \mathcal{R}|(X \cup \{x\})$ *is treelike. Additionally, the algorithm returns the tree* T' *displayed by* $\mathcal{R}'$ *in case of positive answer, or returns a conflict* C *between* $\mathcal{R}'$ *with* $|C| \leq 4$ *in case of negative answer.*

Proof. In a first step, the algorithm checks whether $\mathcal{R}$ contains two different triples on the same set of three labels x, ℓ, ℓ'. In such a case, they form a conflict of size 3 which is then returned by the algorithm.

If no such conflict is found, the algorithm proceeds to a second step during which it determines for each internal node u of T, the relative subtree in which u would accept to insert x: its left subtree (L), its right subtree (R), or the subtree *above* it (A), namely the part of the tree that is not below u. To that aim, the algorithm checks that the triples x, ℓ, ℓ', with ℓ, ℓ' labels under u in T, all indicate the same subtree relative to u. More formally, let v, v' be the two children of u. An *u-fork* is a pair $\{\ell, \ell'\}$ where $\ell \in L(v), \ell' \in L(v')$. Each u-fork $\{\ell, \ell'\}$ gives an opinion $o_{\ell,\ell'}$ on the positioning of x w.r.t. u in T, where $o_{\ell,\ell'}$ is computed from $\mathcal{R}$ as follows: if $\ell x|\ell' \in \mathcal{R}$ then $o_{\ell,\ell'}$ is set to L, if $\ell' x|\ell \in \mathcal{R}$ then $o_{\ell,\ell'}$ is set to R, otherwise, $\ell\ell'|x \in \mathcal{R}$ and $o_{\ell,\ell'}$ is set to A. The algorithm considers each internal node u in turn and computes the opinions $o_{\ell,\ell'}$ of the u-forks $\{\ell, \ell'\}$. If two u-forks indicate a different subtree for x, then the algorithm easily identifies a conflict. In such a case, it can be shown that there exist ℓ, ℓ_1, ℓ_2 s.t. $o_{\ell,\ell_1} \neq o_{\ell,\ell_2}$ (or $o_{\ell_1,\ell} \neq o_{\ell_2,\ell}$), in which case $C = \{x, \ell_1, \ell_2, \ell\}$ is a conflict, which is then returned by the algorithm. Otherwise, all u-forks indicate the same subtree for x, and the opinion of u, denoted o_u is defined to be this direction (L, R or A).

In a third step, the algorithm checks that the opinions of the different nodes u in T consistently indicate a single position to insert x in T. The opinions are compatible iff for each edge u, v of T with u above v, we have: (i) if v is the left child of u, then $o_u = R \Rightarrow o_v = A$, (ii) if v is the right child of u, then $o_u = L \Rightarrow o_v = A$, (iii) if v is a child of u, then $o_u = A \Rightarrow o_v = A$. If one pair of nodes u, v does not meet the above requirements, then by considering $\{\ell, \ell'\}$ v-fork and $\{\ell, \ell''\}$ u-fork, we obtain a conflict $C = \{x, \ell, \ell', \ell''\}$. Otherwise, consider the sets of nodes u s.t. $o_u \neq A$, they form a (possibly empty) path in T starting at the root and ending at a node v. Then $\mathcal{R}|(X \cup \{x\})$ is treelike, and displays the tree obtained from T by inserting x above v, which is returned by the algorithm.

We now justify the running time of the algorithm. The first step trivially takes $O(n^2)$ time. Consider the second step. Given a node u, let F_u be the set of u-forks, then an internal node u is processed in time $O(|F_u|)$. Therefore, the time required by the second step is $\sum_u O(|F_u|) = O(n^2)$. Now consider the third step. The algorithm checks that for each edge u, v of T, Conditions (i)-(ii)-(iii) hold: for a given edge, checking the conditions or finding a conflict is done in constant time, hence the time required by this step is $O(n)$. It follows that the total time required by the algorithm is $O(n^2)$. □

Theorem 5. *There is an algorithm* Find-Tree-Or-Conflict($\mathcal{R}$) *which takes a complete collection of triples $\mathcal{R}$, and in $O(n^3)$ time decides if $\mathcal{R}$ is treelike, returns a tree T displayed by $\mathcal{R}$ in case of positive answer, or a conflict C between $\mathcal{R}$ with $|C| \leq 4$ in case of negative answer.*

Proof. We use the procedure Insert-Label-Or-Find-Conflict to decide treelikeness as follows. We iteratively insert each label, starting from an empty tree, until: (i) either every label has been inserted, in which case the collection is treelike and the displayed tree is returned, (ii) or a conflict is found and returned. □

Using bounded search, we obtain:

Theorem 6. *The* P-Smast-CR *problem can be solved in $O(4^p n^3)$ time.*

References

1. Aho, A.V., Sagiv, Y., Szymanski, T.G., Ullman, J.D.: Inferring a tree from lowest common ancestors with an application to the optimization of relational expressions. SIAM Journal on Computing 10(3), 405–421 (1981)
2. Xia, Y., Yang, Y.: Mining closed and maximal frequent subtrees from databases of labeled rooted trees. IEEE Transactions on Knowledge and Data Engineering 17(2), 190–202 (2005)
3. Gordon, A.G.: Consensus supertrees: the synthesis of rooted trees containing overlapping sets of labelled leaves. Journal of Classification 3, 335–348 (1986)
4. Ranwez, V., Berry, V., Criscuolo, A., Guillemot, S., Douzery, E.: Vote or veto: desirable properties for supertree methods. submitted to syst. biol. LIRMM (2007)
5. Jansson, J., Ng, J.H.K., Sadakane, K., Sung, W.K.: Rooted Maximum Agreement Supertrees. Algorithmica 4(43), 293–307 (2005)
6. Berry, V., Nicolas, F.: Maximum agreement and compatible supertrees. Journal of Discrete Algorithms (in press 2007)
7. Kao, M.Y.: Encyclopedia of algorithms (2007) http://refworks.springer.com/algorithms/
8. Steel, M., Warnow, T.: Kaikoura tree theorems: computing the maximum agreement subtree. Information Processing Letters 48(2), 77–82 (1993)
9. Henzinger, M., King, V., Warnow, T.: Constructing a Tree from Homeomorphic Subtrees, with Applications to Computational Evolutionary Biology. Algorithmica 24(1), 1–13 (1999)
10. Guillemot, S., Berry, V.: Fixed-parameter tractability of the maximum agreement supertree problem. Technical report, LIRMM (2007)
11. Bandelt, H., Dress, A.: Reconstructing the shape of a tree from observed dissimilarity data. Advances in Applied Mathematics 7, 309–343 (1986)
12. Fernau, H.: Parameterized algorithmics: A graph-theoretic approach. Habilitationsschrift, Universität Tübingen, Germany (2005)
13. Gramm, J., Niedermeier, R.: A fixed-parameter algorithm for minimum quartet inconsistency. Journal of Computer and System Sciences 67(4), 723–741 (2003)

Two-Dimensional Range Minimum Queries*

Amihood Amir[1], Johannes Fischer[2], and Moshe Lewenstein[1]

[1] Computer Science Department,
Bar Ilan University,
Ramat Gan 52900, Israel
{moshe,amir}@cs.biu.ac.il

[2] Ludwig-Maximilians-Universität München,
Institut für Informatik,
Amalienstr. 17, D-80333 München
Johannes.Fischer@bio.ifi.lmu.de

Abstract. We consider the two-dimensional Range Minimum Query problem: for a static $(m \times n)$-matrix of size $N = mn$ which may be preprocessed, answer on-line queries of the form "where is the position of a minimum element in an axis-parallel rectangle?". Unlike the one-dimensional version of this problem which can be solved in provably optimal time and space, the higher-dimensional case has received much less attention. The only result we are aware of is due to Gabow, Bentley and Tarjan [1], who solve the problem in $O(N \log N)$ preprocessing time and space and $O(\log N)$ query time. We present a class of algorithms which can solve the 2-dimensional RMQ-problem with $O(kN)$ additional space, $O(N \log^{[k+1]} N)$ preprocessing time and $O(1)$ query time for any $k > 1$, where $\log^{[k+1]}$ denotes the iterated application of $k+1$ logarithms. The solution converges towards an algorithm with $O(N \log^* N)$ preprocessing time and space and $O(1)$ query time. All these algorithms are significant improvements over the previous results: query time is optimal, preprocessing time is quasi-linear in the input size, and space is linear. While this paper is of theoretical nature, we believe that our algorithms will turn out to have applications in different fields of computer science, e.g., in computational biology.

1 Introduction

One of the most basic problems in computer science is finding the minimum (or maximum) of a list of numbers. An elegant and interesting dynamic version of this problem is the *Range Minimum Query (RMQ) problem*. The popular one dimensional version of this problem is defined as follows:

INPUT: An array $A[1..n]$ of natural numbers.
We seek to preprocess the array A in a manner that yields efficient solutions to the following queries:

* The second author was partially funded by the German Research Foundation (DFG, Bioinformatics Initiative).

QUERY: Given $1 \leq i \leq j \leq n$, output an index k, $i \leq k \leq j$, such that $A[k] \leq A[\ell]$, $i \leq \ell \leq j$.

Clearly, with $O(n^2)$ preprocessing time, one can answer such queries in constant time. Answering queries in time $O(j-i)$ needs no preprocessing. The surprising news is that linear time preprocessing can still yield constant time query solutions. The trek to this result was long. Harel and Tarjan [2] showed how to solve the *Lowest Common Ancestor (LCA)* problem with a linear time preprocessing and constant time queries. The LCA problem has as its input a tree. The query gives two nodes in the tree and requests the lowest common ancestor of the two nodes. It turns out that constructing a Cartesian tree of the array A and seeking the LCA of two indices, gives the minimum in the range between them [1].

The Harel-Tarjan algorithm was simplified by Schieber and Vishkin [3] and then by Berkman et al. [4] who presented optimal work parallel algorithms for the LCA problem. The parallelism mechanism was eliminated and a simple serial algorithm was presented by Bender and Farach-Colton [5]. In all above papers, there was an interplay between the LCA and the RMQ problems. Fischer and Heun [6] presented the first algorithm for the RMQ problem with linear preprocessing time, optimal $2n + o(n)$ bits of additional space, and constant query time that makes no use of the LCA algorithm. In fact, LCA can then be solved by doing RMQ on the array of levels of the tree's inorder tree traversal. This last result gave another beautiful motivation to the naturally elegant RMQ problem.

The problem of finding the minimum number in a given range is by no means restricted to one dimension. In this paper, we investigate the two-dimensional case. Consider an $(m \times n)$-matrix of $N = mn$ numbers. One may be interested in preprocessing it so that queries seeking the minimum in a given rectangle can be answered efficiently. Gabow, Bentley and Tarjan [1] solve the problem in $O(N \log N)$ preprocessing time and space and $O(\log N)$ query time.

We present a class of algorithms which can solve the 2-dimensional RMQ-problem with $O(kN)$ additional space, $O(N \log^{[k+1]} N)$ preprocessing time and $O(1)$ query time for any $k > 1$. The solution converges towards an algorithm with $O(N \log^* N)$ preprocessing time and space and $O(1)$ query time.

2 Preliminaries

Let us first give some general definitions. By $\log n$ we mean the binary logarithm of n, and $\log^{[k]} n$ denotes the k-th iterated logarithm of n, i.e. $\log^{[k]} n = \log\log\ldots\log n$, where there are k log's. Further, $\log^* n$ is the usual *iterated logarithm* of n, i.e., $\log^* n = \min\{k : \log^{[k]} n \leq 1\}$. For natural numbers $l \leq r$, the notation $[l : r]$ stands for the set $\{l, l+1, \ldots, r\}$.

Now let us formally define the problem which is the issue of this paper. We are given a 2-dimensional array $A[0 : m-1][0 : n-1]$ of size $m \times n$. We wish to preprocess A such that queries asking for the position of the minimal element in an axis-parallel rectangle (denoted by $\textsc{rmq}(y_1, x_1, y_2, x_2)$ for *range* *minimum query*) can be answered efficiently. More formally, $\text{RMQ}(y_1, x_1, y_2, x_2) = \arg\min_{(y,x)\in[y_1:y_2]\times[x_1:x_2]}\{A[y][x]\}$. Throughout this paper, let $N = mn$ denote the size of the input.

3 Methods

For simplicity, assume that the input array is a square, i.e., we have $m = n$ and $N = n^2$. The reader can verify that this assumption is not necessary for the validity of our algorithm. Further, because the query time will be constant throughout this section, we do not always explicitly mention this fact.

We first give a high-level overview of the algorithm (see also Fig. 1). The idea is to cover the input array with grids of decreasing widths $s_1, s_2, \ldots$, thus dividing the array into blocks of decreasing size. For each grid of a certain width, we preprocess the array such that queries which *cross* the grid of a certain width s_k, but no grid of width $s_{k'}$ for $k' < k$, can be answered in constant time. Each such preprocessing will use $O(N)$ space and $O(N)$ time to construct. E.g., query q_1 in Fig. 1 will be answered on level 1 because it crosses the grid with width s_1, whereas q_2 will be answered on level 3. If the query rectangle does not cross any of the grids (e.g., q_3 in Fig. 1), we solve it by having precomputed *all* queries inside such small blocks which we call *microblocks.* If the size of these microblocks is constant, this constitutes no extra (asymptotic) time for preprocessing (leading to the $\log^*$-solution); otherwise we have to employ sorting of the blocks for a constant time preprocessing, leading to the $O(N \log^{[k+1]} N)$ preprocessing time. The details are as follows.

3.1 A General Trick for Query Precomputation

Assume we want to answer in $O(1)$ time all queries $\text{RMQ}(y_1, x_1, y_1 + l_y, x_1 + l_x)$ for y_1 taken from a certain subset of indices $Y \subseteq [0 : n-1]$ (and likewise x_1), and certain query lengths $l_y \in L_y = [1 : |L_y|]$ ($l_x \in L_x$). It suffices to precompute the answers for query rectangles whose side lengths are a power of 2; i.e., precompute $\text{RMQ}(y_1, x_1, y_1 + l_y, x_1 + l_x)$ for all $y_1 \in Y, x_1 \in X$, $l_y \in \{2^1, 2^2, 2^3, \ldots, |L_y|\}$, and $l_x \in \{2^1, 2^2, 2^3, \ldots, |L_x|\}$ and store the results in a table. These precomputations can be done in optimal time using dynamic programming, similar to the one-dimensional case [6]. The reason why precomputing these queries is enough is given by the simple fact that *all* queries can be answered by decomposing them into 4 different rectangles whose side lengths is a power of 2; see Fig. 2. Note the similarity to the *sparse table* algorithm for the 1-dimensional solution [5]. We denote by $g(|Y|, |X|, |L_y|, |L_x|) := |Y| \cdot |X| \cdot \log |L_y| \cdot \log |L_x|$ the space occupied by the resulting table for this kind of preprocessing.

Note how this idea already yields an RMQ-algorithm with $O(N \log^2 n)$ preprocessing time and space and $O(1)$ query time: simply perform the above preprocessing for $X, Y, L_x, L_y = [1 : n]$; the space needed is then $g(n, n, n, n) = N \log^2 n$.

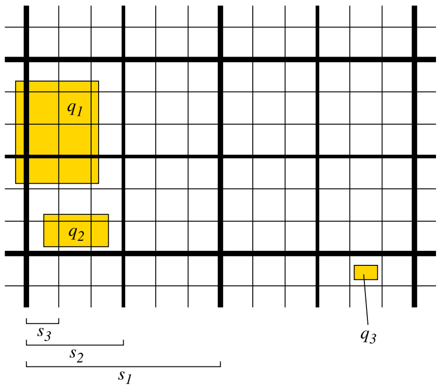


Fig. 1. Covering the input array with grids of different width. q_1, q_2, q_3 denote queries.

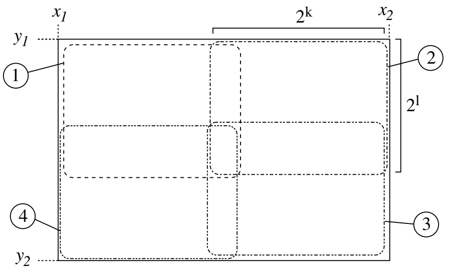


Fig. 2. Decomposing a query rectangle $[y_1 : y_2] \times [x_1 : x_2]$ into four equal-sized overlapping rectangles whose side lengths are a power of two. Taking the position of where the overall minimum occurs is the answer to the query.

3.2 $O(N)$ Preprocessing of the First Level

We now present a preprocessing to answer all queries which cross the grid for width $s := s_1 := \log n$. The array is partitioned into blocks of size $s \times s$. Then a query can be decomposed into at most 9 different sub-queries, as seen in Fig. 3. Query number 1 exactly spans over several blocks in both x- and y-direction. Queries 2–5 span over several blocks in one direction, but not in the other direction. Queries 6–9 lie completely inside one block (but meet at least one of the four block "boundaries").

Next, we show how to preprocess A such that all queries 1–9 can be answered in constant time. Taking the position where the overall minimum occurs is the final result.

Queries of type 1. We apply the idea from Sect. 3.1 on the set $Y = X = L_y = L_x = \{0, s, 2s, \ldots, n/s\}$; i.e., we precompute $\text{RMQ}(ys, xs, (y + 2^k)s, (x + 2^l)s)$ for all $x, y \in \{0, \ldots, n/s\}$ and all $k, l \in \{0, \ldots, \log(n/s)\}$. The results are stored in a table of size $g(n/s, n/s, n/s, n/s) \leq n/s \cdot n/s \cdot \log n \cdot \log n = O(N)$. As usual, queries of this type are then answered by selecting the minimum of at most 4 overlapping precomputed queries.

Queries of type 2–5. We just show how to handle queries 2 and 4; the ideas for 3 and 5 are similar. Note that unlike Fig. 3 suggests, such queries are not guaranteed to share an upper or lower edge with the grid; the general appearance of these queries can be seen in Fig. 4. So the task is to answer all queries $\text{RMQ}(y_1, x_1 s, y_1 + l_y, x_1 s + l_x)$ for all $y_1 \in \{0, \ldots, n-1\}$, $x_1 \in \{0, \ldots, n/s\}$, $l_y \in \{1, \ldots, s\}$ and $l_x \in \{s, 2s, \ldots, n/s\}$. It is easy to verify that simply applying the trick from Sect. 3.1 would result in super-linear space; we therefore have to introduce another "preprocessing layer" by bundling $s' := s^2$ cells into one superblock. Then divide the type 2 (or 4)-query into a query that spans over

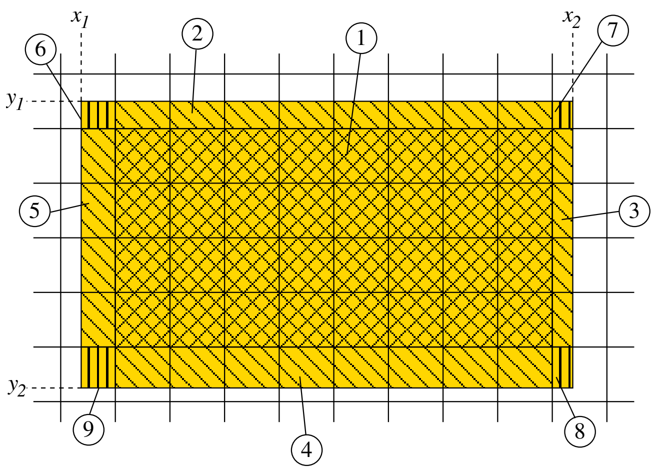


Fig. 3. Decomposing a query rectangle $[y_1 : y_2] \times [x_1 : x_2]$ into at most 9 sub-queries

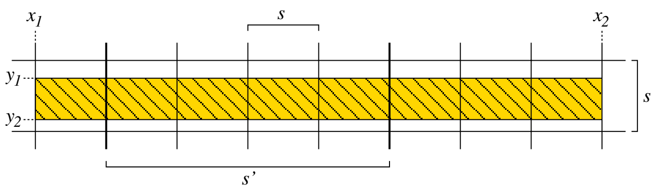


Fig. 4. Queries spanning over more than one block in x-direction, but not in the y-direction. Sub-queries 2 and 4 from Fig. 3 are special cases of these.

several superblocks, and at most two queries that span over several blocks, but *not* over a superblock. All three such queries are handled with the usual idea; this means that the space needed for the superblock-queries is $g(n, n/s', s, n/s') = n \cdot n/\log^2 n \cdot \log\log n \cdot \log(n/\log^2 n) \leq n^2 \log\log n/\log n = O(N)$. The space for the block-queries is $g(n, n/s, s, s'/s) = n \cdot n/\log n \cdot \log\log n \cdot \log\log n = O(N)$.

Queries of type 6–9. Again, unlike Fig. 3 suggests, it is *not* sufficient to precompute queries that have a common border with *two* edges of a block; we also have to precompute queries that share an edge with just *one* block-edge. (E.g., imagine the query in Fig. 4 were shifted slightly to the left. Then there would be a part of the query in the block to the very left which only touches the right border of the block.) We just show how to solve queries that share a border with the upper edge of a block (see Fig. 5); these structures have to be duplicated for the other three edges. This means that we want to answer $\text{RMQ}(y_1 s, x_1, y_1 s + l_y, x_1 + l_x)$ for all $y_1 \in \{0, \ldots, n/s\}$, $x_1 \in \{0, \ldots, n-1\}$, and $l_y, l_x \in \{1, \ldots, s\}$. In this case, the idea from Sect. 3.1 can be applied directly, leading to a space consumption of $g(n/s, n, s, s) = n/\log n \cdot n \cdot \log\log n \cdot \log\log n = O(N)$.

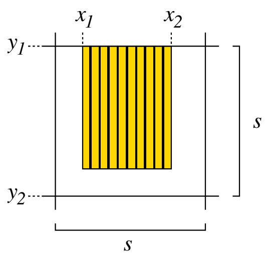


Fig. 5. Queries lying completely within a block, but sharing the upper edge with it. Sub-queries 8 and 9 from Fig. 3 are special cases of these.

3.3 Recursive Partitioning

We are left with the task to answer RMQs which lie completely inside one of the blocks with side length $s = \log n$. We now offer two recursion strategies which yield the $\log^{[k]}$- and $\log^*$-algorithms that have been promised before.

The first idea is to recurse at least one more time into the blocks, and thereafter precompute all queries which lie completely in one of the microblocks. To be precise, we take each of the $(n/s)^2$ resulting blocks from Sect. 3.2 and prepare them with the same method. Then the resulting blocks have side length $s_2 := \log^{[2]} n$. Continue this process until the resulting blocks have side length $s_k = \log^{[k]} n$ for some fixed $k > 1$. As each level needs $O(N)$ space, the resulting space is $O(Nk)$. We now show that already for $k = 2$ we can precompute all queries inside of microblocks in $O(N)$ space and $O(N \log^{[k+1]} N)$ time. We denote by $S := s_k^2$ the size of the microblocks (i.e., the number of elements one microblock contains).

The idea is to precompute all RMQs for all permutations of $[1 : S]$ and look up the result for a certain query block in the right place in this precomputed table. To do so, assign a *type* to each microblock $[y_1 : y_1 + s_k - 1] \times [x_1 : x_1 + s_k - 1]$ in A as follows: (conceptually) write the elements from the microblock row-wise into an array B; i.e., $B[1..S] = A[y_1][x_1..x_1 + s_k - 1] \dots A[y_1 + s_k - 1][x_1..x_1 + s_k - 1]$. Then stably-sort B to obtain a permutation π of $\{1, \dots, S\}$ s.th. $B[\pi_1] \le B[\pi_2] \le \dots \le B[\pi_S]$. The index of π in an enumeration of all permutations of $\{1, \dots, S\}$ is the microblock-type. As there are N/S blocks of size $S = s_k^2$ to be sorted, this takes a total of $O(N/S \times S \log S) = O(N \log^{[k+1]} n)$ time.[1]

The reason for assigning the same type to microblocks whose elements are in the same order can be seen by the following (obvious) lemma:

Lemma 1. *Let A_1 and A_2 two arrays that have the same relative order π. Then* $\text{RMQ}_{A_1}(y_1, x_1, x_2, y_2) = \text{RMQ}_{A_2}(y_1, x_1, x_2, y_2)$ *for all values of* y_1, x_1, y_2, x_2. □

This implies that the following is enough to answer RMQs inside of microblocks: For all permutations π of $\{1, \dots, S\}$, precompute all possible RMQs inside the block

[1] In the special case where the elements from the original array are in the range from 1 to N, we can bucket-sort all blocks simultaneously in $O(N)$ time.

$$\begin{pmatrix} \pi_1 & \cdots & \pi_{s_k} \\ \pi_{s_k+1} & \cdots & \pi_{2s_k} \\ \vdots & \ddots & \vdots \\ \pi_{(s_k-1)s_k+1} & \cdots & \pi_S \end{pmatrix}$$

and store them in a table P (for "precomputed") of size

$$\begin{aligned} S^2(S)! &= s_k^4\sqrt{2\pi s_k^2}\cdot\left(\frac{s_k^2}{e}\right)^{s_k^2}\cdot(1+O(s_k^{-2}))\text{ (by Stirling)} \\ &\le \log^5\log n\cdot\left(\log^2\log n\right)^{\log^2\log n}\cdot(1+O(s_k^{-2}))\text{ (because } k>1) \\ &= (\log\log n)^{2\log^2\log n+5}\cdot(1+O(s_k^{-2})) \\ &= O(N)\ . \end{aligned}$$

The last equation is true because $b^2 = O(2^b)$, so $(2b^2+5)/\log_b 2 \le 2^{b+1}$ for large enough b; exponentiating with 2 yields $b^{2b^2+5} \le 2^{\left(2^{b+1}\right)} = \left(2^{\left(2^b\right)}\right)^2$, which yields the result with $b = \log\log n$. Now to answer a query, simply look up the result in this table.

The second idea is to recurse further into the blocks until the resulting microblocks have constant size; this happens after $O(\log^* n)$ recursive steps. If the resulting micro-blocks have constant size they can be sorted in $O(1)$ time each; and because there are $(n/\log^* n)^2$ microblocks this takes a total of $O(N)$ time. The space consumed by this kind of preprocessing is clearly bounded by $O(N\log^* N)$ due to the number of recursive steps performed.

If we now apply the same recursive steps also for *answering* queries, this yields, of course, $O(\log^{[k+1]} N)$ and $O(\log^* N)$ query time, respectively. The next section shows how to reduce query time to $O(1)$.

3.4 What's Left: How to Find the Right Grid

For both the $\log^{[k]}$- and the $\log^*$-algorithm it remains to show how to determine in $O(1)$ time the grid with the largest width $s_i = \log^{[i]} n$ such that the query block crosses this grid. In other words, we wish to find the *smallest* $1 \le i \le k$ such that the query crosses the grid with width s_i, because at this level the answers have been precomputed and can hence be looked up. We will just show how to do this for the x-direction; the ideas for the y-direction are similar. For simplicity, assume that s_j is a multiple of s_{j+1} for all $1 \le j < k$.

Let $\text{RMQ}(y_1, x_1, y_2, x_2)$ be the given query, and let $l_x := x_2 - x_1 + 1$ denote the side length of the query rectangle in x-direction. Let i be defined such that $s_{i-1} > l_x \ge s_i$. This i can be found in $O(1)$ time by precomputing an array $I[1:n]$ with $I[l] = \min\{k : l \ge s_k\}$; then $i = I[l_x]$. Then the query crosses *at least* one column from the s_i-grid, and *at most* one column from the s_{i-1}-grid. As an example, consider query q_1 in Fig. 6. We have $s_2 > l_x \ge s_3$, and indeed, q_3 crosses a column from the s_3-grid, but no column from the s_2-grid. Likewise, for query q_2 we have $s_2 > l_x \ge s_1$, and it crosses an s_1- and an s_2-column. Let

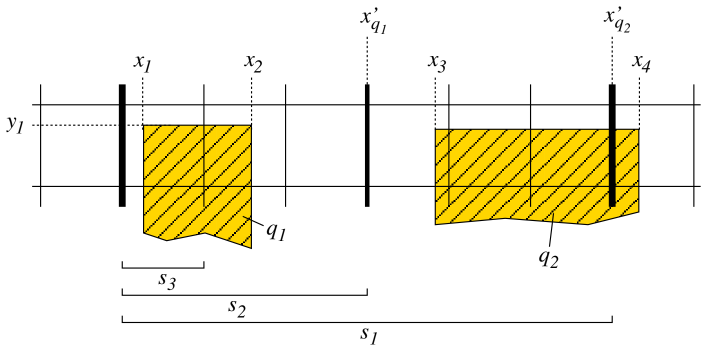


Fig. 6. How to determine the level on which a specific query has been precomputed

$x' := \lfloor \frac{x_2}{s_{i-1}} \rfloor \cdot s_{i-1}$ be the x-coordinate of where this crossing with a column from the s_{i-1}-grid can occur.

Assume first that $x' \notin [x_1 : x_2]$ (as for q_1 in Fig. 6). This means that the s_{i-1}-grid does *not* cross the query rectangle in x-direction; and the same is true for all $i' < i$. So we know for sure that i is the smallest value such that the s_i-grid passes through the query rectangle in x-direction. In this case we are done.

Now assume that $x' \in [x_1 : x_2]$ (as for q_2 in Fig. 6). In other words, the s_{i-1}-grid crosses the query rectangle in x-direction at position x'. In this case we are not yet done, because it could still be that an $s_{i'}$-column with $i' < i - 1$ also passes through x'. To find the smallest such i', define an array $I'[0 : n-1]$ such that $I'[x'] = j$ iff j is the smallest value such that there is a column from the s_j-grid passing through x'. This array can certainly be precomputed during the k rounds of the preprocessing algorithm from the previous sections. As an example, in Fig. 6 it could still be that a column from a different grid (say from a hypothetical s_0-grid) passes through the query rectangle. But as this *must* happen at x'_{q_2}, we find this value at $I'[x'_{q_2}]$.

In total, we do the above for both the x- and y-direction, and look up the query result at the minimum level i from both steps. If, on the other hand, we find that the query rectangle does not cross a grid in any direction, the result can be looked up in table P of precomputed queries.

4 Conclusion

We have seen a class of algorithms which solve the two-dimensional RMQ-problem. While some ideas of our algorithm were similar to the one-dimensional counterpart of the problem, others were completely new, e.g., the idea of iterating the algorithm for k levels, while still handling all queries as if they were on the first level. Note that preprocessing time of our algorithm is not yet linear in the size of the input, as it is the case with the 1D-RMQ (though being *very* close to linear!). We conjecture that achieving linear time is impossible. In particular, we believe that it should be possible to show that there is no such nice relation as the one between the number of different RMQs and the number of different Cartesian Trees in the one-dimensional case [7]. This could mean that ideas such as sorting or recursing become un-avoidable, thereby hinting at a super-linear lower bound.

Although our results are currently more of theoretical nature, we believe that our solution will turn out to have interesting applications in the future. One conceivable application comes from computational biology, where one often wishes to identify minimal (or maximal) numbers in a given region of an alignment tableau [8].

Acknowledgments

J.F. wishes to thank Simon Ginzinger for fruitful discussions on this subject.

References

1. Gabow, H.N., Bentley, J.L., Tarjan, R.E.: Scaling and related techniques for geometry problems. In: Proc. of the ACM Symp. on Theory of Computing, pp. 135–143. ACM Press, New York (1984)
2. Harel, D., Tarjan, R.E.: Fast algorithms for finding nearest common ancestors. SIAM J. Comput. 13(2), 338–355 (1984)
3. Schieber, B., Vishkin, U.: On finding lowest common ancestors: Simplification and parallelization. SIAM J. Comput. 17(6), 1253–1262 (1988)
4. Berkman, O., Breslauer, D., Galil, Z., Schieber, B., Vishkin, U.: Highly parallelizable problems. In: Proc. of the ACM Symp. on Theory of Computing, pp. 309–319. ACM Press, New York (1989)
5. Bender, M.A., Farach-Colton, M., Pemmasani, G., Skiena, S., Sumazin, P.: Lowest common ancestors in trees and directed acyclic graphs. J. Algorithms 57(2), 75–94 (2005)
6. Fischer, J., Heun, V.: A new succinct representation of RMQ-information and improvements in the enhanced suffix array. In: Proc. ESCAPE. LNCS (to appear)
7. Fischer, J., Heun, V.: Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE. In: Lewenstein, M., Valiente, G. (eds.) CPM 2006. LNCS, vol. 4009, pp. 36–48. Springer, Heidelberg (2006)
8. Gusfield, D.: Algorithms on Strings, Trees, and Sequences. Cambridge University Press, Cambridge (1997)

Tiling Periodicity[⋆]

Juhani Karhumäki[1], Yury Lifshits[2], and Wojciech Rytter[3,4]

[1] Turku University, Finland
karhumak@utu.fi
[2] Steklov Institute of Mathematics at St.Petersburg, Russia
yura@logic.pdmi.ras.ru
[3] Department of Mathematics and Informatics, Copernicus University, Torun, Poland
[4] Warsaw University, Poland
rytter@mimuw.edu.pl

Abstract. We contribute to combinatorics and algorithmics of words by introducing new types of periodicities in words. A *tiling period* of a word w is partial word u such that w can be decomposed into several disjoint parallel copies of u, e.g. $a \diamond b$ is a tiling period of $aabb$. We investigate properties of tiling periodicities and design an algorithm working in $O(n \log(n) \log\log(n))$ time which finds a tiling period of minimal size, the number of such periods and their compact representation. The combinatorics of tiling periods differs significantly from that for classical full periods, for example unlike the classical case the same word can have many different primitive tiling periods. We consider also a related new type of periods called in the paper *multi-periods*. As a side product of the paper we solve an open problem posted by T. Harju.

1 Introduction

The number p is a *full period* (period, in short) of a word w of length n iff $p|n$ and $w_i = w_{i+p}$ whenever both sides are defined. Define by *period*(w) the shortest nonzero full period of w.

In this paper we extend the notion of a full period. Namely, we are interested in tilings of a word where the tiles themselves may contain "transparent" letters. A **tiler** (or partial word) is a word over $\Sigma \cup \{\diamond\}$ alphabet, where $\diamond$ is a special transparent (or undefined) letter. In other words, a tiler is a sequence of connected words (blocks) with gaps between the blocks. The **size** of a tiler is the number of defined symbols.

Imagine that we have several copies of a tiler printed on transparencies. Then, this tiler is a period for some word, if we can put these copies into the stack such that they form a single connected word without overlapping of visible letters. Thus, a tiler S is called a **tiling period** of an (ordinary) word T if we can split T

[⋆] Support by grant N 8206039 of the Academy of Finland for the first author, by grants INTAS 04-77-7173 and NSh-8464.2006.1 for the second author, and by grant of the Polish Ministry of Science and Higher Education N 206 004 32/0806 is gratefully acknowledged.

into disjoint parallel copies of S satisfying the following: **(a)** All defined (visible) letters of S-copies match the text letters; **(b)** Every text letter is covered by *exactly one* defined (visible) letter. Similarly we define a tiling period x of a tiler x: x consists of several disjoint copies of y. The word (a tiler) is **primitive** if it has no proper tiling period. The tiling period of x is **minimal** iff it has minimal size.

Example. For example $a\ a\ \diamond \diamond\ b\ b$ is a tiling period of *aaaabbbbaaaabbbb* and $a \diamond b \diamond \diamond \diamond \diamond \diamond c \diamond d$ is a tiling period of *aabbaabbccddccdd*. Both have size 4, the first one is not primitive.

Problem. We investigate basic properties of tiling periodicity and comparing it to the classical notion. We address the following questions: (1) How to enlist all possible tiling periods? (2) Does every word have a unique *primitive* tiling period? (3) How to find all tiling periods of minimal size? (4) How many periods can a word of length n have? (5) What is the relation between the primitive classical and the primitive tiling periods? We have several reasons to be interested in tiling periodicity. Firstly of all, it is a natural generalization of classical notion, i.e. any full period is also a tiling period. Secondly, in the case when a tiling period is relatively small, we can describe a long word by just its tiling period and the length. Hence, we get a new class of words with low Kolmogorov complexity. Tiling periodicity might be useful for the new text compression methods (especially for generalizing run-length encoding). Note here, that the ratio between the "size" of tiling period and the length of classical full period may be arbitrarily small. Next, the notion of tiling periodicity provides a geometrical intuition about structure of the text. We have a conjecture that tiling periodicity is not expressible by word equations. Yet another motivation for studying tiling periodicity is a hope for applications in pattern discovery in some real data.

There are natural sources of tiling periodicity when considering multidimensional $n_1 \times \cdots \times n_d$ rectangles. Let every integer point in it be colored. We sort all points in lexicographical order of their coordinates and write down all their colors in a single sequence. Assume that the initial rectangle was tiled by a smaller one $m_1 \times \cdots \times m_d$ where $m_1|n_1, \ldots, m_d|n_d$ and every copy of the smaller rectangle was colored in the same way. Then the color sequence for the bigger rectangle has a tiling period. We comment this example later after introducing some useful terminology. Automatically generated texts and XML-files might be another possible sources of tiling periodicity.

Our results. Tilling periodicity looks very simple and natural, but up to our knowledge it was never formulated before in its whole generality. We introduce a partial order on tiling periods and discover that contrary to the classical case there might be several incomparable primitive tiling periods. This helps to disprove a *common subtiler* conjecture by Tero Harju. However we prove that every primitive tiling period of a word T is also a tiling period of the primitive full period of T. This property tells us that tiling periodicity lives "inside" classical one. Finally, we present an algorithm which in $O(n \log(n) \log\log(n))$ time finds a tiling period of minimal size, the number of such periods and their compact representation.

We present a complete hierarchy of all possible tiling periods. In particular we get a recursive formula for computing function $L(n)$ that gives the maximal number of tiling periods for a unary word of length n. The value $L(n)$ might be even more than the text length. Actually, Bodini and Rivals obtain this recursive formula a few months earlier: their paper [3] was submitted in January 2006, while our results were reported only in May 2006 [10]. Here we keep our proof since (comparing to [3]) it construct explicit one-to-one correspondence between tilers and length factorizations, and introduces *levels* in the set of all tilers.

Related results. Prior to other work only tilings of a *unary* word were considered [3]. In the paper [13] authors present an algorithm for finding all tilers that have at least q (quorum parameter) matches with the text. Another related notion is *cover* [1]: a word C is a cover for a word T if any letter of T belongs to some occurrence of C in T. One of the most important results related to periodicity is a theorem by Fine and Wilf [5]. They studied the necessary and sufficient condition under which from p-periodicity and q-periodicity we can derive $\gcd(p,q)$-periodicity (recall that gcd is the greatest common divisor). We tried to prove the similar property for tiling periods. Surprisingly, it does not hold. Even two tiling periods of the same text may have no common subperiod. Our attempt to generalize the notion of periodicity is not the first one. Recently, Simpson and Tijdeman generalized Fine and Wilf theorem for multidimensional periodicity [17]. Berstel and Boasson [9] followed by Shur and Konovalova (Gamzova) [16,15] and Blanchet-Sadri [2] extensively studies partial words and their periodicity properties. However, overlapping of periods (where every letter of the text is covered exactly once) were not considered and therefore in their terms only partial words may have partial periods (i.e. periods with gaps). In the paper [8] borders of a partial word are studied. Katona and Szàsz introduced in [11] a sort of tiling periodicity in two dimensions. They consider tilers consisting only of two letters. The notion of periodicity was generalized for infinite words under the name *almost periodic sequences*, see e.g. [14].

2 Properties of Tiling Periodicity

We say that one tiler S is *smaller* than another tiler Q, and write $S \prec Q$, if S is a proper tiling period of Q. In the case of full periodicity any text has a single primitive full period. We are interested in the following question: is it true that for every ordinary word there exists a unique primitive tiling period (i.e. it is smaller than any other tiling period)? It can be reformulated in an alternative way: do any two tiling periods have a common tiling "subperiod"? Surprisingly, the answer is negative. Figure 1 presents the shortest known example T (24 letters) with two incomparable periods (the proof of their primitiveness is omitted in this version).

We can get more incomparable tiling periods. Let $T_{A_1B_1}$ and $T_{A_2B_2}$ be obtained from T by just using different letters. We construct the text T_2 by replacing every A in T by $T_{A_1B_1}$ and every B in T by $T_{A_2B_2}$. Now we can describe four incomparable tiling periods for T_2. The text T_2 has length 24^2. Group all letters to 24 blocks of 24 letters. Choose 12 of blocks using one of two partial words above. Then inside each block also keep only 12 letters using either first or second tiling period (the same in every block). We can repeat the construction several times. The text T_k has size 24^k and has 2^k incomparable periods. Asymptotically it has $n^{\frac{\log 2}{\log 24}} > \sqrt[5]{n}$ incomparable primitive tiling periods.

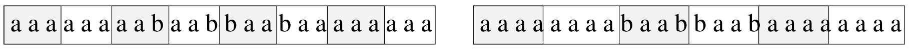

a a a a a a a a b a a b b a a b a a a a a a a a
a a a a a a a a b a a b b a a b a a a a a a a a

Fig. 1. Two primitive tiling periods of an example word, both of size 12

In 2003 in his lecture course [6] Tero Harju asked the following question. Assume that some colored cellular figure has tiling by one pattern and by another one. Is it always true that there exists the third one such that both first two can be tiled by that third one? This question is motivated by studying defect effect in combinatorics of words [7]. Our example shows that the answer is negative even for one-dimensional (but disconnected!) figures.

We consider now the situation when the number of tilers in a word is maximal, this happens for a unary word of length n (i.e only one character is used). Then we reformulate tiling periodicity in the algebraic terms (like $s_i = s_{i+p}$ for full periodicity). This reformulation helps us to prove that any primitive tiling period is smaller than the primitive full period and to compute all tiling periods of minimal size.

Lemma 1. *Take any tiling period P. Then all continuous blocks in P are of the same size s and the length of any gap in P is a multiple of s.*

Proof. Indeed, in the text tiling the second copy of P is shifted by the size of the first block s_1. Hence, for avoiding overlapping all other blocks are smaller or equal than s_1. Assume now, that the block b is the first one which is *strictly* smaller than s_1. Then the gap between b in the first copy of P and b in the second copy of P is smaller than s_1. Hence, that gap cannot be filled by anything. Every gap in P is filled by several blocks of some copies of P. Hence it must be a multiple of s.

Theorem 1. *There is a one-to-one correspondence between tiling periods of a unary word and decompositions $n = n_1 \cdot \ldots \cdot n_k$, where $n_2, \ldots, n_k \geq 2$.*

Proof. At first, we describe a set of tiling periods (hierarchial construction) for the word of length n over the unary alphabet. Then we prove that this set is complete, i.e. contains all possible tiling periods.

We divide the set of all periods in levels. Every period in our system is associated with a special *code*. The whole text itself is the only period from 0-level and has the code n. For every decomposition $n = n_1 \cdot n_2$ with $n_2 \geq 2$ the block of size n_1 is a period from 1-level with a code $n_1 \cdot n_2$. The 0-level and 1-level actually represent all classical (i.e. connected) full periods.

We now explain how to construct a period Q of $k+1$-level from a period P of $k-1$-level with the code $n_1 \cdot \ldots \cdot n_k$. Take any decomposition $n_1 = m_1 \cdot m_2 \cdot m_3$ with $m_2, m_3 \geq 2$. Through all our construction the first number of a code is equal to block size of a tiling period (all continuous blocks have equal sizes in our constructions). To construct the new tiler Q we take every block of P, divide it into m_3 groups of size $m_1 \cdot m_2$, and in every such group keep only first m_1 letters. We can notice that P can be tiled by m_2 copies of Q (with shifts $0, m_1, \ldots, m_1 \cdot (m_2 - 1)$). Hence, Q is also a tiling period for the text with the code $m_1 \cdot m_2 \cdot m_3 \cdot n_2 \cdot \ldots \cdot n_k$. Note, that our construction maintains the inequality $n_2, \ldots, n_k \geq 2$ for all codes.

We now prove that any tiling period is included in our construction. Consider a tiler P. Let s be the block size and $g \cdot s$ be the length of the first gap. The *plain power* of P is defined as the union of P-copies shifted by $0, s, 2s, \ldots, g \cdot s$.

Claim: The plain power of any tiling period P is also a tiling period.

Proof: take a text splitting in P-copies. Let s be the block size and $g \cdot s$ be the length of the first gap. We divide all copies of P into groups of $g+1$ copies in the following way. Consider all P-copies from the left to the right. The gap between first two blocks of the first copy can be filled *only by first* blocks of other P-copies. These copies together with the first one form the first group. Now assume that we already formed several such groups. Consider the first copy of P unused so far. Look at its first gap. All P-copies from the previous groups has long (at least $s \cdot (g+1)$) continuous blocks. Hence, this gap is also filled by new, still unused P-copies. Since we process all copies from the left one to the right one, all of them contribute to this gap filling by their first block. Therefore these copies form the next group we need. Every group itself is exactly a plain power of P. Hence, the initial text has also splitting in the plain power copies. The claim is proved.

Assume now that there exists some P outside of our construction. Then there also exists some tiling period P' such that (1) it is outside of our hierarchy and (2) its plain power Q is included in our hierarchy. Let us derive a contradiction from that. Indeed, let $n_1 \cdot \ldots \cdot n_k$ be the code of Q, let s be the block size of P' and $s \cdot g$ be the length of its first gap. Then n_1 is the block size of Q and it is a multiple of $s \cdot (g+1)$. Let $m_1 = s, m_2 = g+1, m_3 = n_1/(s \cdot (g+1))$. Now we see that P' is in fact included in our hierarchy as a tiling period with the code $m_1 \cdot m_2 \cdot m_3 \cdot n_2 \cdot \ldots \cdot n_k$. Therefore, our hierarchy is complete.

Corollary 1. Let $L(n)$ be the number of tiling periods for the word of length n over a unary alphabet. The theorem above states that $L(n)$ is equal to the number of factorizations $n = n_1 \cdot \ldots n_k$, where $n_2, \ldots, n_k \geq 2$. By grouping all decompositions by the rightmost factor we obtain the recurrence: $L(1) = 1$; $L(n) = 1 + \sum_{d|n, d \neq n} L(d)$.

Remark. Two related sequences are included in the On-line Encyclopedia of Integer Sequences [18] maintained by N.J.A. Sloane. The sequence A067824 is defined by the formula in Corollary 1. The sequence A107736 is the number of polynomials p with coefficients in $\{0, 1\}$ that divide $x^n - 1$ and such that $(x^n - 1)/((x - 1)p(x))$ has all coefficients in $\{0, 1\}$. But this is exactly the number of tiling periods for the unary text of the length n. Indeed, multiplying p by some polynomial with coefficients in $\{0, 1\}$ we are trying to split n "ones" in several parallel copies of p without overlapping. Till August 2006 Encyclopedia indicated that "A067824 and A107736 agree at least on first 300 terms, but no proof of equivalence is known". After recent work [3], Theorem 1 and Corollary 1 give a new independent proof of their equality. Indeed, the number of polynomials is equal to the number of tiling periods, the number of tiling periods is equal to the number of factorizations (Theorem 1), the number of factorizations satisfies the recursive formula (Corollary 1). The function $L(n)$ and the number of factorizations also appear in Knuth's book [12]. However, no closed formula for $L(n)$ is known so far.

Definition. Assume we count positions starting from 0. For a divisor p of n, by p-block we mean a subword of the form $x[i\cdot p..(i+1)\cdot p-1]$, where $0 \le i \le (n-1)/p$. We say that a word $T = T_0 \ldots T_{n-1}$ has a **multi-period** (a, b) (or a period a ranged by b) if $b|n$, and all b-blocks have a full period a. Observe that the word corresponding to this period can be different in each block. Classical full period p coincides with the multi-period (p, n).

Lemma 2. *A word T has a tiling period with the code $n_1 \cdot \ldots \cdot n_{2k}$ or $n_1 \cdot \ldots \cdot n_{2k+1}$ iff it has multi-periods respectively*

$$(n_1, n_1n_2), \ldots, (\prod_{i=1}^{2k-1} n_i, \prod_{i=1}^{2k} n_i) \quad \textit{or} \quad (n_1, n_1n_2), \ldots, (\prod_{i=1}^{2k-1} n_i, \prod_{i=1}^{2k} n_i).$$

Proof. We first describe **Step 1:** from tiling period P to multiperiodicity. We use induction over tiler's level. Consider the corresponding text tiling. Recall that all copies of P can be divided into groups of size n_2 with internal shifts $0, n_1, \ldots, n_1(n_2 - 1)$. If we divide the whole text into the blocks of size n_1n_2, every block is covered by P-copies from the same groups, and therefore it is n_1-periodic (inside the block). We proved (n_1, n_1n_2)-multiperiodicity. All others follow from the induction hypothesis for the plain power of P.

Step 2: from multi-periods to a tiling periodicity.
Consider the *top* multi-period $(\prod_{i=1}^{2k-1} n_i, \prod_{i=1}^{2k} n_i)$. Let us consider the text tiling by tiler Q with the code $(\prod_{i=1}^{2k-1} n_i) \cdot n_{2k}$ for the first statement of lemma and Q with the code $(\prod_{i=1}^{2k-1} n_i) \cdot n_{2k} \cdot n_{2k+1}$ for the second case. Directly from this top multiperiodicity every copy of Q has the same letters on the same places. Hence, Q is a tiling period for the text. Continuing the reasoning, with the help of the second multiperiodicity we find another tiling period R inside Q. Finally, the last multiperiodicity gives us the tiling period with the code we promised in the lemma's statement.

Lemma 3. *If the text T has a full period p and a multi-period (a, b), then either $b|p$ or the text has also full period $gcd(a, p)$.*

Proof. Take any letter T_i. We are going to make several moves of size $\pm p$ and $+a$ for reaching position $i + \gcd(a, p)$. We want to be on the same character every time and hence we can not make a move of size $+a$ from the last a-blocks in every b-block. As we know from extended Euclid algorithm, there exist integers k and l such that $\gcd(a, p) = ka - lp$. We will use the following *greedy* strategy. If we are able to make a move of size $+a$ we do this. Otherwise we try to make several moves of size $\pm p$. After making exactly k moves of size $+a$ we just make all the remaining moves of size $\pm p$ and we are done. We cannot follow greedy strategy only if for some position $j < p$ from all points $j, j+p, \ldots j+(\frac{n}{p}-1)p$ we cannot jump $+a$. This means that all that points belong to the last a-groups in b-blocks. Suppose now that p is not divided by b. Then $u = \gcd(p, b) \leq \frac{b}{2}$. Since $p|n$ and $b|n$, among the numbers $j \bmod b, \ldots, j + (\frac{n}{p} - 1)p \bmod b$, there exists all residues h modulo b such that $h \equiv j (\bmod\ u)$. Hence, one of these values is smaller than u which does not exceed $b/2$. But this means that the corresponding point does not belong to the last a-group in the b-block.

Theorem 2. *Any primitive tiling period Q of word T is also a tiling period for the primitive full period of T.*

Proof. We use induction over the text length. In the case of the one-letter-text theorem is true. Let now p be the length of the full period and (a, b) be the top multiperiodicity from the code of Q (here we use Lemma 1). From $p = n$ theorem follows immediately. Assume now that $p < n$. By hierarchial construction Q consists of some letters from the first a-block in every b-blocks in the text. Let us apply Lemma 2. If $b|p$, then we can produce a new tiling period Q' restricting Q to the first p symbols in the text. By p-periodicity Q can be split in several parallel copies of Q' with shifts $0, p, 2p, \ldots, (\frac{n}{p} - 1)p$. We get the contradiction against the primitivity of Q. Therefore the text has full period $\gcd(a, p)$. But p is the minimal full period. Hence, $p|a$. Since the text is p-periodic, it is also a-periodic and b-periodic. That means that we can again restrict Q only for the first a-block. Either we get a new smaller tiling period Q' (and that gives us contradiction) or $b = n$ and all letters of Q belong to the first a-block. We now see that both full period p and the tiling period Q are periods for the first a-block. Since $a \leq b/2 \leq n/2$, we can apply induction hypothesis.

Corollary 2. Take any tiling period Q and any full period p. Then they have a common "tiling subperiod".

Proof. Consider a primitive tiling period Q' that is smaller than Q. Consider the primitive full period p'. From folklore we know that $p'|p$. By Theorem 2 Q' is a tiling period for the first p'-block of the text. Hence, Q' is the required tiling subperiod for Q and p.

Remark. Using the technique from Lemma 1, Lemma 2 and Theorem 2 it is possible to prove that the primitive tiling period is unique for $n = 2^k$. We just suggest the proof scheme here. Take two primitive incomparable tiling periods. Consider their top multiperiodicities. Apply the reasoning of Lemma 2. Either one of the periods is not primitive or these multiperiodicities are the same. Going down we prove either their equivalence or their non-primitivity.

3 Algorithm *Compute-Minimal-Tilers*

We define the size of tiling period as the number of defined letters in it. In the algorithm we use the fact that for every tiling period there is a corresponding chain of *embedded* multiperiodicities $(a_1, b_1), \ldots, (a_k, b_k)$. By "embedded" we mean that $b_i | a_{i+1}$ holds for every i. Notice that the size of a tiling period is equal to $n \prod_{i=1}^{k} \frac{a_i}{b_i}$

```
**ALGORITHM** *Compute-Minimal-Tilers* $(w)$

1. Construct the acyclic tiling graph $\mathcal{G}$ of multi-periods.
   (a) $V = \{(a,b) : 1 \le a < b \le n, \ \& \ a|b \ \& \ b|n \ \& \ MultiPeriod(a,b)\ )$
   (b) $E = \{(a,b) \to (c,d) : b|c\}$.
   (c) Assign the weight $b/a$ to every node $(a,b) \in V$.
2. Find in $\mathcal{G}$ a path $\pi$ having maximal product $val(\pi)$ of its node-weights;
3. **Output** $tiler(w,\pi)$ (of size $\frac{n}{val(\pi)}$).
```

$$\begin{tikzcd}
(16,32) \arrow[r] \arrow[rr, bend left] & (4,8) \arrow[r] & (1,2)
\end{tikzcd}$$

Fig. 2. The graph $\mathcal{G}$ for the word $w = aabbaabbccddccddaabbaabbccddccdd$. The path with maximal product of weights is $\pi : (16,32) \to (4,8) \to (1,2)$. We have $val(\pi) = 8$ and $tiler(w,\pi) = a\diamond b\diamond\diamond\diamond\diamond\diamond\diamond c\diamond d$. The size of $tiler(w,\pi)$ equals $\frac{32}{8}$, since $n = |w| = 32$.

We describe how to construct the tiler $tiler(w,\pi)$ corresponding to a path π of multi-periods:

$$\pi = (p_1,q_1) \to (p_2,q_2) \to (p_3,q_3) \to \ldots (p_k,q_k).$$

For each p_i-block of w we replace all symbols which are not in the first q_i block in this block by $\diamond$. Then from the resulting word we remove all ending $\diamond$'s.

Remark. The following $O(n)$ size *max-paths graph* representation $\mathcal{G}' \subseteq \mathcal{G}$ of all tiling periods of minimal size can be constructed with additional linear work (since the graph is extremely small). For each vertex v of $\mathcal{G}$ compute all outgoing edges which are on a maximal path starting from v. The graph of all edges on maximal-product paths represents all tiling periods of minimal size. In particular we can compute easily the number of such periods.

We describe now an efficient implementation. Let w be a word of length n and let $Divisors(n)$ denotes the set of divisors of n. The subwords are given by their starting-ending positions in w. Recall that by p-block we always mean a subword of length p with a starting position being multiple of p. Denote by *period*(w) the size of the smallest full period of a word w. The basic operation in the algorithm *Compute-Minimal-Tilers* is the boolean function $MultiPeriod(q, p)$, called *main query*, which can be expressed in terminology of the blocks as follows:

MultiPeriod(q, p):

for given $p, q \in Divisors(n)$ check if each p-block has a full period q.

Lemma 4. *Assume we can preprocess the word in time $O(F(n))$ to compute each query MultiPeriod in logarithmic time. Then the algorithm works in time $O(n + F(n))$,*

Proof. By $d(n)$ we denote the number of divisors of n. It is known that $d(n) = O(n^\epsilon)$, for any constant $\epsilon > 0$. Hence the number of nodes and edges is $O(n)$. The construction of the graph can be done in linear time after preprocessing. Computation of a maximal path in time $O(n)$ is very easy, since it is an acyclic graph with linear number of edges.

Theorem 3. [Fast-Preprocessing]
We can preprocess the word in $O(n \log n \log\log n)$ time in such a way that each MultiPeriod query can be answered in constant time.

The algorithm *Compute-Minimal-Tilers* is doing a sublinear number of Multi-Period queries, hence the theorem implies immediately the following speed-up result.

Theorem 4. *The minimal-size tiler of a word can be found in $O(n \log n \log\log n)$ time.*

We now show the proof of Theorem 3. Firstly we introduce and concentrate on *small queries*. A **small query** operation is to compute for a given subword u of w (given by interval in w) the value *period*(u).

We say that a natural number is a *2-power* number if it is a power of two. We use the idea of *basic factors*: the subwords with 2-power lengths. Denote by $subword_k(i)$ the subword of size 2^k starting at position i in a given word w. We can add suitable number of endmarkers to guarantee that for each original position we have subword of the corresponding length.

Define the table $NEXT$ such that for each $0 \leq k \leq \log n,\ 0 \leq i < n$ the value of $NEXT[k, i]$ is the first position $j > i$ such that $subword_k(i) = subword_k(j)$. If there is no such j then $NEXT[k, i] = -1$.

Our basic data structure is the dictionary of basic factors, we refer to [4] for detailed definition. Denote this data structure by $DBF(w)$. For each position i in w and $0 \leq k \leq \log n$ we have a unique label $NAME[k,i] \in [1..n]$ such that $subword_k(i) = subword_k(j) \Leftrightarrow NAME[k,i] = NAME[k,j]$.

Lemma 5. *The tables NAME and NEXT can be computed in $O(n \log n)$ time.*

Proof. The table $NAME$ is the basic part of $DBF(w)$ and can be computed within required complexity using Karp-Miller-Rosenberg algorithm, see [4].

We show how to compute the table $NEXT$. Let us fix $k \leq \log n$. We sort lexicographically the pairs $(NAME[k,i], i)$. Then the block of elements in the sorted sequence with the same first component r gives the increasing sequence of positions i with the same value $NAME[k,i] = r$. Let $SORTED_1[j]$, $SORTED_2[j]$ be the first and second component of the j-th pair in the sorted sequence.

We execute:

```
for $i := 0$ to $n-1$ do NAME[k,i]:=-1;
for $j := 1$ to $n-1$ do
    if $SORTED_1[j-1] = SORTED[j]$ then
        $NEXT[k, SORTED_2[j-1] := SORTED_2[j]$;
```

The radix sorting of pairs of integers can be done in linear time for each k. We have logarithmic number of k's, hence the whole computation of $NEXT$ takes $O(n \log n)$ time. This completes the proof.

Lemma 6. [Small Queries]
Assume the tables NAME and NEXT are already computed. Then for any subword u of w (given by interval in w) we can compute period(u) in $O(\log n)$ time.

Proof. We show now how we compute $period(u)$ for $u = w[p..q]$. We can check if u has a full period of length $u/2$ or $u/3$ in constant time, since we can check equality of constant number of subwords in constant time. The DBF data structure allows to check in constant time equality of subwords which lengths are not necessarily 2-powers (decomposing them into ones which are, possibly overlapping each other).

Now let us go to smaller candidate periods, assume $period(u) \leq |u|/4$. Let us take the prefix v of u which size is a largest power 2^k such that $|u|/4 \leq 2^k \leq |u|/2$.

Claim. Let $u = w[p,q]$, if $period(w[p..q]) \leq |v|/4$ then $period(v)$ is equal to $NEXT[k,p] - p$.

The claim follows from the fact that period(u) in this case is a size of a primitive word v such that u is a full power of v. This primitive word can start an occurrence only at positions which are multiples of v, due to primitivity. Hence the first such internal position after p should be equal to $|v| = period(u)$. We can verify this “candidate period” in $O(\log n)$ time using NAME table. This completes the proof of the lemma.

We define now the following data structure. For each $p \in Divisors(n)$ define $LCM[p]$ as the least common multiple of the smallest full periods of all p-blocks of w.

Lemma 7. *We can precompute the table* LCM *in* $O(n \log n \log \log n)$ *time.*

Proof. Assume we constructed tables $NEXT$ and $NAME$. We can do it within required complexity due to Lemma 5. Then for each $p \in Divisors(n)$ we can compute the set of periods of p-blocks in time $O(p \log n)$ due to Lemma 6. Then we compute the lowest common multiple of all these periods in time $O(p \log n)$ for each p. Now the thesis follows from the well known number-theory fact that

$$\sum_{p \in Divisors(n)} p \;=\; O(n \log \log n)$$

This completes the proof of the lemma.

We can finish now the proof of Theorem 3. We know that $MultiPeriod(p, q) = true$ iff $LCM[p]$ is a divisor of q. This property can be checked in constant time using the precomputed values of $LCM[p]$. Consequently, this completes the proof of Theorem 3.

4 Directions for Further Work

There are a lot of natural, important and perhaps not so difficult questions we can suggest for further work on tiling periodicity. They are summarized in the list below.

1. Introduce and study *not full* tiling periodicity, approximate tiling periodicity, tilings by two (or more) partial words. This kind of tilings might be even more useful for compression purposes.
2. Calculate how often do words have proper tiling periods for various models of random words. Compare the answer with the classical case. Characterize the equivalent of primitive words.
3. Is it true that all primitive tiling periods are minimal-size tiling periods?

References

1. Apostolico, A., Farach, M., Iliopoulos, C.S.: Optimal superprimitivity testing for strings. Inf. Process. Lett. 39(1), 17–20 (1991)
2. Blanchet-Sadri, F.: Periodicity on partial words. Computers and Mathematics with Applications 47(1), 71–82 (2004)
3. Bodini, O., Rivals, E.: Tiling an interval of the discrete line. In: Lewenstein, M., Valiente, G. (eds.) CPM 2006. LNCS, vol. 4009, pp. 117–128. Springer, Heidelberg (2006)
4. Crochemore, M., Rytter, W.: Jewels of stringology: text algorithms. World. Sc. (2003)

5. Fine, N., Wilf, H.: Uniqueness theorems for periodic functions. Proc. Amer. Math. Soc. 16, 109–114 (1965)
6. Harju, T.: Defect theorem, lecture notes of Combinatorics of words Tarragona course (2002/2003)
7. Harju, T., Karhumäki, J.: Many aspects of defect theorems. Theor. Comput. Sci. 324(1), 35–54 (2004)
8. Iliopoulos, C.S., Mohamed, M., Mouchard, L., Perdikuri, K., Smyth, W.F., Tsakalidis, A.K.: String regularities with don't cares. Nord. J. Comput. 10(1), 40–51 (2003)
9. Boasson, L., Berstel, J.: Partial words and a theorem of Fine and Wilf. Theor. Comput. Sci. 218(1), 135–141 (1999)
10. Karhumäki, J., Lifshits, Y.: Tiling periodicity, May 2006, Dagstuhl seminar Combinatorial and Algorithmic Foundations of Pattern and Association Discovery (2006) http://kathrin.dagstuhl.de/files/Materials/06/06201/06201.LifshitsYury1.Slides.pdf
11. Katona, G.O.H., Szász, D.O.H.: Matching problems. J. of Combinatorial Theory Ser B 10(1), 60–92 (1971)
12. Knuth, D.: The Art of Computer Programming, Fascicle 3: Generating All Combinations and Partitions, vol. 4. Addison-Wesley, Reading (2005)
13. Pisanti, N., Crochemore, M., Grossi, R., Sagot, M.-F.: Bases of motifs for generating repeated patterns with wild cards. IEEE/ACM Trans. Comput. Biology Bioinform. 2(1), 40–50 (2005)
14. Pritykin, Y., Raskin, M.: Almost periodicity and finite automata. Technical Report (2007) available at http://lpcs.math.msu.su/~pritykin/files/apfinaut.zip
15. Shur, A.M., Gamzova, Y.V.: Partial words and the periods interaction property. Izvestiya RAN 68(2), 199–222 (2004)
16. Shur, A.M., Konovalova, Y.V.: On the periods of partial words. In: Sgall, J., Pultr, A., Kolman, P. (eds.) MFCS 2001. LNCS, vol. 2136, pp. 657–665. Springer, Heidelberg (2001)
17. Simpson, R.J., Tijdeman, R.: Multi-dimensional versions of a theorem of Fine and Wilf and a formula of Sylvester. Proc. Amer. Math. Soc. 131, 1661–1667 (2003)
18. Sloane, N.J.A.: The on-line encyclopedia of integer sequences. http://www.research.att.com/~njas/sequences

Fast and Practical Algorithms for Computing All the Runs in a String*

Gang Chen[1], Simon J. Puglisi[2], and W.F. Smyth[1,2]

[1] Algorithms Research Group, Department of Computing & Software
McMaster University, Hamilton, Ontario, Canada L8S 4K1
smyth@mcmaster.ca
www.cas.mcmaster.ca/cas/research/algorithms.htm

[2] Department of Computing, Curtin University, GPO Box U1987
Perth WA 6845, Australia
{puglissj,smyth}@computing.edu.au

Abstract. A ***repetition*** in a string $\boldsymbol{x}$ is a substring $\boldsymbol{w} = \boldsymbol{u}^e$ of $\boldsymbol{x}$, maximum $e \geq 2$, where $\boldsymbol{u}$ is not itself a repetition in $\boldsymbol{w}$. A ***run*** in $\boldsymbol{x}$ is a substring $\boldsymbol{w} = \boldsymbol{u}^e\boldsymbol{u}^*$ of "maximal periodicity", where $\boldsymbol{u}^e$ is a repetition and $\boldsymbol{u}^*$ a maximum-length possibly empty proper prefix of $\boldsymbol{u}$. A run may encode as many as $|\boldsymbol{u}|$ repetitions. The maximum number of repetitions in any string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ is well known to be $\Theta(n \log n)$. In 2000 Kolpakov & Kucherov showed that the maximum number of runs in $\boldsymbol{x}$ is $O(n)$; they also described a $\Theta(n)$-time algorithm, based on Farach's $\Theta(n)$-time suffix tree construction algorithm (STCA), $\Theta(n)$-time Lempel-Ziv factorization, and Main's $\Theta(n)$-time leftmost runs algorithm, to compute all the runs in $\boldsymbol{x}$. Recently Abouelhoda *et al.* proposed a $\Theta(n)$-time Lempel-Ziv factorization algorithm based on an "enhanced" suffix array — a suffix array together with other supporting data structures. In this paper we introduce a collection of fast space-efficient algorithms for computing all the runs in a string that appear in many circumstances to be superior to those previously proposed.

1 Introduction

Periodicity (repetition) in infinite strings was the first topic of stringology [30]; counting and computing the maximum-length adjacent repeating substrings (repetitions) in a finite string was, along with pattern-matching, one of the earliest computational problems on strings to be studied [17,19]. Given a nonempty string $\boldsymbol{u}$ and an integer $e \geq 2$, we call $\boldsymbol{u}^e$ a ***repetition***; if $\boldsymbol{u}$ itself is not a repetition, then $\boldsymbol{u}^e$ is a ***proper repetition***. Given a string $\boldsymbol{x}$, a ***repetition in x*** is a substring

$$\boldsymbol{x}[i..i+e|\boldsymbol{u}|-1] = \boldsymbol{u}^e,$$

where $\boldsymbol{u}^e$ is a proper repetition and neither $\boldsymbol{x}\big[i+e|\boldsymbol{u}|..i+(e+1)|\boldsymbol{u}|-1)\big]$ nor $\boldsymbol{x}[i-|\boldsymbol{u}|..i-1]$ equals $\boldsymbol{u}$. Following [29], we say the repetition has ***generator u***,

* The work of the first and third authors was supported in part by grants from the Natural Sciences & Engineering Research Council of Canada.

period $|\boldsymbol{u}|$, and ***exponent*** e; it can be specified by the integer triple $(i, |\boldsymbol{u}|, e)$. It is well known [17,3] that the maximum number of repetitions in a string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ is $\Theta(n \log n)$, and that the number of repetitions in $\boldsymbol{x}$ can be computed in $\Theta(n \log n)$ time [3,2,20].

A string $\boldsymbol{u}$ is a ***run*** iff it is periodic of (minimum) period $p \leq |\boldsymbol{u}|/2$. Thus $\boldsymbol{x} = abaabaabaabaab = (aba)^4ab$ is a run of period $|aba| = 3$. A substring $\boldsymbol{u} = \boldsymbol{x}[i..j]$ of $\boldsymbol{x}$ is a ***run in x*** iff it is a run of period p and neither $\boldsymbol{x}[i-1..j]$ nor $\boldsymbol{x}[i..j+1]$ is a run of period p (***nonextendible***). The run $\boldsymbol{u}$ has ***exponent*** $e = \lfloor |\boldsymbol{u}|/p \rfloor$ and possibly empty ***tail*** $\boldsymbol{t} = \boldsymbol{x}[i+ep..j]$ (proper prefix of $\boldsymbol{x}[i..i+p-1]$). Thus

$$\begin{array}{cccccccccccccccc} & & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \\ \boldsymbol{x} & = & b & a & a & a & b & a & a & b & a & a & b & a & b & a \end{array}$$

has a run $\boldsymbol{x}[3..12]$ of period $p = 3$ and exponent $e = 3$ with tail $\boldsymbol{t} = a$ of length $t = |\boldsymbol{t}| = 1$. It can also be specified by a triple $(i, j, p) = (3, 12, 3)$, and it includes the repetitions $(aab)^3$, $(aba)^3$ and $(baa)^2$ of period $p = 3$. In general, for $e = 2$ a run ***encodes*** $t+1$ repetitions; for $e > 2$, p repetitions. Clearly, computing all the runs in $\boldsymbol{x}$ specifies all the repetitions in $\boldsymbol{x}$.

Runs were introduced by Main [18], who showed how to compute the leftmost occurrence of every run in $\boldsymbol{x} = \boldsymbol{x}[1..n]$ by

(1) computing $\text{ST}_{\boldsymbol{x}}$, the suffix tree of $\boldsymbol{x}$ [32];
(2) using $\text{ST}_{\boldsymbol{x}}$ to compute $\text{LZ}_{\boldsymbol{x}}$, the Lempel-Ziv factorization of $\boldsymbol{x}$ [16];
(3) using $\text{LZ}_{\boldsymbol{x}}$ to compute leftmost runs.

Since steps (2) and (3) require only $\Theta(n)$ (linear) time, the use of Farach's linear-time STCA [5] enables the leftmost runs to be computed in linear time. In [14] Kolpakov & Kucherov proved that the maximum number of runs in any string of length n is $\Theta(n)$, and then showed how to compute all the runs in $\boldsymbol{x}$ from the leftmost ones in linear time. Thus in theory all runs, hence all repetitions, could be computed in linear time, though Farach's algorithm is not practical for large n.

In [1] Abouelhoda, Kurtz & Ohlebusch show how to compute $\text{LZ}_{\boldsymbol{x}}$ from a suffix array $\text{SA}_{\boldsymbol{x}}$, together with other linear structures, rather than from $\text{ST}_{\boldsymbol{x}}$. Since there now exist practical linear-time suffix array construction algorithms (SACAs) [9,12], it thus becomes feasible to compute all the runs in $\boldsymbol{x}$ in $\Theta(n)$ time for large values of n.

In this paper we describe variants of a worst-case linear-time algorithm (CPS) that, given $\text{SA}_{\boldsymbol{x}}$ and the corresponding longest common prefix array $\text{LCP}_{\boldsymbol{x}}$, computes $\text{LZ}_{\boldsymbol{x}}$ in guaranteed $\Theta(n)$ time and, according to our experiments, does so generally faster and generally with lower space requirements than either of the algorithms AKO [1] or KK-LZ (a suffix tree-based implementation of Ukkonen's algorithm [31] by Kolpakov & Kucherov specifically designed for alphabet size $\alpha \leq 4$ [13]). Ukkonen's algorithm constructs ST on-line and so permits LZ to be built from subtrees of ST; this gives it an advantage, at least in terms of space, over the fast and compact version of McCreight's STCA [25] due to Kurtz [15]. Note also [26] that the linear-time algorithms [9,12] for computing $\text{SA}_{\boldsymbol{x}}$ are not, in practice, as fast as other algorithms [24,22] that have only supralinear worst-case time bounds. Thus in testing AKO and CPS we make use of the supralinear SACA [22] that is probably at present the fastest in practice.

In Section 2 we describe our new algorithms. Section 3 summarizes the results of experiments that compare the algorithms with each other and with existing algorithms. Section 4 outlines future work.

2 Description of the Algorithms

Given a string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ on an alphabet A of size α, we refer to the suffix $\boldsymbol{x}[i..n]$, $i \in 1..n$, simply as ***suffix*** i. Then $\text{SA}_{\boldsymbol{x}}$ is an array $1..n$ in which $\text{SA}_{\boldsymbol{x}}[j] = i$ iff suffix i is the j^{th} in lexicographical order among all the suffixes of $\boldsymbol{x}$. Let $\text{lcp}_{\boldsymbol{x}}(i_1, i_2)$ denote the ***longest common prefix*** of suffixes i_1 and i_2 of $\boldsymbol{x}$. Then $\text{LCP}_{\boldsymbol{x}}$ is an array $1..n+1$ in which $\text{LCP}_{\boldsymbol{x}}[1] = \text{LCP}_{\boldsymbol{x}}[n+1] = -1$, while for $j \in 2..n$,

$$\text{LCP}_{\boldsymbol{x}}[j] = \Big|\text{lcp}_{\boldsymbol{x}}\big(\text{SA}_{\boldsymbol{x}}[j-1], \text{SA}_{\boldsymbol{x}}[j]\big)\Big|.$$

Given $\boldsymbol{x}$ and $\text{SA}_{\boldsymbol{x}}$, $\text{LCP}_{\boldsymbol{x}}$ can be quickly computed in $\Theta(n)$ time [11,23]. When the context is clear, we write SA for $\text{SA}_{\boldsymbol{x}}$, LCP for $\text{LCP}_{\boldsymbol{x}}$. For example:

$$\begin{array}{rccccccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\boldsymbol{x} = & a & b & a & a & b & a & b & a & \\
\text{SA}_{\boldsymbol{x}} = & 8 & 3 & 6 & 1 & 4 & 7 & 2 & 5 & \\
\text{LCP}_{\boldsymbol{x}} = & \text{-}1 & 1 & 1 & 3 & 3 & 0 & 2 & 2 & \text{-}1
\end{array}$$

The ***LZ factorization*** $\text{LZ}_{\boldsymbol{x}}$ of $\boldsymbol{x}$ is a factorization $\boldsymbol{x} = \boldsymbol{w_1 w_2} \cdots \boldsymbol{w_k}$ such that each $\boldsymbol{w_j}$, $j \in 1..k$, is

(a) a letter that does *not* occur in $\boldsymbol{w_1 w_2} \cdots \boldsymbol{w_{j-1}}$; or otherwise
(b) the longest substring that occurs at least twice in $\boldsymbol{w_1 w_2} \cdots \boldsymbol{w_j}$.

For our example string, $\boldsymbol{w_1} = a$, $\boldsymbol{w_2} = b$, $\boldsymbol{w_3} = a$, $\boldsymbol{w_4} = aba$, $\boldsymbol{w_5} = ba$. Typically, integer pairs (POS, LEN) specify the factorization, where POS gives a position in $\boldsymbol{x}$ and LEN the corresponding length at that position (by convention zero if the position contains a "new" letter). The example thus yields $(\text{POS}, \text{LEN}) = (1,0), (2,0), (3,1), (4,3), (7,2)$. Normally $\text{LZ}_{\boldsymbol{x}}$ is computed by first computing POS and LEN as arrays $\text{POS}[1..n]$ and $\text{LEN}[1..n]$, where $\text{POS}[i] = j < i$, $j > 0$, means that the longest match for a prefix of suffix i of $\boldsymbol{x}$ that occurs left of i in $\boldsymbol{x}$ is at position $j = \text{POS}[i]$ and has length $\text{LEN}[i]$; $\text{POS}[i] = 0$ means that i is the leftmost occurrence of letter $\boldsymbol{x}[i]$ in $\boldsymbol{x}$. As mentioned above, $\text{LZ}_{\boldsymbol{x}}$ can be quickly computed from $\text{ST}_{\boldsymbol{x}}$ in $\Theta(n)$ time [33], also from $\text{SA}_{\boldsymbol{x}}$ [1]. Our new algorithm is displayed in Figure 1.

The basic strategy of CPS is first to locate, in a left-to-right traversal of SA, a next position i_2 such that $\text{LCP}[i_2] > \text{LCP}[i_3]$ for some least $i_3 > i_2$; then second to backtrack (using stack S) from i_2, setting $\text{POS}[p_2] \leftarrow p1$ or $\text{POS}[p_1] \leftarrow p2$ according as $p_1 = \text{SA}[i_1] < p_2 = \text{SA}[i_2]$ or not. until the LCP

```
  — Using $\mathrm{SA}_{\boldsymbol{x}}$ and $\mathrm{LCP}_{\boldsymbol{x}}$, compute $\mathrm{POS}[1..n]$ and $\mathrm{LEN}[1..n]$.
$i_1 \leftarrow 1;\ i_2 \leftarrow 2;\ i_3 \leftarrow 3$
while $i_3 \le n{+}1$ do
  — Identify the next position $i_2 < i_3$ with $\mathrm{LCP}[i_2] > \mathrm{LCP}[i_3]$.
    while $\mathrm{LCP}[i_2] \le \mathrm{LCP}[i_3]$ do
        $\mathtt{push}(S, i_1);\ i_1 \leftarrow i_2;\ i_2 \leftarrow i_3;\ i_3 \leftarrow i_3{+}1$
  — Backtrack using the stack $S$ to locate the first $i_1 < i_2$ such that
  — $\mathrm{LCP}[i_1] < \mathrm{LCP}[i_2]$, at each step setting the larger position in POS
  — corresponding to equal LCP to point leftwards to the smaller one,
  — if it exists; if not, then $\mathrm{POS}[i] \leftarrow i$.
    $p_2 \leftarrow \mathrm{SA}[i_2];\ \ell_2 \leftarrow \mathrm{LCP}[i_2]$
    $\mathrm{assign}(\mathrm{POS}, \mathrm{LEN}, p_2)$
    while $\mathrm{LCP}[i_1] = \ell_2$ do
        $i_1 \leftarrow \mathtt{pop}(S)$
        $\mathrm{assign}(\mathrm{POS}, \mathrm{LEN}, p_2)$
    $\mathrm{SA}[i_1] \leftarrow p_2$
  — Reset pointers for the next stage.
    if $i_1 > 1$ then
        $i_2 \leftarrow i_1;\ i_1 \leftarrow \mathtt{pop}(S)$
    else
        $i_2 \leftarrow i_3;\ i_3 \leftarrow i_3{+}1$

procedure $\mathrm{assign}(\mathrm{POS}, \mathrm{LEN}, p_2)$
$p_1 \leftarrow \mathrm{SA}[i_1]$
if $p_1 < p_2$ then
    $\mathrm{POS}[p_2] \leftarrow p_1;\ \mathrm{LEN}[p_2] \leftarrow \ell_2;\ p_2 \leftarrow p_1$
else
    $\mathrm{POS}[p_1] \leftarrow p_2;\ \mathrm{LEN}[p_1] \leftarrow \ell_2$
```

Fig. 1. Algorithm CPS: computing $\mathrm{LZ}_{\boldsymbol{x}}$

value for the position i_1 popped from S falls below $\mathrm{LCP}[i_2]$. This processing does not guarantee that, for equal LCP (LEN), each corresponding position in POS necessarily points to the *leftmost* occurrence in $\boldsymbol{x}$, the norm for LZ factorization; however, the Main and KK runs algorithms do not require this property for their correct functioning, they require only that each position in POS should point left. In other terminology, what is in fact computed by CPS is a ***quasi suffix array*** (QSA) [6]. We call the algorithm of Figure 1 CPSa.

CPSa maintains the invariant that $i_1 < i_2 < i_3$, terminating when i_3 is incremented beyond $n{+}1$. There are two main stages corresponding to two simple inner **while** loops. The first of these pushes all entries i_1 (actually, the previous value of i_2) onto S until $\mathrm{LCP}[i_2] > \mathrm{LCP}[i_3]$. The second **while** loop assigns

$$\mathrm{POS}\big[\max\{p_1, p_2\}\big] \leftarrow \min\{p_1, p_2\}$$

(thus ensuring that POS always points left) corresponding to the current LCP value, until that value changes.

Now observe that none of the position pointers i_1, i_2, i_3 will ever point to any position i in SA such that $\mathrm{POS}\big[\mathrm{SA}[i]\big]$ has been previously set. It follows that the storage for SA and LCP can be dynamically reused to specify the location and contents of the array POS, thus saving $4n$ bytes of storage — neither the Main nor the KK algorithm requires SA/LCP. In Figure 1 this is easily accomplished by inserting $i_2 \leftarrow i_1$ at the beginning of the second inner **while** loop, then replacing

$$\text{POS}[p_2] \leftarrow p_1 \text{ by } \text{SA}[i_2] \leftarrow p_2;\ \text{LCP}[i_2] \leftarrow p_1$$
$$\text{POS}[p_1] \leftarrow p_2 \text{ by } \text{SA}[i_2] \leftarrow p_1;\ \text{LCP}[i_2] \leftarrow p_2$$

POS can then be computed by a straightforward in-place compactification of SA and LCP into SA (now redefined as POS). We call this second algorithm CPSb.

But more storage can be saved. Remove all reference to LEN from CPSb, so that it computes only POS and in particular allocates no storage for LEN. Then, after POS is computed, the space previously required for LCP becomes free and can be reallocated to LEN. Observe that only those positions in LEN that are required for the LZ-factorization need to be computed, so that the total computation time for LEN is $\Theta(n)$. In fact, without loss of efficiency, we can avoid computing LEN as an array and compute it only when required; given a sentinel value $\text{POS}[n+1] = \$$, the simple function of Figure 2 computes LEN corresponding to $\text{POS}[i]$. We call the third version CPSc.

```
function LEN($\boldsymbol{x}$, POS, $i$)
$j \leftarrow \text{POS}[i]$
if $j = i$ then
    LEN $\leftarrow 0$
else
    $\ell \leftarrow 1$
    while $\boldsymbol{x}[i+\ell] = \boldsymbol{x}[j+\ell]$ do
        $\ell \leftarrow \ell+1$
    LEN $\leftarrow \ell$
```

Fig. 2. Computing LEN corresponding to POS[i]

Since at least one position in POS is set at each stage of the main **while** loop, it follows that the execution time of CPS is linear in n. For CPSa space requirements total $17n$ bytes (for $\boldsymbol{x}$, SA, LCP, POS & LEN) plus $4s$ bytes for a stack of maximum size s. For $\boldsymbol{x} = a^n$, $s = n$, but in practical cases s will be close to the maximum height of $\text{SA}_{\boldsymbol{x}}$ and so s is bounded by $O(\log_\alpha n)$ [10].

For CPSb and CPSc, the minimum space required is $13n$ and $9n$ bytes, respectively, plus stack. Observe that for CPSa and CPSb the original (and somewhat faster) method [11] for computing LCP can be used, since it requires $13n$ bytes of storage, not greater than the total space requirements of these two variants. For CPSc, however, to achieve $9n$ bytes of storage, the Manzini variant [23] for computing LCP must be used. In fact, as described below, we test two versions of CPSc, one that uses the original LCP calculation (and therefore requires no additional space for the stack), the other using the Manzini variant (CPSd).

We remark that all versions of Algorithm CPS can easily be modified (with the introduction of another stack) to compute the LZ factorization in its usual form.

3 Experimental Results

We implemented the three versions of CPS described above, with two variants of CPSc; we call them `cpsa`, `cpsb`, `cpsc` ($13n$-byte LCP calculation), and `cpsd` ($9n$-byte LCP calculation). We also implemented the other SA-based LZ-factorization algorithm, `ako` of [1]. The implementation `kk-lz` of Kolpakov and Kucherov's algorithm was obtained from [13]. All programs were written in C or C++. We are confident that all implementations tested are of high quality.

All experiments were conducted on a 2.8 GHz Intel Pentium 4 processor with 2Gb main memory. The operating system was RedHat Linux Fedora Core 1 (Yarrow) running kernel 2.4.23. The compiler was g++ (gcc version 3.3.2) executed with the -O3 option. All running times given are the average of four runs and do not include time spent reading input files. Times were recorded with the standard C `getrusage` function. Memory usage was recorded with the `memusage` command available with most Linux distributions.

Times for the `cps` implementations and `ako` include time required for SA and LCP array construction. The implementation of `kk-lz` is only suitable for strings on small alphabets ($|\Sigma| \leq 4$) so times are only given for some files. File `chr22` was originally on an alphabet of five symbols A,C,G,T,N but was reduced by one of replacing occurrences of N randomly by the other four symbols. The N's represent ambiguities in the sequencing process. Results are not given for `ako` and `kk-lz` on some files because the memory required exceeded the capacity of the test machine.

We conclude:

(1) If speed is the main criterion, KK-LZ remains the algorithm of choice for DNA strings of moderate size.
(2) For other strings encountered in practice, CPSb is consistently faster than AKO except for some strings on very large alphabets; it also uses substantially less space, especially on run-rich strings.
(3) Overall, and especially for strings on alphabets of size greater than 4, CPSd is probably preferable since it will be more robust for main-memory use on very large strings: its storage requirement is consistently low (about half that of AKO, including on DNA strings) and it is only 25–30% slower than CPSb (and generally faster than AKO).

4 Discussion

The algorithms presented here make use of full-size suffix arrays, but there have been many "succinct" or "compressed" suffix structures proposed [21,8,28] that make use of as little as n bytes. We wish to explore the use of such structures in

Table 1. Description of the data set used in experiments

String	Size (bytes)	Σ	# runs	Description
fib35	9227465	2	7049153	The 35th Fibonacci string (see [29])
fib36	14930352	2	11405771	The 36th Fibonacci string
fss9	2851443	2	2643406	The 9th run rich string of [7]
fss10	12078908	2	11197734	The 10th run rich string of [7]
rnd2	8388608	2	3451369	Random string, small alphabet
rnd21	8388608	21	717806	Random string, larger alphabet
ecoli	4638690	4	1135423	E.Coli Genome
chr22	34553758	4	8715331	Human Chromosome 22
bible	4047392	62	177284	King James Bible
howto	39422105	197	3148326	Linux Howto files
chr19	63811651	4	15949496	Human Chromosome 19

Table 2. Runtime in milliseconds for suffix array construction and LCP computation

String	`saca`	`lcp13n`	`lcp9n`
fib35	5530	2130	3090
fib36	10440	3510	5000
fss9	1490	660	960
fss10	8180	2810	4070
rnd2	2960	2360	3030
rnd21	2840	2620	3250
ecoli	1570	1340	1700
chr22	14330	12450	16190
bible	1140	1020	1270
howto	12080	11750	14490
chr19	28400	25730	31840

Table 3. Runtime in milliseconds (in parentheses peak memory usage in bytes per input symbol) for the LZ-factorization algorithms. Underlining indicates least time/space.

String	`cpsa`	`cpsb`	`cpsc`	`cpsd`	`ako`	`kk-lz`
fib35	9360 (19.5)	$\underline{8560}$ (15.5)	9240 (13.0)	10200 ($\underline{11.5}$)	12870 (26.9)	10060 (19.9)
fib36	16730 (19.5)	$\underline{15420}$ (15.5)	16240 (13.0)	17730 ($\underline{11.5}$)	23160 (26.9)	18680 (20.8)
fss9	2680 (19.1)	2430 (15.1)	2690 (13.0)	2990 ($\underline{11.1}$)	3740 (25.4)	$\underline{1270}$ (21.3)
fss10	13240 (19.1)	12170 (15.1)	13390 (13.0)	14650 ($\underline{11.1}$)	17890 (25.4)	$\underline{7850}$ (22.5)
rnd2	6950 (17.0)	$\underline{6130}$ (13.0)	7010 (13.0)	7680 ($\underline{9.0}$)	9920 (17.0)	9820 (11.8)
rnd21	7100 (17.0)	$\underline{6270}$ (13.0)	7130 (13.0)	7760 ($\underline{9.0}$)	7810 (17.0)	– (–)
ecoli	3800 (17.0)	3350 (13.0)	3830 (13.0)	4190 ($\underline{9.0}$)	4740 (17.0)	$\underline{1610}$ (11.0)
chr22	35240 (17.0)	30320 (13.0)	36480 (13.0)	40220 ($\underline{9.0}$)	65360 (17.0)	$\underline{18240}$ (11.1)
bible	2930 (17.0)	$\underline{2540}$ (13.0)	2970 (13.0)	3220 ($\underline{9.0}$)	3670 (17.0)	– (–)
howto	32150 (17.0)	27750 (13.0)	33760 (13.0)	36500 ($\underline{9.0}$)	$\underline{23830}$ (17.0)	– (–)
chr19	70030 (17.0)	61230 (13.0)	71910 (13.0)	78020 ($\underline{9.0}$)	– (–)	$\underline{40420}$ (11.1)

this context. More generally, we note that all algorithms that compute runs or repetitions need to compute all the information required for ***repeats*** — that is, not necessarily adjacent repeating substrings. Since runs generally occur sparsely in strings [14], it seems that they should somehow be computable with less heavy machinery. Recent results [7,27,4] may suggest more economical methods. In the shorter term, we are working on methods that compute the LCP as a byproduct of SA construction, also those that bypass LCP computation.

References

1. Abouelhoda, M.I., Kurtz, S., Ohlebusch, E.: Replacing suffix trees with enhanced suffix arrays. J. Discrete Algs. 2, 53–86 (2004)
2. Apostolico, A., Preparata, F.P.: Optimal off-line detection of repetitions in a string. Theoret. comput. sci. 22, 297–315 (1983)
3. Crochemore, M.: An optimal algorithm for computing the repetitions in a word. Inform. process. lett. 12(5), 244–250 (1981)
4. Fan, K., Puglisi, S.J., Smyth, W.F., Turpin, A.: A new periodicity lemma. SIAM J. Discrete Math. 20(3), 656–668 (2006)
5. Martin Farach, Optimal suffix tree construction with large alphabets Proc. 38^{th} FOCS pp. 137–143 (1997)
6. Franek, F., Holub, J., Smyth, W.F., Xiao, X.: Computing quasi suffix arrays. J. Automata, Languages & Combinatorics 8(4), 593–606 (2003)
7. Franek, F., Simpson, R. J., Smyth, W. F.: The maximum number of runs in a string. In: Miller, M., Park, K.(eds.) Proc. 14^{th} AWOCA, pp. 26–35 (2003)
8. Grossi, R., Vitter, J.S.: Compressed suffix arrays and suffix trees with applications to text indexing & string matching. SIAM J. Computing 35(2), 378–407 (2005)
9. Kärkkäinen, J., Sanders, P.: Simple linear work suffix array construction. In: Proc. 30^{th} ICALP. pp. 943–955 (2003)
10. Karlin, S., Ghandour, G., Ost, F., Tavare, S., Korn, L.J.: New approaches for computer analysis of nucleic acid sequences. Proc. Natl. Acad. Sci. USA 80, 5660–5664 (1983)
11. Kasai, T., Lee, G., Arimura, H., Arikawa, S., Park, K.: Linear-time longest-common-prefix computation in suffix arrays and its applications. In: Amir, A., Landau, G.M. (eds.) CPM 2001. LNCS, vol. 2089, Springer, Heidelberg (2001)
12. Ko, P., Aluru, S.: Space efficient linear time construction of suffix arrays. In: Baeza-Yates, R.A., Chávez, E., Crochemore, M. (eds.) CPM 2003. LNCS, vol. 2676, Springer, Heidelberg (2003)
13. Kolpakov, R., Kucherov, G.: `http://bioinfo.lifl.fr/mreps/`
14. Kolpakov, R., Kucherov, G.: On maximal repetitions in words. J. Discrete Algs. 1, 159–186 (2000)
15. Kurtz, S.: Reducing the space requirement of suffix trees. Software Practice & Experience 29(13), 1149–1171 (1999)
16. Lempel, A., Ziv, J.: On the complexity of finite sequences. IEEE Trans. Information Theory 22, 75–81 (1976)
17. Lentin, A., Schützenberger, M.P.: A combinatorial problem in the theory of free monoids, Combinatorial Mathematics & Its Applications. In: Bose, R.C., Dowling, T.A. (eds.) University of North Carolina Press, pp. 128–144 (1969)
18. Main, M.G.: Detecting leftmost maximal periodicities. Discrete Applied Maths 25, 145–153 (1989)

19. Main, M.G., Lorentz, R.J.: An O(n log n) Algorithm for Recognizing Repetition, Tech. Rep. CS-79–056, Computer Science Department, Washington State University (1979)
20. Main, M.G., Lorentz, R.J.: An O(nlog n) algorithm for finding all repetitions in a string. J. Algs. 5, 422–432 (1984)
21. Mäkinen, V., Navarro, G.: Compressed full-text indices, ACM Computing Surveys (to appear)
22. Maniscalco, M., Puglisi, S.J.: Faster lightweight suffix array construction. In: Ryan, J., Dafik (eds.) Proc. 17^{th} AWOCA pp. 16–29 (2006)
23. Manzini, G.: Two space-saving tricks for linear time LCP computation. In: Hagerup, T., Katajainen, J. (eds.) SWAT 2004. LNCS, vol. 3111, Springer, Heidelberg (2004)
24. Manzini, G., Ferragina, P.: Engineering a lightweight suffix array construction algorithm. Algorithmica 40, 33–50 (2004)
25. McCreight, E.M.: A space-economical suffix tree construction algorithm. J. Assoc. Comput. Mach. 32(2), 262–272 (1976)
26. Puglisi, S.J., Smyth, W.F., Turpin, A.: A taxonomy of suffix array construction algorithms, ACM Computing Surveys (to appear)
27. Rytter, W.: The number of runs in a string: improved analysis of the linear upper bound. In: Durand, B., Thomas, W. (eds.) Proc. 23^{rd} STACS. LNCS, vol. 2884, pp. 184–195. Springer, Heidelberg (2006)
28. Sadakane, K.: Space-efficient data structures for flexible text retrieval systems. In: Bose, P., Morin, P. (eds.) ISAAC 2002. LNCS, vol. 2518, Springer, Heidelberg (2002)
29. Smyth, B.: Computing Patterns in Strings, Pearson Addison-Wesley, p. 423 (2003)
30. Thue, A.: Über unendliche zeichenreihen. Norske Vid. Selsk. Skr. I. Mat. Nat. Kl. Christiana 7, 1–22 (1906)
31. Ukkonen, E.: On-line construction of suffix trees. Algorithmica 14, 249–260 (1995)
32. Weiner, P.: Linear pattern matching algorithms. In: Proc. 14th Annual IEEE Symp. Switching & Automata Theory, pp. 1–11 (1973)
33. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Trans. Information Theory 23, 337–343 (1977)

Longest Common Separable Pattern Among Permutations

Mathilde Bouvel[1], Dominique Rossin[1], and Stéphane Vialette[2]

[1] CNRS, Université Paris Diderot, Laboratoire d'Informatique Algorithmique: Fondements et Applications, 2 Place Jussieu, Case 7014, F-75251 Paris Cedex 05, France
{mbouvel,rossin}@liafa.jussieu.fr

[2] Laboratoire de Recherche en Informatique (LRI), bât.490, Univ. Paris-Sud XI, F-91405 Orsay cedex, France
Stephane.Vialette@lri.fr

Abstract. In this paper, we study the problem of finding the longest common separable pattern among several permutations. We first give a polynomial-time algorithm when the number of input permutations is fixed and next show that the problem is **NP**–hard for an arbitrary number of input permutations even if these permutations are separable.

On the other hand, we show that the **NP**–hard problem of finding the longest common pattern between two permutations cannot be approximated better than within a ratio of $\sqrt{\mathsf{opt}}$ (where opt is the size of an optimal solution) when taking common patterns belonging to pattern-avoiding permutation classes.

1 Introduction and Basic Definitions

A permutation π is said to be a pattern (or to occur) within a permutation σ if σ has a subsequence that is order-isomorphic to π. For example, a permutation contains the pattern 123 (resp. 321) if it has an increasing (resp. decreasing) subsequence of length three. Here, note that members need not actually be consecutive, merely increasing (resp. descreasing). Within the last few years, the study of the *pattern containment* relation on permutations has become a very active area of research in both combinatorics and computer science.

In combinatorics, much research focused on closed classes of permutations, *i.e.*, permutations that are closed downwards under forming subpermutations. A huge literature is devoted to this subject. To cite only a few of a plethora of suitable examples, Knuth considered permutations that do not contain the pattern 312 [15], Lovàsz considered permutations that do not contain the pattern 213 [17] and Rotem those that do not contain 231 nor 312 [20].

Surprisingly enough, there is considerably less research on algorithmic aspects of pattern involvement. Actually, it appears to be a difficult problem to decide whether a permutation occurs as a pattern in another permutation. Indeed, the problem in this general version is **NP**–complete [6] and only a few special cases are known to be polynomial-time solvable. Of particular interest here, the case of *separable patterns*, *i.e.*, permutations that contain neither the subpattern 3142 nor 2413, was however proved to be solvable in $\mathcal{O}(kn^6)$ time and $\mathcal{O}(kn^4)$ space in [6], where k is the length of the pattern and n is the length of the target permutation. The design of efficient algorithms for the recognition of a fixed pattern in a permutation is considered in [3] and in particular a $\mathcal{O}(n^5 \log n)$ time algorithm is given for finding separable patterns. L. Ibarra subsequently improved the complexity for separable patterns to $\mathcal{O}(kn^4)$ time and $\mathcal{O}(kn^3)$ space in [14]. Beside separable patterns, only a few restricted cases were considered. A $\mathcal{O}(n \log \log n)$ time algorithm is presented in [9] for finding the longest increasing or decreasing subpermutation of a permutation of length n.

In the present paper we consider the problem of extending the classical longest common subsequence problem into the realm of permutations. The *longest common subsequence* problem (LCS) is to find a longest sequence which is a subsequence of all sequences in a set of input sequences. The LCS problem is known to be polynomial-time solvable for a fixed number of input sequences [10] and to be **NP**–hard for the general case of an arbitrary number (*i.e.*, not a priori fixed) of input sequences [18], even in the case of the binary alphabet. In the context of permutations, the LCS-like problem naturally asks to find a longest permutation that occurs within each input permutation. However, oppositely to the classical LCS problem, the permutation-related LCS problem is **NP**–hard even for two permutations since the problem of deciding whether a permutation occurs in another permutation is **NP**–hard [6], and hence additional restrictions on the input permutations or the common pattern are needed for algorithmic solutions. We focus in this paper on the case where the common pattern is required to be a separable permutation, a non-trivial class of permutations for which the pattern involvement problem is polynomial-time solvable. The problem we consider is thus the following: Given a set of permutations, find a longest separable permutation that occurs as a pattern in each input permutation. Of particular importance in this context, observe that we do not impose here the input permutations to be separable.

This paper is organized as follows. In the remainder of Section 1, we briefly discuss basic notations and definitions that we will use throughout. In Section 2, we give a polynomial-time algorithm for finding the largest common separable pattern that appears as a pattern in a fixed number of permutations. Section 3 is devoted to proving hardness of the problem. Finally, some inapproximation issues are presented in Section 4.

1.1 Permutations

A permutation $\sigma \in S_n$ is a bijective map from $[1..n]$ to itself. The integer n is called the *length* of σ. We denote by σ_i the image of i under σ. A permutation can be seen as a word $\sigma_1\sigma_2 \ldots \sigma_n$ containing exactly once each letter $i \in [1..n]$. For each entry σ_i of a permutation σ, we call i its *index* and σ_i its *value*.

Definition 1 (Pattern in a permutation). *A permutation $\pi \in S_k$ is a* pattern *of a permutation $\sigma \in S_n$ if there is a subsequence of σ which is order-isomorphic to π; in other words, if there is a subsequence $\sigma_{i_1}\sigma_{i_2} \ldots \sigma_{i_k}$ of σ*

(with $1 \leq i_1 < i_2 < \ldots < i_k \leq n$) such that $\sigma_{i_\ell} < \sigma_{i_m}$ whenever $\pi_\ell < \pi_m$. We also say that π is involved *in σ and call $\sigma_{i_1}\sigma_{i_2}\ldots\sigma_{i_k}$ an* occurrence *of π in σ.*

A permutation σ that does not contain π as a pattern is said to *avoid* π. Classes of permutations of interest are the *pattern-avoiding permutation classes*: the class of all permutations avoiding the patterns $\pi_1, \pi_2 \ldots \pi_k$ is denoted $S(\pi_1, \pi_2, \ldots, \pi_k)$, and $S_n(\pi_1, \pi_2, \ldots, \pi_k)$ denotes the set of permutations of length n avoiding $\pi_1, \pi_2, \ldots, \pi_k$.

Example 1. For example $\sigma = 142563$ contains the pattern 1342, and 1563, 1463, 2563 and 1453 are the occurrences of this pattern in σ. But $\sigma \in S(321)$: σ avoids the pattern 321 as no subsequence of length 3 of σ is isomorphic to 321, *i.e.*, is decreasing.

If a pattern π has an occurrence $\sigma_{i_1}\sigma_{i_2}\ldots\sigma_{i_k}$ in a permutation σ of length n, let I and V be two subintervals of $[1..n]$ such that $\{i_1, i_2, \ldots, i_k\} \subseteq I$ and $\{\sigma_{i_1}, \sigma_{i_2}, \ldots, \sigma_{i_k}\} \subseteq V$; then we say that π has an occurrence in σ in the intervals I of indices and V of values, or that π is a pattern of σ using the intervals I of indices and V of values in σ.

Among the pattern-avoiding permutation classes, we are particularly interested here in the separable permutations.

Definition 2 (Separable permutation). *The class of separable permutations, denoted $\mathcal{S}ep$, is $\mathcal{S}ep = S(2413, 3142)$.*

There are numerous characterizations of separable permutations, for example in terms of permutation graphs [6], of interval decomposition [8,5,7], or with ad-hoc structures like the separating trees [6,14]. Separable permutations have been widely studied in the last decade, both from a combinatorial [21,11] and an algorithmic [4,6,14] point of view.

We define two operations of concatenation on permutation patterns:

Definition 3 (Pattern concatenation). *Consider two patterns π and π' of respective lengths k and k'. The positive and negative concatenations of π and π' are defined respectively by:*

$$\pi \oplus \pi' = \pi_1 \cdots \pi_k(\pi'_1 + k) \cdots (\pi'_{k'} + k)$$
$$\pi \ominus \pi' = (\pi_1 + k') \cdots (\pi_k + k')\pi'_1 \cdots \pi'_{k'}$$

The following property, whose proof is straightforward with separating trees, is worth noticing for our purpose:

Property 1. If both π and π' are separable patterns, then $\pi \oplus \pi'$ and $\pi \ominus \pi'$ are also separable. Conversely, any separable pattern π of length at least 2 can be decomposed into $\pi = \pi_1 \oplus \pi_2$ or $\pi = \pi_1 \ominus \pi_2$ for some smaller but non-empty separable patterns π_1 and π_2.

1.2 Pattern Problems on Permutations

The first investigated problem on patterns in permutations is the *Pattern Involvement* Problem:

Problem 1 (Pattern Involvement Problem).
INPUT: A pattern π and a permutation σ.
OUTPUT: A boolean indicating whether π is involved in σ or not.

It was shown to be **NP**–complete in [6]. However, in [6] the authors also exhibit a particular case in which it is polynomial-time solvable: namely when the pattern π in input is a separable pattern.

Another problem of interest is the *Longest Common Pattern* Problem (LCP for short):

*Problem 2 (*LCP *Problem).*
INPUT: A set X of permutations.
OUTPUT: A pattern of maximal length occurring in each $\sigma \in X$.

This problem is clearly **NP**–hard in view of the complexity of Problem 1. We showed in [7] that it is solvable in polynomial time when $X = \{\sigma_1, \sigma_2\}$ with σ_1 a separable permutation (or more generally, when the length of the longest *simple permutation* [2] involved in σ_1 is bounded).

In this paper, we will consider a restriction of Problem 2. For any class $\mathcal{C}$ of (pattern-avoiding) permutations, we define the *Longest Common $\mathcal{C}$-Pattern* Problem (LC$\mathcal{C}$P for short):

*Problem 3 (*LC$\mathcal{C}$P *Problem).*
INPUT: A set X of permutations.
OUTPUT: A pattern of $\mathcal{C}$ of maximal length occurring in each $\sigma \in X$.

In particular, we focus in this paper on the *Longest Common Separable Pattern* Problem (LC$\mathcal{S}ep$P) which in fact is LC$\mathcal{C}$P where $\mathcal{C} = \mathcal{S}ep$.

To our knowledge, complexity issues of the LC$\mathcal{C}$P Problem are still unexplored. We will show in this paper that the LC$\mathcal{S}ep$P Problem is **NP**–hard in general, but solvable in polynomial-time when the cardinality of the set X of permutations in the input is bounded by any constant K. However the method we use in our polynomial-time algorithm for solving LC$\mathcal{S}ep$P on K permutations is specific to separable patterns and cannot be extended to any class $\mathcal{C}$ of pattern-avoiding permutations.

2 Polynomial Algorithm for the Longest Common Separable Pattern Among a Finite Number of Permutations

In [6], the authors show that the problem of deciding whether a permutation π is a pattern of a permutation σ is **NP**–complete. A consequence is that the problem of finding a longest common pattern among two or more permutations in **NP**–hard. However, they describe a polynomial-time algorithm for solving the Pattern Involvement Problem when the pattern π is *separable*. This algorithm uses dynamic programming, and processes the permutation according to one of its separating trees.

With the same ideas, we described in [7] a polynomial-time algorithm for finding a longest common pattern between two permutations, provided that one of them is separable. Notice that a longest common pattern between two permutations, one of them being separable, is always separable.

In this section, we generalize the result obtained in [7] giving a polynomial-time algorithm for finding a longest common *separable* pattern among K permutations, K being any fixed integer, $K \geq 1$. Notice that we make no hypothesis on the K input permutations.

Like in [6] and [7], our algorithm will use dynamic programming. However, since we do not have a separability hypothesis on any of the permutations, we cannot design an algorithm based on a separating tree associated to one of the permutations in the input. To compute a longest common separable pattern among the input permutations, we will only consider sub-problems corresponding to K-tuples of intervals of indices and values, one such pair of intervals for each permutation.

Namely, let us consider K permutations $\sigma^1, \ldots, \sigma^K$, of length $n_1, \ldots, n_K$ respectively, and denote by n the maximum of the n_q's, $1 \leq q \leq K$. For computing a longest common separable pattern among $\sigma^1, \ldots, \sigma^K$, we will consider a dynamic programming array M of dimension $4K$, and when our procedure for filling in M ends, we intend that $M(i_1, j_1, a_1, b_1, \ldots, i_K, j_K, a_K, b_K)$ contains a common separable pattern π among $\sigma^1, \ldots, \sigma^K$ that is of maximal length among those using, for any $q \in [1..K]$, intervals $[i_q..j_q]$ of indices and $[a_q..b_q]$ of values in σ^q. If we are able to fill in M in polynomial time, with the above property being satisfied, the entry $M(1, n_1, 1, n_1, \ldots, 1, n_K, 1, n_K)$ will contain, at the end of the procedure, a longest common separable pattern among $\sigma^1, \ldots, \sigma^K$.

Algorithm 1 shows how the array M can indeed be filled in in polynomial time. In Algorithm 1, *Longest* is the naive linear-time procedure that runs through a set S of patterns and returns a pattern in S of maximal length.

Before giving the details of the proof of our algorithm for finding a longest common separable pattern, we state two lemmas. They should help understanding how common separable patterns can be merged, or on the contrary split up, to exhibit other common separable patterns. We are also interested in the stability of the maximal length property when splitting up patterns.

Lemma 1. *Let π_1 be a common separable pattern among $\sigma^1, \ldots, \sigma^K$ that uses the intervals $[i_q..h_q - 1]$ of indices and $[a_q..c_q - 1]$ (resp. $[c_q..b_q]$) of values in σ^q, for all $q \in [1..K]$.*

Let π_2 be a common separable pattern among $\sigma^1, \ldots, \sigma^K$ that uses the intervals $[h_q..j_q]$ of indices and $[c_q..b_q]$ (resp. $[a_q..c_q - 1]$) of values in σ^q, for all $q \in [1..K]$.

Then $\pi = \pi_1 \oplus \pi_2$ (resp. $\pi = \pi_1 \ominus \pi_2$) is a common separable pattern among $\sigma^1, \ldots, \sigma^K$ that uses the intervals $[i_q..j_q]$ of indices and $[a_q..b_q]$ of values in σ^q, for all $q \in [1..K]$.

Algorithm 1. Longest common separable pattern among K permutations

```
 1: INPUT: $K$ permutations $\sigma^1, \ldots, \sigma^K$ of length $n_1, \ldots, n_K$ respectively

 2: CREATE AN ARRAY $M$:
 3: for any integers $i_q$, $j_q$, $a_q$ and $b_q \in [1..n_q]$, for all $q \in [1..K]$ do
 4:     $M(i_1, j_1, a_1, b_1, \ldots, i_K, j_K, a_K, b_K) \leftarrow \epsilon$
 5: end for

 6: FILL IN $M$:
 7: for any integers $i_q$, $j_q$, $a_q$ and $b_q \in [1..n_q]$, $i_q \leq j_q$, $a_q \leq b_q$, for all $q \in [1..K]$, by
    increasing values of $\sum_q (j_q - i_q) + (b_q - a_q)$ do
 8:     if $\exists q \in [1..K]$ such that $i_q = j_q$ or $a_q = b_q$ then
 9:         if $\forall q \in [1..K], \exists h_q \in [i_q..j_q]$ such that $\sigma^q_{h_q} \in [a_q..b_q]$ then
10:             $M(i_1, j_1, a_1, b_1, \ldots, i_K, j_K, a_K, b_K) \leftarrow 1$
11:         else
12:             $M(i_1, j_1, a_1, b_1, \ldots, i_K, j_K, a_K, b_K) \leftarrow \epsilon$
13:         end if
14:     else
15:         /*$\forall q \in [1..K], i_q < j_q$ and $a_q < b_q$ /*
            $M(i_1, j_1, a_1, b_1, \ldots, i_K, j_K, a_K, b_K) \leftarrow Longest(S_\oplus \cup S_\ominus \cup S)$ where

            $S_\oplus = \{M(i_1, h_1 - 1, a_1, c_1 - 1, \ldots, i_K, h_K - 1, a_K, c_K - 1) \oplus M(h_1, j_1, c_1, b_1,$
                $\ldots, h_K, j_K, c_K, b_K) : i_q < h_q \leq j_q, a_q < c_q \leq b_q, \forall q \in [1..K]\}$
            $S_\ominus = \{M(i_1, h_1 - 1, c_1, b_1, \ldots, i_K, h_K - 1, c_K, b_K) \ominus M(h_1, j_1, a_1, c_1 - 1,$
                $\ldots, h_K, j_K, a_K, c_K - 1) : i_q < h_q \leq j_q, a_q < c_q \leq b_q, \forall q \in [1..K]\}$
            $S = \{1\}$ if $\forall q \in [1..K], \exists h_q \in [i_q..j_q]$ such that $\sigma^q_{h_q} \in [a_q..b_q]$,
              $= \{\epsilon\}$ otherwise.

16:     end if
17: end for

18: OUTPUT: $M(1, n_1, 1, n_1, \ldots, 1, n_K, 1, n_K)$
```

Proof. The proof is technical but straighforward.

Lemma 2. *Let π be a common separable pattern of maximal length among $\sigma^1, \ldots, \sigma^K$ among those using the intervals $[i_q..j_q]$ of indices and $[a_q..b_q]$ of values in σ^q, for all $q \in [1..K]$.*

If $\pi = \pi_1 \oplus \pi_2$ (resp. $\pi = \pi_1 \ominus \pi_2$), with π_1 and π_2 non-empty *separable patterns, then there exist indices $(h_q)_{q \in [1..K]}$ and values $(c_q)_{q \in [1..K]}$, with $i_q < h_q \leq j_q, a_q < c_q \leq b_q, \forall q \in [1..K]$, such that:*

i) π_1 is a common separable pattern of maximal length among $\sigma^1, \ldots, \sigma^K$ among those using the intervals $[i_q..h_q - 1]$ of indices and $[a_q..c_q - 1]$ (resp. $[c_q..b_q]$) of values in σ^q, for all $q \in [1..K]$, and *ii)* π_2 *is a common separable pattern of maximal length among* $\sigma^1, \ldots, \sigma^K$ *among those using the intervals* $[h_q..j_q]$ *of indices and* $[c_q..b_q]$ *(resp.* $[a_q..c_q - 1]$*) of values in* σ^q*, for all* $q \in [1..K]$*.*

Proof. This is a direct consequence of the definition of a separable permutation.

Proposition 1. *Algorithm 1 is correct: it outputs a longest common separable pattern among the K permutations in the input.*

Proof. Consider the array M returned by Algorithm 1. We show by induction on $\sum_q (j_q - i_q) + (b_q - a_q)$ using the previous two Lemmas that $M(i_1, j_1, a_1, b_1, \ldots, i_K, j_K, a_K, b_K)$ contains a common separable pattern π among $\sigma^1, \ldots, \sigma^K$ that is of maximal length among those using, for any $q \in [1..K]$, intervals $[i_q..j_q]$ of indices and $[a_q..b_q]$ of values in σ^q. □

Proposition 2. *Algorithm 1 runs in time* $\mathcal{O}(n^{6K+1})$ *and space* $\mathcal{O}(n^{4K+1})$*.*

Proof. Algorithm 1 handles an array M of size $\mathcal{O}(n^{4K})$, where each cell contains a pattern of length at most n, so that the total space complexity is $\mathcal{O}(n^{4K+1})$. For filling in one entry $M(i_1, j_1, a_1, b_1, \ldots, i_K, j_K, a_K, b_K)$, if $\exists q \in [1..K]$ such that $i_q = j_q$ or $a_q = b_q$ (lines 9 to 13 of Algorithm 1), the time complexity is $\mathcal{O}(n^K)$. If no such q exists (line 15 of Algorithm 1), the time complexity needed to fill in $M(i_1, j_1, a_1, b_1, \ldots, i_K, j_K, a_K, b_K)$, using the entries of M previously computed, is $\mathcal{O}(n^{2K+1})$. Indeed, we search for an element of maximal length among $\mathcal{O}(n^{2K})$ elements, each element being computed in $\mathcal{O}(n)$-time as the concatenation of two previously computed entries of M. Consequently, the total time complexity to fill in M is $\mathcal{O}(n^{6K+1})$. □

A consequence of Propositions 1 and 2 is:

Theorem 1. *For any fixed integer K, the problem of computing a longest common separable pattern among K permutations is in P.*

We may wonder whether a longest common separable pattern between two permutations σ^1 and σ^2 (computed in polynomial time by Algorithm 1) is a good approximation of a longest common pattern between σ^1 and σ^2 (whose computation is **NP**–hard). Section 4 gives a negative answer to this question, by the more general Corollary 1.

3 Hardness Result

We proved in the preceding section that the LC$\mathcal{S}ep$P problem is polynomial-time solvable provided a constant number of input permutations. We show here that the C$\mathcal{S}ep$P problem (the general decision version of LC$\mathcal{S}ep$P), is **NP**–complete.

*Problem 4 (*C$\mathcal{S}ep$P *Problem).*
INPUT: A set X of permutations and an integer k.
OUTPUT: A boolean indicating if there a separable pattern of length k occurring in each $\sigma \in X$.

Actually, we will prove more, namely that the C$\mathcal{S}ep$P problem is **NP**–complete even if each input permutation is separable. An immediate consequence is the **NP**–hardness of LC$\mathcal{S}ep$P. For ease of exposition, our proof is given in terms of matching diagrams.

Definition 4 (Matching Diagram). *A matching diagram G of size n is a vertex-labeled graph of order i.e., number of vertices, $2n$ and size i.e., number of edges, n where each vertex is labeled by a distinct label from $\{1, 2, \ldots, 2n\}$ and each vertex $i \in \{1, 2, \ldots, n\}$ (resp. $j \in \{n+1, n+2, \ldots, 2n\}$) is connected by an edge to exactly one vertex $j \in \{n+1, n+2, \ldots, 2n\}$ (resp. $i \in \{1, 2, \ldots, n\}$). We denote the set of vertices and edges of G by $V(G)$ and $E(G)$, respectively.*

It is well-known that matching diagrams of size n are in one-to-one correspondence with permutations of length n (see Figure 1 for an illustration).

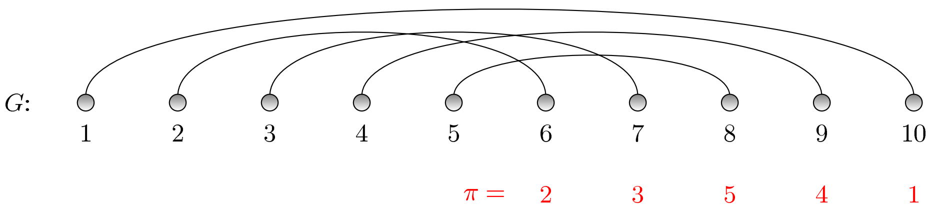
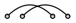
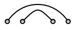
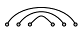
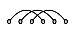
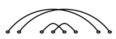


Fig. 1. Shown here is the correspondence between the permutation $\pi = 2\ 3\ 5\ 4\ 1$ and the associated matching diagram G

Let G and G' be two matching diagrams. The matching diagram G' is said to *occur* in G if one can obtain G' from G by a sequence of edge deletions. More formally, the deletion of the edge (i, j), $i < j$, consists in (1) the deletion of the edge (i, j), (2) the deletion of the two vertices i and j, and (3) the relabeling of all vertices $k \in [i+1..j-1]$ to $k-1$ and all the vertices $k > j$ to $k-2$. Therefore, the decision version of the LC$\mathcal{S}ep$P is equivalent to the following problem: Given a set of matching diagrams and a positive integer k, find a matching diagram of size k which occurs in each input diagram [16].

Clearly, two edges in a matching diagram G are either crossing  or nested . Moreover, it is easily seen that an occurrence in G of a matching diagram G' of which all edges are crossing (resp. nested) correspond to an occurrence in the permutation associated with G of an *increasing* (resp. *decreasing*) subsequence.

For the purpose of permutations, convenient matching diagrams are needed. A matching diagram is called a *tower* if it is composed of pairwise nested edges  and a *staircase* if it is composed of pairwise crossing edges . A matching diagram is called a *tower of staircases* if its edge set can be partitioned in nested staircases .

Theorem 2. *The C$\mathcal{S}ep$P problem is* **NP***–complete even if each input permutation is separable.*

Proof. C$\mathcal{S}ep$P is clearly in **NP**. For proving hardness, we reduce from the INDEPENDENT-SET problem which is known to be **NP**–complete [13] . Let G be an arbitrary graph instance of the INDEPENDENT-SET problem. Write $V(G) = \{1, 2, \ldots, n\}$. We now detail the construction of $n+1$ matching diagrams $G_0, G_1, G_2, \ldots, G_n$, each corresponding to a separable permutation. First the matching diagram G_0 is a tower of n staircases $A_{0,1}, A_{0,2}, \ldots, A_{0,n}$, each of size $n+1$ (see Figure 2, middle part; staircases are represented by shaded forms), *i.e.*,

$$\forall j,\ 1 \leq j \leq n, \qquad |A_{0,j}| = n+1.$$

Each matching diagram G_i, $1 \leq i \leq n$, is composed of two crossing towers of n staircases each referred as $A_{i,1}, A_{i,2}, \ldots, A_{i,n}$ and $B_{i,1}, B_{i,2}, \ldots, B_{i,n}$ (see Figure 2, bottom part), and defined as follows:

$$\forall i,\ 1 \leq i \leq n,\ \forall j,\ 1 \leq j \leq n, \qquad |A_{i,j}| = \begin{cases} n+1 & \text{if } i \neq j \\ n & \text{if } i = j \end{cases}$$

$$\forall i,\ 1 \leq i \leq n,\ \forall j,\ 1 \leq j \leq n, \qquad |B_{i,j}| = \begin{cases} n+1 & \text{if } (i,j) \notin E(G) \\ n & \text{if } (i,j) \in E(G). \end{cases}$$

It is simple matter to check that all matching diagrams G_i, $0 \leq i \leq n$, correspond to separable permutations and that our construction can be carried on in polynomial time. This ends our construction.

It can be shown that that there exists an independent set $V' \subseteq V(G)$ of size k in G if and only if there exists a matching diagram G_{sol} of size $n^2 + k$ that occurs in each input matching diagram G_i, $0 \leq i \leq n$ (details omitted). □

4 Approximation Ratio

In this section, we return to the LC$\mathcal{C}$P Problem for K permutations. As said before, the general LCP Problem is **NP**–hard as well as the Pattern Involvement Problem [6]. In this section we prove the following result:

Theorem 3. *For all $\epsilon > 0$ and $\mathcal{C}$, a pattern-avoiding permutation class, there exists a sequence $(\sigma_n)_{n \in \mathbb{N}}$ of permutations $\sigma_n \in S_n$ such that*

$$|\pi_n| = o\left(n^{0.5+\epsilon}\right)$$

where π_n is a longest permutation of class $\mathcal{C}$ involved as a pattern in σ_n.

Before proving this result we need the following Lemma.

Lemma 3. *Given a permutation $\pi \in S_k$, the number of permutations $\sigma \in S_n$ such that π is involved in σ is at most $(n-k)!\binom{n}{k}^2$.*

Proof. The statement follows from a simple counting argument. □

We can now prove Theorem 3.

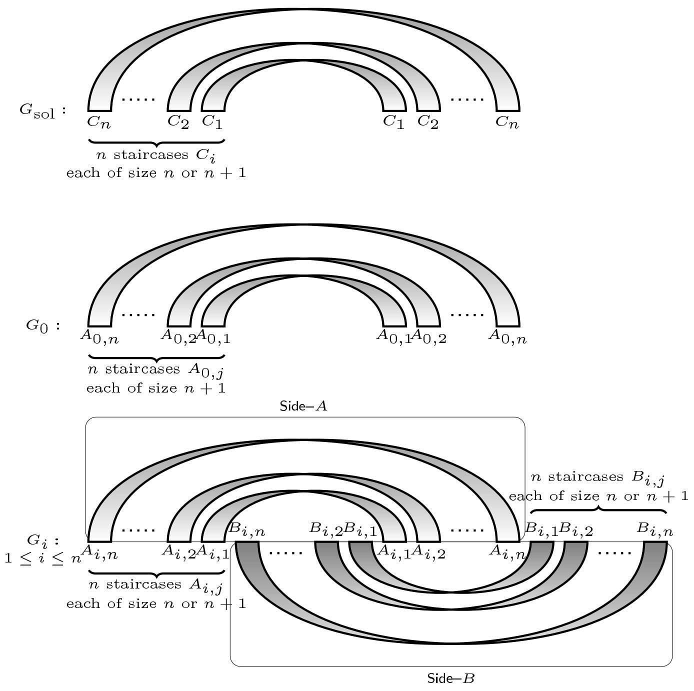

G_{sol} :
C_n ..... C_2 C_1 C_1 C_2 ..... C_n
n staircases C_i each of size n or $n+1$
G_0 :
$A_{0,n}$ ..... $A_{0,2}$ $A_{0,1}$ $A_{0,1}$ $A_{0,2}$ ..... $A_{0,n}$
n staircases $A_{0,j}$ each of size $n+1$
Side–A
G_i : $1 \leq i \leq n$
$A_{i,n}$ ..... $A_{i,2}$ $A_{i,1}$ $A_{i,1}$ $A_{i,2}$ ..... $A_{i,n}$
$B_{i,n}$ ..... $B_{i,2}$ $B_{i,1}$ $B_{i,1}$ $B_{i,2}$ ..... $B_{i,n}$
n staircases $B_{i,j}$ each of size n or $n+1$
n staircases $A_{i,j}$ each of size n or $n+1$
Side–B

Fig. 2. Reduction in the proof of Theorem 2

Proof. We make the proof by contradiction. We first prove that if the result were false, every permutation of length n would contain a pattern of $\mathcal{C}$ of length *exactly* $k = \lceil n^{0.5+\epsilon} \rceil$. Next, we show that the number of permutations of length n containing one permutation of $\mathcal{C} \bigcap S_k$ as a pattern is strictly less than $n!$.

Suppose that there exist $\epsilon > 0$ and $\mathcal{C}$ a pattern-avoiding permutation class such that for every permutation $\sigma \in S_n$, the longest pattern $\pi \in \mathcal{C}$ of σ has length $|\pi| \geq \lceil |\sigma|^{0.5+\epsilon} \rceil = k$. As $\mathcal{C}$ is closed - every pattern τ of a permutation $\pi \in \mathcal{C}$ is also in $\mathcal{C}$- for every permutation $\sigma \in S_n$ there exists a pattern $\tau \in \mathcal{C}$ of σ whose length is *exactly* $|\tau| = k$.

But, by [19], there exists a constant c, which depends only on the patterns forbidden in $\mathcal{C}$, such that the number of permutations in $\mathcal{C} \bigcap S_k$ is at most c^k. By Lemma 3, for each permutation in $\mathcal{C} \bigcap S_k$, the number of permutations in S_n having this permutation as a pattern is at most $(n-k)!\binom{n}{k}^2$. Thus the number of permutations in S_n having a pattern in $\mathcal{C} \bigcap S_{\geq k}$ is at most $c^k(n-k)!\binom{n}{k}^2$. But with $k = \lceil n^{0.5+\epsilon} \rceil$, $c^k(n-k)!\binom{n}{k}^2 = o\left(n^{n^{1-2\epsilon}}\right) = o\left(n!\right)$. Note that a similar proof is given in [12] for finding the smallest permutation containing all patterns of a given length. □

Corollary 1. *The* LCP *Problem cannot be approximated with a tighter ratio than* $\sqrt{\mathsf{opt}}$ *by the* LC$\mathcal{C}$P *Problem, where* $\mathcal{C}$ *is a pattern-avoiding permutation class, and* opt *is the size of an optimal solution to the* LCP *Problem.*

Proof. Consider the LCP Problem between σ_n and σ_n where $(\sigma_n)_{n\in\mathbb{N}}$ is the sequence defined in Theorem 3. Then the optimal solution to the LCP Problem is σ_n. But the solution to the LC$\mathcal{C}$P Problem is a longest pattern of σ_n belonging to the class $\mathcal{C}$ for example π_n. By Theorem 3, such a pattern may have size $\sqrt{|\sigma_n|}$ asymptotically. □

5 Conclusion

We have given a polynomial-time algorithm for LC$\mathcal{S}ep$P on K permutations and shown a hardness result for this problem if the number of input permutations is not fixed.

Some classes $\mathcal{C}$ of permutations are known for which even the *Recognition* Problem (*i.e.*, deciding if a permutation belongs to $\mathcal{C}$) is **NP**–hard, so that LC$\mathcal{C}$P on K permutations must be **NP**–hard for those classes. [1] gives the example of the class of 4-stack sortable permutations.

However, we are not aware of any example of *finitely based* pattern-avoiding permutation classes (with a finite number of excluded patterns) for which the Recognition Problem is **NP**–hard. Thus an open question is to know if the LC$\mathcal{C}$P problem for K permutations is polynomial-time solvable for any finitely based $\mathcal{C}$, or to exhibit such a class $\mathcal{C}$ for which this problem is **NP**–hard.

References

1. Albert, M.H., Aldred, R.E.L., Atkinson, M.D., van Ditmarsch, H.P., Handley, B.D., Handley, C.C., Opatrny, J.: Longest subsequences in permutations. Australian J. Combinatorics 28, 225–238 (2003)
2. Albert, M.H., Atkinson, M.D., Klazar, M.: The enumeration of simple permutations. Journal of integer sequences, 6(4) (2003)
3. Albert, M.H., Aldred, R.E.L., Atkinson, M.D., Holton, D.A.: Algorithms for pattern involvement in permutations. In: Eades, P., Takaoka, T. (eds.) ISAAC 2001. LNCS, vol. 2223, pp. 355–366. Springer, Heidelberg (2001)
4. Bérard, S., Bergeron, A., Chauve, C., Paul, C.: Perfect sorting by reversals is not always difficult. IEEE/ACM Transactions on Computational Biology and Bioinformatics, 4(1) (2007)
5. Bergeron, A., Chauve, C., de Montgolfier, F., Raffinot, M.: Computing common intervals of permutations, with applications to modular decomposition of graphs. In: Brodal, G.S., Leonardi, S. (eds.) ESA 2005. LNCS, vol. 3669, pp. 779–790. Springer, Heidelberg (2005)
6. Bose, P., Buss, J.F., Lubiw, A.: Pattern matching for permutations. Information Processing Letters 65(5), 277–283 (1998)
7. Bouvel, M., Rossin, D.: The longest common pattern problem for two permutations. Pure Mathematics and Applications, to be published, arXiv:math.CO/0611679 (2007)

8. Bui-Xuan, B.-M., Habib, M., Paul, C.: Revisiting T. Uno and M. Yagiura's algorithm. In: Deng, X., Du, D.-Z. (eds.) ISAAC 2005. LNCS, vol. 3827, pp. 146–155. Springer, Heidelberg (2005)
9. Chang, M.-S, Wang, G.-H: Efficient algorithms for the maximum weight clique and maximum weight independent set problems on permutation graphs. Information Processing Letters 43, 293–295 (1992)
10. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 2nd edn. MIT Press and McGraw-Hil, Cambridge, MA,New York (2001)
11. Ehrenfeucht, A., Harj, T., ten Pas, P., Rozenberg, G.: Permutations, parenthesis words, and Schröder numbers. Discrete Mathematics 190, 259–264 (1998)
12. Eriksson, H., Eriksson, K., Linusson, S., Wästlund, J.: Dense packing of patterns in a permutation. Annals of Combinatorics (to appear)
13. Garey, M.R., Johnson, D.S.: Computers and Intractablility: A Guide to the Theory of NP-Completeness. W. H. Freeman, San Francisco (1979)
14. Ibarra, L.: Finding pattern matchings for permutations. Information Processing Letters 61, 293–295 (1997)
15. Knuth, D.E.: Fundamental Algorithms, 3rd edn. The Art of Computer Programming, vol. 1. Addison-Wesley, Reading (1973)
16. Kubica, M., Rizzi, R., Vialette, S., Walen, T.: Approximation of rna multiple structural alignment. In: Lewenstein, M., Valiente, G. (eds.) CPM 2006. LNCS, vol. 4009, pp. 211–222. Springer, Heidelberg (2006)
17. Lovász, L.: Combinatorial problems and exercices. North-Holland, Amsterdam (1979)
18. Maier, D.: The Complexity of Some Problems on Subsequences and Supersequences. J. ACM 25, 322–336 (1978)
19. Marcus, A., Tardos, G.: Excluded permutation matrices and the Stanley-Wilf conjecture. J. Combin. Th. A 107, 153–160 (2004)
20. Rotem, D.: Stack-sortable permutations. Discrete Math. 33, 185–196 (1981)
21. West, J.: Generating trees and the Catalan and Schröder numbers. Discrete Mathematics 146, 247–262 (1995)

Suffix Arrays on Words*

Paolo Ferragina[1] and Johannes Fischer[2]

[1] Dipartimento di Informatica, University of Pisa
ferragina@di.unipi.it
[2] Institut für Informatik, Ludwig-Maximilians-Universität München
Johannes.Fischer@bio.ifi.lmu.de

Abstract. Surprisingly enough, it is not yet known how to build *directly* a suffix array that indexes just the k positions at word-boundaries of a text $T[1, n]$, taking $O(n)$ time and $O(k)$ space in addition to T. We propose a class-note solution to this problem that achieves such optimal time and space bounds. Word-based versions of indexes achieving the same time/space bounds were already known for suffix trees [1,2] and (compact) DAWGs [3,4]. Our solution inherits the simplicity and efficiency of suffix arrays, with respect to such other word-indexes, and thus it foresees applications in *word-based approaches* to data compression [5] and computational linguistics [6]. To support this, we have run a large set of experiments showing that word-based suffix arrays may be constructed twice as fast as their full-text counterparts, and with a working space as low as 20%. The space reduction of the final word-based suffix array impacts also in their query time (i.e. less random access binary-search steps!), being faster by a factor of up to 3.

1 Introduction

One of the most important tasks in classical string matching is to construct an *index* on the input data in order to answer future queries faster. Well-known examples of such indexes include suffix-trees, word graphs and suffix arrays (see e.g. [7]). Despite the extensive research that has been done in the last three or four decades, this topic has recently re-gained popularity with the rise of compressed indexes [8] and new applications such as data compression, text mining and computational linguistics.

However, all of the indexes mentioned so far are *full-text* indexes, in the sense that they index any position in the text and thus allow to search for occurrences of patterns starting at *arbitrary* text positions. In many situations, deploying the full-text feature might be like using a "cannon to shoot a fly", with undesired negative impacts on both query time and space usage. For example, in

* The first author has been partially supported by the Italian MIUR grant Italy-Israel FIRB "Pattern Discovery Algorithms in Discrete Structures, with Applications to Bioinformatics", and by the Yahoo! Research grant on "Data compression and indexing in hierarchical memories". The second autor has been partially funded by the German Research Foundation (DFG, Bioinformatics Initiative).

European languages, words are separated by special symbols such as spaces or punctation signs; in a dictionary of URLs, "words" are separated by dots and slashes. In both cases, the results found by a word-based search with a full-text index should have to be filtered out by discarding those results that do not occur at word boundaries. Possibly a time-costly step! Additionally, indexing every text position would affect the overall space occupancy of the index, with an increase in the space complexity which could be estimated in practice as a multiplicative factor 5–6, given the average word length in linguistic texts. Of course, the use of word-based indexes is not limited to pattern searches, as they have been successfully used in many other contexts, like data compression [5] and computational linguistics [6], just to cite a few.

Surprisingly enough, *word based* indexes have been introduced only recently in the string-matching literature [1], although they were very famous in Information Retrieval many years before [9]. The basic idea underlying their design consists of storing just a *subset* of the text positions, namely the ones that correspond to word beginnings. As we observed above, it is easy to construct such indexes if $O(n)$ additional space is allowed at construction time (n being the text size): Simply build the index for every position in the text, and then discard those positions which do not correspond to word beginnings. Unfortunately, such a simple (and common, among practitioners!) approach is not space optimal. In fact $O(n)$ construction time cannot be improved, because this is the time needed to scan the input text. But $O(n)$ additional working space (other than the indexed text and the final suffix array) seems too much because the final index will need $O(k)$ space, where k is the number of words in the indexed text. This is an interesting issue, not only theoretically, because "*... we have seen many papers in which the index simply 'is', without discussion of how it was created. But for an indexing scheme to be useful it must be possible for the index to be constructed in a reasonable amount of time.*" [10] And in fact, the working-space occupancy of construction algorithms for full-text indexes is yet a primary concern and an active field of research [11].

The first result addressing this issue in the word-based indexing realm is due to Anderson et al. [1] who showed that the *word suffix tree* can be constructed in $O(n)$ *expected* time and $O(k)$ working space. In 2006, Inenaga and Takeda [2] improved this result by providing an on-line algorithm which runs in $O(n)$ time in the worst case and $O(k)$ space in addition to the indexed text. They also gave two alternative indexing structures [3, 4] which are generalizations of Directed Acyclic Word Graphs (DAWGs) or compact DAWGs, respectively. All three construction methods are variations of the construction algorithms for (usual) suffix trees [12], DAWGs [13] and CDAWGs [14], respectively.

The only missing item in this quartet is a word-based analog of the suffix array, a gap which we close in this paper. We emphasize the fact that, as it is the case with full-text suffix arrays (see e.g. [15]), we get a class-note solution which is simple and practically effective, thus surpassing the previous ones by all means.

A comment is in order before detailing our contribution. A more general problem than word-based string matching is that of *sparse* string matching, where the set of points to be indexed is given as an arbitrary subset of all n text positions, not necessarily coinciding with the word boundaries. Although the authors of [2,3,4] claim that their indexes can solve this task as well, they did not take into account an exponential factor [16]. To the best of our knowledge, this problem is still open. The only step in this direction has been made by Kärkkäinen and Ukkonen [17] who considered the special case where the indexed positions are evenly spaced.

1.1 Our Contributions

We define a new data structure called the *word(-based) suffix array* and show how it can be constructed *directly* in optimal time and space; i.e., without first constructing the sparse suffix tree. The size of the structure is k RAM words, and at no point during its construction more than $O(k)$ space (in addition to the text) is needed. This is interesting in theory because we could compress the text by means of [18] and then build the word-based index in space $O(k) + nH_h + o(n)$ bits and $O(n)$ time, simultaneously over all $h = o(\log n)$, where H_h is the h-th order empirical entropy of the indexed text (alphabet is assumed to have constant size). If the number k of indexed "words" is relatively "small", namely $k = o(n/\log n)$, this index would take the same space as the best compressed indexes (cf. [8]), but it would need less space to be constructed.

As far as pattern-queries are concerned, it is easy to adapt to word-based suffix arrays the classical pattern searches over full-text suffix arrays. For patterns of length m, we then easily show that *counting* queries take $O(m \log k)$ time, or $O(m + \log k)$ if an additional array of size k is used. Note that this reduces the number of costly binary search step by $O(\log(n/k))$ compared with full-text suffix arrays. *Reporting* queries take $O(occ)$ additional time, where occ is the number of word occurrences reported. We then show that the addition of another data structure, similar to the Enhanced Suffix Array [19], lowers these time bounds to $O(m)$ and $O(m + occ)$, respectively.

In order to highlight the simplicity, and hence practicality, of our word-based suffix array we test it over various datasets, which cover some typical applications of word-based indexes: natural and artificial language, structured data and prefix-search on hosts/domains in URLs. Construction time is twice faster than state-of-the-art algorithms applied to full-text suffix arrays, and the working space is lowered by 20%. As query time is faster by up to a factor 3 *without* post-filtering the word-aligned occurrences, and up to 5 orders of magnitude *including* post-filtering, we exclude the idea of using a full-text suffix array for finding word-aligned occurrences already at this point.

2 Definitions

Throughout this article we let T be a text of length n over a constant-sized alphabet Σ. We further assume that certain characters from a constant-sized subset W of the alphabet act as *word boundaries*, thus dividing T in a natural sense into k tokens, hereafter called *words*. Now let I be the set of positions

bucket->	1			2		3	4		5	
	1	2	3	4	5	6	7	8	9	10
A=	4	9	22	6	24	18	1	11	14	27
	a	a	a	a	a	a	a	a	b	b
	#	#	#	a	a	a	b	b	a	a
	a	a	a	#	#	b	#	#	a	a
	a	b	a	a	b	#	a	b	#	#
	#	#	#	#	a	a	#	a	a	
	a	b	b	a	a	#	a	a	a	
	#	a	a	b	#	a	a	#	b	
	.	.	.	.		.	.	.	.	
	.	.	.	.		.	.	.	.	

Fig. 1. The initial radix-sort in step 1

	1^{1}	2^{4}	3^{6}	4^{9}	5^{11}	6^{14}	7^{18}	8^{22}	9^{24}	10^{27}
T'=	**4**	**1**	**2**	**1**	**4**	**5**	**3**	**1**	**2**	**5**
SA=	2	8	4	3	9	7	1	5	10	6
	1	**1**	**1**	**2**	**2**	**3**	**4**	**5**	**5**	**5**
	2	**2**	**4**	**1**	**5**	**1**	**1**	**3**		**3**
	1	**5**	**5**	**4**		**2**	**2**	**1**		**1**
	4		**3**	**5**		**5**	**1**	**2**		**2**
	5		**1**	**3**			**4**	**5**		**5**
	3		**2**	**1**			**5**			
	1		**5**	**2**			**3**			
	2			**5**			**1**			
	5						**2**			
							5			

Fig. 2. The new text T' and its (full-text) suffix array SA

where new words start: $1 \in I$ and $i \in I \setminus \{1\} \iff T_{i-1} \in W$. (The first position of the text is always taken to be the beginning of a new word.) Similar to [2] we define the set of all suffixes of T starting at word boundaries as $\mathit{Suffix}_I(T) = \{T_{i..n} : i \in I\}$. Then the *word suffix array* $A[1..k]$ is a permutation of I such that $T_{A[i-1]..n} < T_{A[i]..n}$ for all $1 < i \leq k$; i.e., A represents the lexicographic order of all suffixes in $\mathit{Suffix}_I(T)$.

Definition 1 (Word Aligned String Matching). *For a given pattern P of length m let $O_P \subseteq I$ be the set of word-aligned positions where P occurs in T: $i \in O_P$ iff $T_{i..n}$ is prefixed by P and $i \in I$. Then the tasks of word aligned string matching are (1) to answer whether or not O_P is empty* (decision query)*, (2) to return the size of O_P* (counting query)*, and (3) to enumerate the members of O_P in some order* (reporting query)*.*

3 Optimal Construction of the Word Suffix Array

This section describes the optimal $O(n)$ time and $O(k)$ space algorithm to construct the word suffix array. For simplicity, we describe the algorithm with only one word separator (namely #). The reader should note, however, that all steps are valid and can be computed in the same time bounds if we have more than one (but constantly many) word separators. We also assume that the set I of positions to be indexed is implemented as an increasingly sorted array.

As a running example for the algorithm we use the text $T = ab\#a\#aa\#a\#ab\#baa\#aab\#a\#aa\#baa\#$, so $I = [1, 4, 6, 9, 11, 14, 18, 22, 24, 27]$.

1. The goal of this step is to establish a "coarse" sorting of the suffixes from $\mathit{Suffix}_I(T)$. In particular, we want to sort these suffixes using their *first word* as the sort key. To do so, initialize the array $A[1..k] = I$. Then perform a radix-sort of the elements in A: at each level $l \geq 0$, bucket-sort the array A using $T_{A[i]+l}$ as the sort key for $A[i]$. Stop the recursion when a bucket contains only one element, or when a bucket consists only of suffixes starting

	1	2	3	4	5	6	7	8	9	10
A=	4	22	9	6	24	18	1	11	27	14
	a	a	a	a	a	a	a	a	b	b
	#	#	#	a	a	a	b	b	a	a
	a	a	a	#	#	b	#	#	a	a
	a	a	b	a	b	#	a	b	#	#
	#	#	#	#	a	a	#	a		a
	a	b	b	a	a	#	a	a		a
	#	a	a	b	#	a	a	#		b
	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮		⋮

Fig. 3. The final word suffix array

```
LCP[1] ← −1, $h$ ← 0
for $i \leftarrow 1, \ldots, k$ do
    $p \leftarrow A^{-1}[i], h \leftarrow \max\{0, h - A[p]\}$
    if $p > 1$ then
        while $T_{A[p]+h} = T_{A[p-1]+h}$ do
            $h \leftarrow h + 1$
        end
        LCP[$p$] ← $h$
    end
    $h \leftarrow h + A[p]$
end
```

Fig. 4. $O(n)$-time longest common prefix computation using $O(k)$ space (adapted from [20])

with $w\#$ for some $w \in (\Sigma \setminus \{\#\})^\star$. Since each character from T is involved in at most one comparison, this step takes $O(n)$ time. See Fig. 1 for an example.

2. Construct a new text $T' = b(I[1])b(I[2]) \ldots b(I[k])$, where $b(I[i])$ is the bucket-number (after step 1) of suffix $T_{I[i]..n} \in \mathit{Suffix}_I(T)$. In our example, $T' = \mathbf{4121453125}$. (We use boldface letters to emphasize the fact that we are using a new alphabet.) This step can clearly be implemented in $O(k)$ time.
3. We now build the (full-text) suffix array SA for T'. Because the linear-time construction algorithms for suffix arrays (e.g., [15]) work for integer alphabets too, we can employ any of them to take $O(k)$ time. See Fig. 2 for an example. In this figure, we have attached to each position in the new text T' the corresponding position in T as a superscript (i.e., the array I), which will be useful in the next step.
4. This step derives the word suffix array A from SA. Scan SA from left to right and write the corresponding suffix to A: $A[i] = I[\mathsf{SA}[i]]$. This step clearly takes $O(k)$ time. See Figure 3 for an example.

Theorem 1. *Given a text T of length n consisting of k words drawn from a constant-sized alphabet Σ, the word suffix array for T can be constructed in optimal $O(n)$ time and $O(k)$ extra space.*

Proof. Time and space bounds have already been discussed in the description of the algorithm; it only remains to prove the correctness. This means that we have to prove $T_{A[i-1]..n} \leq T_{A[i]..n}$ for all $1 < i \leq k$ after step 4. Note that after step 1 we have $T_{A[i-1]..x} \leq T_{A[i]..y}$, where x and y are defined so that T_x and T_y is the first # after $T_{A[i-1]}$ and $T_{A[i]}$, respectively. We now show that steps 2–4 refine this ordering for buckets of size greater than one. In other words, we wish to show that in step 4, buckets $[l : r]$ sharing a common prefix $T_{A[i]..x}$ with T_x being the first # for all $l \leq i \leq r$ are sorted using the lexicographic order of $T_{x+1..n}$ as a sort key. But this is simple: because the newly constructed text T' from step 2 respects the order of $T_{A[i]..x}$, and because step 3 establishes the correct lexicographic order of T', the $I[\mathsf{SA}[i]]$'s are the correct sort keys for step 4. □

Table 1. Different methods for retrieving all *occ* occurrences of a pattern at word-boundaries. The full-text suffix array would have the same time- and space-bounds, with k substituted by $n >> k$, and occ by $occ' >> occ$, where occ' is the number of not necessarily word-aligned occurrences of the pattern.

method	space usage (words)	time bounds
bin-naive	k	$O(m \log k + occ)$
bin-improved	$(1 + C)k$, $C \leq 1$	$O((m - \log(Ck)) \log k + occ)$
bin-lcp	$2k$	$O(m + \log k + occ)$
esa-search	$2k + O(k/\log k)$	$O(m\|\Sigma\| + occ)$

To further reduce the required space we can think of compressing T before applying the above construction algorithm, by adopting an entropy-bounded storage scheme [18] which allows constant-time access to any of its $O(\log n)$ contiguous bits. This implies the following:

Corollary 1. *The word suffix array can be built in* $3k \log n + nH_h(T) + o(n)$ *bits and* $O(n)$ *time, where* $H_h(T)$ *is the hth order empirical entropy of* T. *For any* $k = o(n/\log n)$*, the space needed to build and store this data structure is* $nH_h + o(n)$ *bits, simultaneously over all* $h = o(\log n)$.

This result is interesting because it says that, in the case of a tokenized text with long words on average (e.g. dictionary of URLs), the word-based suffix array takes the same space as the best compressed indexes (cf. [8]), but it would need less space to be constructed.

4 Searching in the Word Suffix Array

We now consider how to search for the word-aligned *occ* occurrences of a pattern $P[1, m]$ in the text $T[1, n]$. As searching the word suffix array can be done with the same methods as in the full-text suffix array we keep the discussion short (see also Table 1); the purpose of this section is the completeness of exposition, and to prepare for the experiments in the following section.

Here we also introduce the notion of word-based LCP-array and show that it can be computed in $O(n)$ time and $O(k)$ space. We emphasize that enhancing the word suffix array with the LCP-array actually yields more functionality than just improved string-matching performance. As an example, with the LCP-array it is possible to simulate bottom-up traversals of the corresponding word suffix tree, and augmenting this further allows us also to simulate top-down traversals [19]. Additionally, in the vein of [21, Section 5.3.2], we can derive the word suffix tree from arrays LCP and A. This yields a simple, space-efficient and memory-friendly (in the sense that nodes tend to be stored in the vicinity of their predecessor/children) alternative to the algorithm in [2].

Searching in $O(m \log k)$ time. Because A is sorted lexicographically, it can be binary-searched in a similar manner to the original search-algorithm from

Manber and Myers [22]. We can also apply the two heuristics proposed there to speed up the search in practice (though not in theory): the first builds an additional array of size $|\Sigma|^K$ ($K = \log_{|\Sigma|}(Ck)$ for some $C \leq 1$) to narrow down the initial search interval in A, and the second one reduces the number of character comparisons by remembering the number of matching characters from T and P that have been seen so far.

Searching in $O(m + \log k)$ time. Like in the original article [22] the idea is to pre-compute the longest common prefixes of $T_{A[(l+r)/2]..n}$ with both $T_{A[l]..n}$ and $T_{A[r]..n}$ for all possible search intervals $[l : r]$. Footnote 6 in [22] actually shows that only one of these values needs to be stored, so the additional space needed is one array of size k. Because both the precomputation and the search algorithm are unchanged, we refer the reader to [22] for a complete description of the algorithm.

Searching in $O(m|\Sigma|)$ time. While the previous two searching algorithms have a searching time that is *independent* of the alphabet size, we show in this section how to locate the search interval of P within A in $O(m|\Sigma|)$. We note that for a constant alphabet this actually yields *optimal* $O(m)$ counting time and *optimal* $O(m + occ)$ reporting time.

Define the *LCP-array* $\mathsf{LCP}[1..k]$ as follows: $\mathsf{LCP}[1] = -1$ and for $i > 1$, $\mathsf{LCP}[i]$ is the length of the longest common prefix of the suffixes $T_{A[i-1]..n}$ and $T_{A[i]..n}$. We will now show that this LCP-table can be computed in $O(n)$ time in the order of *inverse word suffix array* A^{-1} which is defined as $A[A^{-1}[i]] = I[i]$; i.e., $A^{-1}[i]$ tells us where the i'th-longest suffix among all *indexed* suffixes from T can be found in A. A^{-1} can be computed in $O(k)$ time as a by-product of the construction algorithm (Section 3). In our example, $A^{-1} = [7, 1, 4, 3, 8, 10, 6, 2, 5, 9]$.

Figure 4 shows how to compute the LCP-array in $O(n)$ time. It is actually a generalization of the $O(n)$-algorithm for lcp-computation in (full-text) suffix arrays [20]. The difference from [20] is that the original algorithm assumes that when going from position p (here $A[p] = i$) to position $p' = A^{-1}[i + 1]$ (hence $A[p'] = i + 1$), the difference in length between $T_{A[p]..n}$ and $T_{A[p']..n}$ is exactly one, whereas in our case this difference may be larger, namely $A[p'] - A[p]$. This means that when going from position p to p' the lcp can decrease by at most $A[p'] - A[p]$ (instead of 1); we account for this fact by adding $A[p]$ to h (line 10) and subtracting p' (i.e. the new p) in the next iteration of the loop (line 3). At any iteration, variable h holds the length of the prefix that $T_{A[p]..n}$ and $T_{A[p-1]..n}$ have in common. Since each text character is involved in at most 2 comparisons, the $O(n)$ time bound easily follows.

Now in order to achieve $O(m|\Sigma|)$ matching time, use the RMQ-based variant of the Enhanced Suffix Array [19] proposed by [23]. This requires $o(k)$ additional space and can be computed in $O(k)$ time.

5 Experimental Results

The aim is to show the practicality of our method. We implemented the word suffix array in C++ (`www.bio.ifi.lmu.de/~{}fischer/wordSA.tgz`). Instead

Table 2. Our Test-files and their characteristics. In the word separator column, LF stands for "line feed", SPC for "space" and TAB for "tabulator".

dataset	size (MB)	$\|\Sigma\|$	word separators used	#words	different words	avg. length
English	333	239	LF, SPC, -	67,868,085	1,220,481	5.27
XML	282	97	SPC, /, <, >, "	53,167,421	2,257,660	5.60
sources	201	230	[10 in total]	53,021,263	2,056,864	3.98
URLs	70	67	LF, /	5,563,810	533,809	13.04
random	250	2	SPC	10,000,001	9,339,339	26.0

of using a linear time algorithm for the construction of suffix arrays, we opted for the method from Larsson and Sadakane [24]. We implemented the search strategies bin-naive, bin-improved, bin-lcp and esa-search from Table 1. Unfortunately, we could not compare to the other word-based indexes [2,3,4] because there are no publicly available implementations. For bin-improved we chose $C = 1/4$, so the index occupies $1.25k$ memory words (apart from T, which takes n bytes). For the RMQ-preprocessing of the esa-search we used the method from Alstrup et al. [25] which is fast in practice, while still being relatively space-conscious (about $1.5k$ words). With the LCP-array and the text this makes a total of $\approx 3.5k$ words.

We tested our algorithms on the files "English", "XML", and "sources" from the Pizza&Chili site [26] (some of them truncated), plus one file of URLs from the `.eu` domain [27]. To test the search algorithms on a small alphabet, we also generated an artificial dataset by taking words of random length (uniformly from 20 to 30) and letters uniformly from $\Sigma = \{a, b\}$. See Table 2 for the characteristics of the evaluated datasets.

Table 3 shows the space consumption and the preprocessing times for the four different search methods. Concerning the space, the first four columns under "space consumption" denote the space (in MB) of the final index (including the text) for the different search algorithms it can subsequently support. Column labeled "peak 1–3" gives the peak memory usage at construction time for search algorithms 1–3; the peak usage for search algorithm 4 is the same as that of the final index. Concerning the construction time, most part of the preprocessing is needed for the construction of the pure word suffix array (method 1); the times for methods 2–4 are only slightly longer than that for method 1.

To see the advantage of our method over the naive algorithm which prunes the full-text suffix array to obtain the word suffix array, Table 4 shows the construction times and peak space consumption of two state-of-the-art algorithms for constructing (full-text) suffix arrays, MSufSort-3.0 [28] and deep-shallow [29]. Note that the figures given in Table 4 are pure construction times for the *full-text* suffix array; pruning this is neither included in time nor space. First look at the peak space consumption in Table 4. MSufSort needs about $7n$ bytes if the input text cannot be overwritten (it therefore failed for the largest dataset), and deep-shallow needs about $5n$ bytes. These two columns should be compared with the column labeled "peak 1–3" in Table 3, because this column gives the space

Table 3. Space consumption (including the text) and preprocessing times for the 4 different search algorithms: bin-naive (1), bin-improved (2), bin-lcp (3), esa-search (4)

	space consumption (MB)					preprocessing times (in sec)			
dataset	1	2	3	4	peak 1–3	1	2	3	4
English	600.2	664.2	859.1	1,296.0	1,118.0	533.64	558.89	631.57	639.95
XML	485.2	485.5	688.1	1,024.3	890.9	328.38	341.95	370.88	377.54
sources	403.4	403.6	605.6	940.6	807.9	281.12	295.95	323.04	329.74
URLs	90.4	90.7	111.6	145.1	132.9	45.75	46.60	47.14	47.97
random	286.1	286.3	324.2	385.0	362.4	224.75	228.17	239.89	241.33

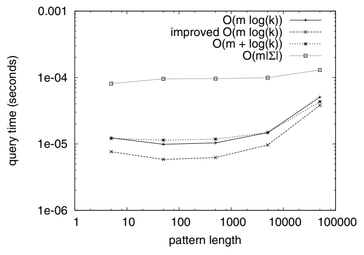
needed to construct the pure word suffix array (i.e., $12k + n$ bytes in our implementation). For all but one data set our method uses significantly less space than both MSufSort (20.9–57.4%) and deep-shallow (36.5–81.9%). For the construction time, compare the last two columns in Table 4 with the preprocessing time for method 1 in Table 3. Again, our method is almost always faster (49.6–90.1% and 64.4–80.2% better than deep-shallow and MSufSort, respectively); the difference would be even larger if we did include the time needed for pruning the full-text suffix array.

Table 4. Space consumption (including the text) and construction times for two different state-of-the-art methods to construct (full-text) suffix arrays

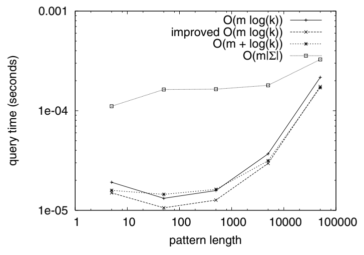
	peak space consumption (MB)		construction time	
dataset	MSufSort-3.0	deep-shallow	MSufSort-3.0	deep-shallow
English	—	1,365.3	—	755.8
XML	1,976.9	1,129.7	363.9	410.8
sources	1,407.7	804.4	193.4	260.6
URLs	484.3	276.7	75.2	71.0
random	1,735.6	991.8	452.7	332.2

We finally tested the different search strategies. In particular, we posed 300,000 counting queries to each index (i.e., determining the interval of pattern P in A) for patterns of length 5, 50, 500, 5,000, and 50,000. The results can be seen in Fig. 5.[1] We differentiated between *random* patterns (left hand side of Fig. 5) and *occurring* patterns (right hand side). There are several interesting points to note. First, the improved $O(m \log k)$-algorithm is almost always the fastest. Second, the $O(m|\Sigma|)$ is not competitive with the other methods, apart from very long patterns or a very small alphabet (Subfig. (h)). And third, the query time for the methods based on binary search (1–3) can actually be higher for short patterns than for long patterns (Fig. (a)-(b)). This is the effect of narrowing down the search for the right border when searching for the left one.

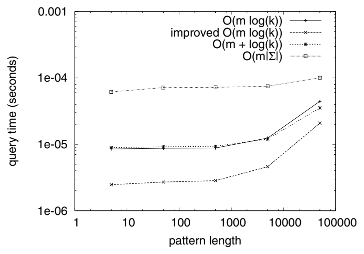
[1] We omit the results for the sources-dataset as they strongly resemble those for the URL-dataset.



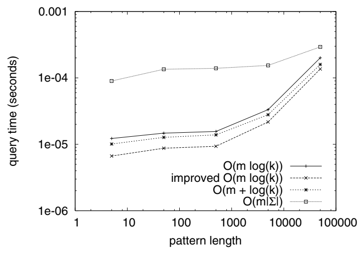
(a) English, random patterns.



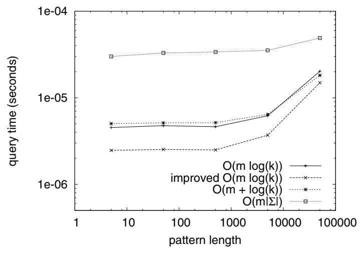
(b) English, occurring patterns.



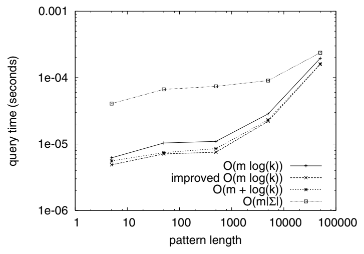
(c) XML, random patterns.



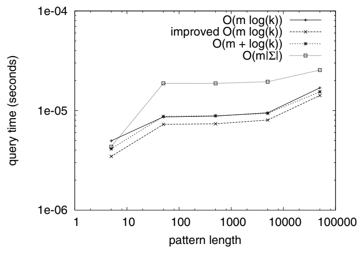
(d) XML, occurring patterns.



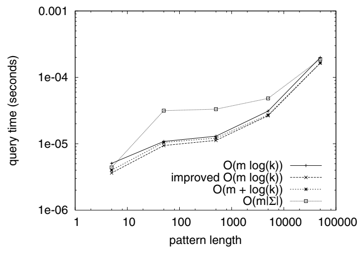
(e) URLs, random patterns.



(f) URLs, occurring patterns.



(g) Random words, random patterns.



(h) Random words, occurring patterns.

Fig. 5. Practical performance of the algorithms from Section 4 (average over 300,000 counting queries; time for index construction is not included). Axes have a log-scale.

6 Conclusions

We have seen a space- and time-optimal algorithm to construct suffix arrays on words. The most striking property was the simplicity of our approach, reflected in the good practical performance. This supersedes all the other known approaches based on suffix trees, DAWG and compact DAWG.

As future research issues we point out the following two. In a similar manner as we compressed T (Corollary 1), one could compress the word-based suffix array A by probably resorting the ideas on word-based Burrows-Wheeler Transform [5] and alphabet-friendly compressed indexes [8]. This would have an impact not only in terms of space occupancy, but also on the search performance of those indexes because they execute $O(1)$ random memory-accesses per searched/scanned character. With a word-based index this could be turned to $O(1)$ random memory-accesses per searched/scanned word, with a significant practical speed-up in the case of very large texts possibly residing on disk.

The second research issue regards the *sparse* string-matching problem in which the set of points to be indexed is given as an arbitrary set, not necessarily coinciding with word boundaries. As pointed out in the introduction, this problem is still open, though being relevant for texts such as biological sequences where natural word boundaries do not occur.

References

1. Andersson, A., Larsson, N.J., Swanson, K.: Suffix Trees on Words. Algorithmica 23(3), 246–260 (1999)
2. Inenaga, S., Takeda, M.: On-Line Linear-Time Construction of Word Suffix Trees. In: Lewenstein, M., Valiente, G. (eds.) CPM 2006. LNCS, vol. 4009, pp. 60–71. Springer, Heidelberg (2006)
3. Inenaga, S., Takeda, M.: Sparse Directed Acyclic Word Graphs. In: Crestani, F., Ferragina, P., Sanderson, M. (eds.) SPIRE 2006. LNCS, vol. 4209, pp. 61–73. Springer, Heidelberg (2006)
4. Inenaga, S., Takeda, M.: Sparse compact directed acyclic word graphs. In: Stringology, pp. 197–211 (2006)
5. Yugo, R., Isal, K., Moffat, A.: Word-based block-sorting text compression. In: Australasian Conference on Computer Science, pp. 92–99. IEEE Press, New York (2001)
6. Yamamoto, M., Church, K.W: Using suffix arrays to compute term frequency and document frequency for all substrings in a corpus. Computational Linguistics 27(1), 1–30 (2001)
7. Gusfield, D.: Algorithms on Strings, Trees, and Sequences. Cambridge University Press, Cambridge (1997)
8. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Computing Surveys (to appear) Preliminary version available at http://www.dcc.uchile.cl/~gnavarro/ps/acmcs06.ps.gz
9. Witten, I.H, Moffat, A., Bell, T.C: Managing Gigabytes: Compressing and Indexing Documents and Images, 2nd edn. Morgan Kaufmann, San Francisco (1999)
10. Zobel, J., Moffat, A., Ramamohanarao, K.: Guidelines for Presentation and Comparison of Indexing Techniques. SIGMOD Record 25(3), 10–15 (1996)

11. Hon, W.K., Sadakane, K., Sung, W.K.: Breaking a Time-and-Space Barrier in Constructing Full-Text Indices. In: Proc. FOCS, pp. 251–260. IEEE Computer Society, Los Alamitos (2003)
12. Ukkonen, E.: On-line Construction of Suffix Trees. Algorithmica 14(3), 249–260 (1995)
13. Blumer, A., Blumer, J., Haussler, D., Ehrenfeucht, A., Chen, M.T., Seiferas, J.I.: The Smallest Automaton Recognizing the Subwords of a Text. Theor. Comput. Sci. 40, 31–55 (1985)
14. Inenaga, S., Hoshino, H., Shinohara, A., Takeda, M., Arikawa, S., Mauri, G., Pavesi, G.: On-line construction of compact directed acyclic word graphs. Discrete Applied Mathematics 146(2), 156–179 (2005)
15. Kärkkäinen, J., Sanders, P., Burkhardt, S.: Linear Work Suffix Array Construction. J. ACM 53(6), 1–19 (2006)
16. Inenaga, S.: personal communication (December 2006)
17. Kärkkäinen, J., Ukkonen, E.: Sparse Suffix Trees. In: Cai, J.-Y., Wong, C.K. (eds.) COCOON 1996. LNCS, vol. 1090, pp. 219–230. Springer, Heidelberg (1996)
18. Ferragina, P., Venturini, R.: A Simple Storage Scheme for Strings Achieving Entropy Bounds. Theoretical Computer Science 372(1), 115–121 (2007)
19. Abouelhoda, M.I., Kurtz, S., Ohlebusch, E.: Replacing Suffix Trees with Enhanced Suffix Arrays. J. Discrete Algorithms 2(1), 53–86 (2004)
20. Kasai, T., Lee, G., Arimura, H., Arikawa, S., Park, K.: Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications. In: Amir, A., Landau, G.M. (eds.) CPM 2001. LNCS, vol. 2089, pp. 181–192. Springer, Heidelberg (2001)
21. Aluru, S. (ed.): Handbook of Computational Molecular Biology. Chapman & Hall/CRC, Sydney, Australia (2006)
22. Manber, U., Myers, E.W.: Suffix Arrays: A New Method for On-Line String Searches. SIAM J. Comput. 22(5), 935–948 (1993)
23. Fischer, J., Heun, V.: A new succinct representation of RMQ-information and improvements in the enhanced suffix array. In: Proc. ESCAPE. LNCS (to appear)
24. Larsson, N.J., Sadakane, K.: Faster suffix sorting. Technical Report LU-CS-TR:99-214, LUNDFD6/(NFCS-3140)/1–20/(1999), Department of Computer Science, Lund University, Sweden (May 1999)
25. Alstrup, S., Gavoille, C., Kaplan, H., Rauhe, T.: Nearest Common Ancestors: A Survey and a New Distributed Algorithm. In: Proc. SPAA, pp. 258–264. ACM Press, New York (2002)
26. Ferragina, P., Navarro, G.: The Pizza & Chili Corpus. Available at `http://pizzachili.di.unipi.it`, `http://pizzachili.dcc.uchile.cl`
27. Università degli Studi di Milano, Laboratory for Web Algorithmics: URLs from the.eu domain. Available at `http://law.dsi.unimi.it/index.php`
28. Maniscalco, M.A., Puglisi, S.J.: An efficient, versatile approach to suffix sorting. ACM Journal of Experimental Algorithmics (to appear) Available at `http://www.michael-maniscalco.com/msufsort.htm`
29. Manzini, G., Ferragina, P.: Engineering a lightweight suffix array construction algorithm. Algorithmica, 40(1), 33–50 (2004) Available at `http://www.mfn.unipmn.it/~manzini/lightweight`

Efficient Computation of Substring Equivalence Classes with Suffix Arrays

Kazuyuki Narisawa[1], Shunsuke Inenaga[2], Hideo Bannai[1], and Masayuki Takeda[1,3]

[1] Department of Informatics, Kyushu University, Fukuoka 819-0395, Japan
[2] Department of Computer Science and Communication Engineering, Kyushu University, Fukuoka 819-0395, Japan
[3] SORST, Japan Science and Technology Agency (JST)
{k-nari,bannai,takeda}@i.kyushu-u.ac.jp, inenaga@c.csce.kyushu-u.ac.jp

Abstract. This paper considers enumeration of substring equivalence classes introduced by Blumer et al. [1]. They used the equivalence classes to define an index structure called compact directed acyclic word graphs (CDAWGs). In text analysis, considering these equivalence classes is useful since they group together redundant substrings with essentially identical occurrences. In this paper, we present how to enumerate those equivalence classes using suffix arrays. Our algorithm uses rank and lcp arrays for traversing the corresponding suffix trees, but does not need any other additional data structure. The algorithm runs in linear time in the length of the input string. We show experimental results comparing the running times and space consumptions of our algorithm, suffix tree and CDAWG based approaches.

1 Introduction

Finding distinct features from text data is an important approach for text analysis, with various applications in literary studies, genome studies, and spam detection [2]. In biological sequences and non-western languages such as Japanese and Chinese, word boundaries do not exist, and thus all substrings of the text are subject to analysis. However, a given text contains too many substrings to browse or analyze. A reasonable approach is to partition the set of substrings into equivalence classes under the equivalence relation of [1] so that an expert can examine the classes one by one [3]. This equivalence relation groups together substrings that correspond to essentially identical occurrences in the text. Such a partitioning is very beneficial for various text mining approaches whose mining criterion is based on occurrence frequencies, since each element in a given equivalence class will have the same occurrence frequency.

In this paper, we develop an efficient algorithm for enumerating the equivalence classes of a given string, as well as useful statistics such as frequency and size for each class. Although the number of equivalence classes in a string w of length n is at most $n+1$, the total number of elements in the equivalence classes is $O(n^2)$, that is, the number of substrings in w. However, each equivalence class can be expressed by a unique maximal (longest) element and multiple minimal elements. Further, these elements can be expressed by a pair of integers representing the beginning and end positions in the string. Thus, we consider these succinct expressions of the equivalence classes, which require only $O(n)$ space. The succinct expressions can easily be computed using the CDAWG data structure proposed by [1], which is an acyclic graph structure whose nodes correspond to the equivalence classes. Although CDAWGs can be constructed in $O(n)$ time and space [4], we present a more efficient algorithm based on suffix arrays.

In Section 3, we first describe an algorithm using suffix trees with suffix links (Algorithm 1), for computing the succinct expressions. Although suffix trees can also be constructed and represented in $O(n)$ time and space [5,6], it has been shown that many algorithms on suffix trees can be efficiently simulated on suffix arrays [7] with the help of auxiliary arrays such as lcp and rank arrays [8,9]. However, previous methods require extra time and space for maintaining suffix link information. In Section 4, we give an algorithm to simulate Algorithm 1 using the suffix, lcp and rank arrays (Algorithm 2). A key feature of this algorithm is that it does not require any extra data structure other than these arrays, making it quite space economical. Section 5 gives results of computational experiments of Algorithm 1, 2, and an algorithm using CDAWGs.

2 Preliminaries

2.1 Notations

Let Σ be a finite set of symbols, called an *alphabet*. An element of Σ^* is called a *string*. Strings x, y and z are said to be a *prefix*, *substring*, and *suffix* of the string $w = xyz$, respectively, and the string w is said to be a *superstring* of substring y. The sets of prefixes, substrings and suffixes of a string w are denoted by $\mathit{Prefix}(w)$, $\mathit{Substr}(w)$ and $\mathit{Suffix}(w)$, respectively. The length of a string w is denoted by $|w|$. The empty string is denoted by ε, that is, $|\varepsilon| = 0$. Let $\Sigma^+ = \Sigma^* - \{\varepsilon\}$. The i-th symbol of a string w is denoted by $w[i]$ for $1 \leq i \leq |w|$, and the substring of w that begins at position i and ends at position j is denoted by $w[i:j]$ for $1 \leq i \leq j \leq |w|$. Also, let $w[i:] = w[i:|w|]$ for $1 \leq i \leq |w|$. For any string $w \in \Sigma^*$ and $x \in \mathit{Substr}(w)$, a *reference pair* of x w.r.t. w is a pair $\langle i, j\rangle$ such that $w[i:j] = x$. For any strings $x, y \in \Sigma^*$, the longest string in $\mathit{Prefix}(x) \cap \mathit{Prefix}(y)$ is called the *longest common prefix* (LCP) of x and y.

2.2 Equivalence Relations on Strings

In this subsection, we recall the equivalence relations introduced by Blumer et al. [10,1], and then state their properties. Throughout this paper, we consider the equivalence classes of the input string w that ends with a distinct symbol \$ that does not appear anywhere else in w. For any string $x \in \mathit{Substr}(w)$, let,

$$\begin{aligned} \mathit{BegPos}(x) &= \{i \mid 1 \leq i \leq |w|, x = w[i:i+|x|-1]\}, \text{ and} \\ \mathit{EndPos}(x) &= \{i \mid 1 \leq i \leq |w|, x = w[i-|x|+1:i]\}. \end{aligned}$$

For any string $y \notin \mathit{Substr}(w)$, let $\mathit{BegPos}(y) = \mathit{Endpos}(y) = \emptyset$.

Now we define two equivalence relations and classes based on *BegPos* and *EndPos*.

Definition 1. *The equivalence relations* $\equiv_L$ *and* $\equiv_R$ *on* Σ^* *are defined by:*

$$x \equiv_L y \Leftrightarrow \mathit{BegPos}(x) = \mathit{BegPos}(y), \ \textit{and}$$
$$x \equiv_R y \Leftrightarrow \mathit{EndPos}(x) = \mathit{EndPos}(y),$$

where x, y *are any strings in* Σ^**. The equivalence class of a string* $x \in \Sigma^*$ *with respect to* $\equiv_L$ *and* $\equiv_R$ *is denoted by* $[x]_{\equiv_L}$ *and* $[x]_{\equiv_R}$*, respectively.*

Notice that any strings not in $\mathit{Substr}(w)$ form one equivalence class under $\equiv_L$, called the degenerate class. Similar arguments hold for $\equiv_R$. The above equivalence classes $[x]_{\equiv_L}$ and $[x]_{\equiv_R}$ correspond to the nodes of *suffix trees* [5] and *directed acyclic word graphs* (*DAWGs*) [10], respectively. For any string $x \in \mathit{Substr}(w)$, let $\overrightarrow{x}$ and $\overleftarrow{x}$ denote the unique longest member of $[x]_{\equiv_L}$ and $[x]_{\equiv_R}$, respectively. For any string $x \in \mathit{Substr}(w)$, let $\overleftrightarrow{x} = \alpha x \beta$ such that $\alpha, \beta \in \Sigma^*$ are the strings satisfying $\overleftarrow{x} = \alpha x$ and $\overrightarrow{x} = x\beta$.

Intuitively, $\overleftrightarrow{x} = \alpha x \beta$ means that:

– Every time x occurs in w, it is preceded by α and followed by β.
– Strings α and β are longest possible.

Note that $\overleftarrow{(\overrightarrow{x})} = \overrightarrow{(\overleftarrow{x})} = \overleftrightarrow{x}$.

Now we define another equivalence relation, whose equivalence classes correspond to the nodes of *compact directed acyclic word graphs* (*CDAWGs*) [1].

Definition 2. *For any string* $x, y \in \Sigma^*$*, we denote* $x \equiv y$*, if and only if*

1. $x \notin \mathit{Substr}(w)$ *and* $y \notin \mathit{Substr}(w)$*, or*
2. $x, y \in \mathit{Substr}(w)$ *and* $\overleftrightarrow{x} = \overleftrightarrow{y}$*.*

The equivalence class of a string x *with respect to* $\equiv$ *is denoted by* $[x]_\equiv$*. For any* $x \in \mathit{Substr}(w)$*, the unique longest member* $\overleftrightarrow{x}$ *of* $[x]_\equiv$ *is called the* representative *of the equivalence class.*

Now we consider a *succinct representation* of each non-degenerate equivalence class under $\equiv$. For any $x \in \mathit{Substr}(w)$, let $\mathit{Minimal}([x]_\equiv)$ denote the set of minimal elements of $[x]_\equiv$, that is,

$$\mathit{Minimal}([x]_\equiv) = \{y \in [x]_\equiv \mid \ z \in \mathit{Substr}(y) \text{ and } z \in [x]_\equiv \text{ implies } z = y\}.$$

Namely, $\mathit{Minimal}([x]_\equiv)$ is the set of strings y in $[x]_\equiv$ such that there is no string $z \in \mathit{Substr}(y) - \{y\}$ with $z \equiv x$.

The following lemma shows that the strings in every non-degenerate equivalence class $[x]_\equiv$ can be represented by a pair of its representative $\overleftrightarrow{x}$ and $\mathit{Minimal}([x]_\equiv)$.

Lemma 1 ([3]). *For any x in $Substr(w)$, let $y_1, \ldots, y_k$ be the elements of $Minimal([x]_\equiv)$. Then,*

$$[x]_\equiv = Pincer(y_1, \overleftrightarrow{x}) \cup \cdots \cup Pincer(y_k, \overleftrightarrow{x}),$$

where $Pincer(y_i, \overleftrightarrow{x})$ is the set of strings z such that $z \in Substr(\overleftrightarrow{x})$ and $y_i \in Substr(z)$.

Now, a succinct representation of a non-degenerate equivalence class $[x]_\equiv$ is a pair of $\overleftrightarrow{x}$ and $Minimal([x]_\equiv)$, where $\overleftrightarrow{x}$ and all strings in $Minimal([x]_\equiv)$ are represented by their reference pairs w.r.t. w. We have the following lemma about the total space requirement for the succinct representations of all the equivalence classes under $\equiv$.

Lemma 2. *A list of succinct representations of all non-degenerate equivalence classes under $\equiv$ requires only $O(|w|)$ space.*

Let the *size* of non-degenerate equivalence class $[x]_\equiv$ be the number of substrings that belong to $[x]_\equiv$, that is, $|[x]_\equiv|$. Let $Freq(x)$ denote the occurrence frequency of x in w. If $x \equiv y$, then $Freq(x) = Freq(y)$. Therefore, we consider the frequency of an equivalence class $[x]_\equiv$ and denote this by $Freq([x]_\equiv)$.

2.3 Data Structures

We use the following data structures in our algorithms.

Definition 3 (Suffix Trees and Suffix Link Trees). *For any string w, the suffix tree of w, denoted $ST(w)$, is an edge-labeled tree structure (V, E) such that*

$$V = \{x \mid x = \overrightarrow{x}, x \in Substr(w)\}, \text{ and}$$
$$E = \{(x, \beta, x\beta) \mid x, x\beta \in V, \beta \in \Sigma^+,\ a = \beta[1],\ \overrightarrow{xa} = x\beta\},$$

where the second component β of each edge $(x, \beta, x\beta)$ in E is its label, and the suffix link tree of w, denoted $SLT(w)$, is a tree structure (V, E_ℓ) such that

$$E_\ell = \{(ax, x) \mid x, ax \in V, a \in \Sigma\}.$$

It is well known that $ST(w)$ with $SLT(w)$ can be computed in linear time and space [11,6].

The root node of $ST(w)$ and $SLT(w)$ is associated with $\varepsilon = \overrightarrow{\varepsilon}$. Since the end-marker \$ is unique in w, every nonempty suffix of strings in w corresponds to a leaf of $ST(w)$, and only such a leaf exists in $ST(w)$. Therefore, each leaf can be identified by the beginning position of the corresponding suffix of w. The values $Freq(x)$ for all nodes $x \in V$ of $ST(w)$ can be computed in linear time and space by a post-order traversal on $ST(w)$.

For any node $x\beta$ with incoming edge $(x, \beta, x\beta)$ of $ST(w)$, let $Paths(x\beta) = \{x\beta' \mid \beta' \in Prefix(\beta) - \{\varepsilon\}\}$. Note that $Paths(x\beta) = [x\beta]_{\equiv_L}$, and therefore $\overrightarrow{z} = x\beta$ for any $z \in Paths(x\beta)$. It is easy to see that $|Paths(x\beta)| = |x\beta| - |x| = |\beta|$.

For any node x of $ST(w)$ such that $x \neq \varepsilon$, let $Parent(x)$ denote the parent of x.

Definition 4 (Suffix Arrays). *The* suffix array *[7] SA of any string w is an array of length $|w|$ such that $SA[i] = j$, where $w[j:]$ is the i-th lexicographically smallest suffix of w.*

The suffix array of string w can be computed in linear time from $ST(w)$, by arranging the out-going edges of any node of $ST(w)$ in the lexicographically increasing order of the first symbols of the edge labels. This way all the leaves of $ST(w)$ are sorted in the lexicographically increasing order, and they correspond to SA of w. Linear-time direct construction of SA has also been extensively studied [12,13,14].

Definition 5 (Rank and LCP Arrays). *The* rank *and* lcp *arrays of any string w are arrays of length $|w|$ such that $rank[SA[i]] = i$, and $lcp[i]$ is the length of the longest common prefix of $w[SA[i-1]:]$ and $w[SA[i]:]$ for $2 \leq i \leq |w|$, and $lcp[1] = -1$.*

Given SA of string w, the *rank* and *lcp* arrays of w can be computed in linear time and space [8].

3 Computing Equivalence Classes Under ≡ Using Suffix Trees

In this section we present a suffix tree based algorithm to compute a succinct representation of each non-degenerate equivalence class, together with its size and frequency. This algorithm will be the basis of our algorithm of Section 4, which uses suffix arrays instead of trees.

The following lemma states how to check the equivalence relation $\equiv$ between two substrings using $ST(w)$.

Lemma 3. *For any $x, y \in Substr(w)$, $x \equiv y$ if and only if $Freq(\overrightarrow{x}) = Freq(\overrightarrow{y})$ and $\overrightarrow{x} \in Suffix(\overrightarrow{y})$ or vise versa.*

Proof. The case that $\overrightarrow{x} = \overrightarrow{y}$ is trivial. We consider the case that $\overrightarrow{x} \neq \overrightarrow{y}$. Assume w.l.o.g. that $|\overrightarrow{x}| < |\overrightarrow{y}|$.

Assume $x \equiv y$. Then we have $\overleftarrow{(\overrightarrow{x})} = \overleftrightarrow{x} = \overleftrightarrow{y} = \overleftarrow{(\overrightarrow{y})}$, which implies that $EndPos(\overrightarrow{x}) = EndPos(\overrightarrow{y})$. Thus we have $Freq(\overrightarrow{x}) = Freq(\overrightarrow{y})$. Since $|\overrightarrow{x}| < |\overrightarrow{y}|$, $\overrightarrow{x} \in Suffix(\overrightarrow{y})$.

Now assume $Freq(\overrightarrow{x}) = Freq(\overrightarrow{y})$ and $\overrightarrow{x} \in Suffix(\overrightarrow{y})$. Since $\overrightarrow{x} \in Suffix(\overrightarrow{y})$, we have $EndPos(\overrightarrow{x}) \supseteq EndPos(\overrightarrow{y})$. Moreover, since $Freq(\overrightarrow{x}) = Freq(\overrightarrow{y})$, we get $EndPos(\overrightarrow{x}) = EndPos(\overrightarrow{y})$. Hence $\overleftrightarrow{x} = \overleftarrow{(\overrightarrow{x})} = \overleftarrow{(\overrightarrow{y})} = \overleftrightarrow{y}$. □

Lemma 4. *For any node $x \in V$ of $ST(w)$ such that $\overleftrightarrow{x} = x$, let $\ell = \max\{i \mid Freq(x[i:]) = Freq(x)\}$. Then, $[x]_\equiv = \bigcup_{y \in [x]_{\equiv_R}} [y]_{\equiv_L} = \bigcup_{i=1}^{\ell} Paths(x[i:])$.*

Proof. By Lemma 3. □

For each node x of $ST(w)$, $|\mathit{Paths}(x)| = |\mathit{Parent}(x)| - |x|$ and it can be pre-computed by a post-order traversal on $ST(w)$. Thus, by the above lemma, the size of each non-degenerate equivalence class can be computed by a post-order traversal on $SLT(w)$.

In what follows, we show how to check whether or not a given node x in the suffix link tree traversal is the representative of the equivalence class under $\equiv$, namely, whether or not $x = \overleftrightarrow{x}$.

Lemma 5. *For any node $x \in V$ of $ST(w)$, $x = \overleftrightarrow{x}$ if and only if $\mathit{Freq}(ax) < \mathit{Freq}(x)$ for any $a \in \Sigma$ such that $ax \in V$.*

Proof. By Lemma 3. $\square$

The following two lemmas follow from Lemma 5.

Lemma 6. *For any leaf node $x \in V$ of $SLT(w)$, $x = \overleftrightarrow{x}$.*

Proof. Since x is also a node of $ST(w)$, $x = \overrightarrow{x}$. We show that for any symbol $a \in \Sigma$, $\mathit{Freq}(ax) < \mathit{Freq}(x)$, and therefore, $x = \overleftarrow{x}$. Since x is a leaf of $SLT(w)$, we have $ax \notin V$ for any $a \in \Sigma$, for which there are the two following cases:

1. $ax \notin \mathit{Substr}(w)$. Then, $\mathit{Freq}(ax) = 0$ while $\mathit{Freq}(x) > 0$.
2. $ax \in \mathit{Substr}(w)$. Consider $\beta \in \Sigma^+$ such that $\overrightarrow{ax} = ax\beta \in V$. Then, we have that $\mathit{Freq}(ax) = \mathit{Freq}(ax\beta) \leq \mathit{Freq}(x\beta) < \mathit{Freq}(x)$.

In both cases we have $\mathit{Freq}(ax) < \mathit{Freq}(x)$, and hence $x = \overleftrightarrow{x}$ from Lemma 5. $\square$

Lemma 7. *For any internal node $x \in V$ of $SLT(w)$ and for any $a \in \Sigma$ with $ax \in V$, $x = \overleftrightarrow{x}$ if and only if $\mathit{Freq}(ax) \neq \mathit{Freq}(x)$.*

Proof. ($\Rightarrow$) Since $x = \overleftrightarrow{x}$, we have $\mathit{Freq}(bx) \neq \mathit{Freq}(x)$ for any $b \in \Sigma$. ($\Leftarrow$) Since $\mathit{Freq}(x) \geq \sum_{b \in \Sigma} \mathit{Freq}(bx)$ and $\mathit{Freq}(x) > \mathit{Freq}(ax) > 0$, we have $\mathit{Freq}(x) > \mathit{Freq}(bx)$ for any $b \in \Sigma$. $\square$

We have the following lemma concerning the minimal members of the non-degenerate equivalence classes.

Lemma 8. *For any nodes $x, ax \in V$ of $ST(w)$ with $a \in \Sigma$, and let yb be the shortest member in $\mathit{Paths}(ax)$ where $y \in \Sigma^*$ and $b \in \Sigma$. Then, $yb \in \mathit{Minimal}([ax]_\equiv)$ if and only if (1) $|\mathit{Paths}(ax)| > |\mathit{Paths}(x)|$ or (2) $\mathit{Freq}(ax) < \mathit{Freq}(x)$.*

Proof. It is clear when $y = \varepsilon$. We consider the case where $y \neq \varepsilon$. Let $y = ay'$ for $y \in \Sigma^*$.

($\Rightarrow$) Assume $ay'b \in \mathit{Minimal}([ax]_\equiv)$, which implies $ay' \notin [ax]_\equiv$ and $y'b \notin [ax]_\equiv$. First, consider the case where $x \notin [ax]_\equiv$. Then clearly (2) holds. For the case where $x \in [ax]_\equiv$, $\overrightarrow{y'b} \neq x$ since $y'b \notin [ax]_\equiv = [x]_\equiv$, and there exists a node corresponding to $\overrightarrow{y'b}$ on the path from y' to x. Therefore (1) holds.

($\Leftarrow$) Since $ay'b$ is the shortest member in $\mathit{Paths}(ax)$, $ay' \notin [ax]_\equiv$. It remains to show $y'b \notin [ax]_\equiv$. If we assume (2), $\mathit{Freq}(ax) < \mathit{Freq}(x) \leq \mathit{Freq}(y'b)$ since $y'b$

Algorithm 1. Algorithm for computing a succinct representation, the size and frequency of each non-degenerate equivalence class using suffix trees

```
   Input: $\mathsf{ST(w)}$,$\mathsf{SLT(w)}$ : suffix tree and suffix link tree of $\mathsf{w}$
   Output: a succinct representation, the size and frequency of each
           non-degenerate equivalence class
1  foreach node $\mathsf{v} \in \mathsf{V}$ in post-order of $\mathsf{ST(w)}$ do
2      calculate and store the values $Freq(\mathsf{v})$ and $|Paths(\mathsf{v})|$;
3  $\mathsf{size} := 0$; $\mathsf{freq} := 0$;
4  foreach node $\mathsf{v} \in \mathsf{V}$ in post-order of $\mathsf{SLT(w)}$ do
5      if $\mathsf{v}$ is a leaf of $\mathsf{SLT(w)}$ or $\mathsf{freq} \neq Freq(\mathsf{v})$ then
6          if $\mathsf{size} \neq 0$ then
7              report $\mathsf{size}$ as the size of $[\mathsf{rep\_v}]_{\equiv}$;
8              $\mathsf{minimal} := \mathsf{minimal} \cup \{\langle i,j \rangle\}$ s.t. $w[i:j]$ is the shortest string in
               $Paths(\mathsf{old\_v})$;
9              report $(\langle i,j \rangle, \mathsf{minimal})$ as a succinct representation of $[\mathsf{rep\_v}]_{\equiv}$,
               where $w[i:j] = \mathsf{rep\_v}$;
10             $\mathsf{minimal} := \emptyset$;
11         $\mathsf{freq} := Freq(\mathsf{v})$; report $\mathsf{freq}$ as the frequency of $[\mathsf{rep\_v}]_{\equiv}$;
12         $\mathsf{size} := |Paths(\mathsf{v})|$; $\mathsf{len} := |Paths(\mathsf{v})|$; $\mathsf{old\_v} := \mathsf{v}$; $\mathsf{rep\_v} := \mathsf{v}$;
13     else
14         if $\mathsf{len} > |Paths(\mathsf{v})|$ then
15             $\mathsf{minimal} := \mathsf{minimal} \cup \{\langle i,j \rangle\}$ s.t. $w[i:j]$ is the shortest string in
               $Paths(\mathsf{old\_v})$;
16         $\mathsf{size} := \mathsf{size} + |Paths(\mathsf{v})|$; $\mathsf{len} := |Paths(\mathsf{v})|$; $\mathsf{old\_v} := \mathsf{v}$;
17     end
18 end
```

is a prefix of x. Therefore, we have $y'b \notin [ax]_{\equiv}$ because $Freq(y'b) \neq Freq(ax)$. Next, assume (1) when (2) does not hold, that is, $Freq(y'b) = Freq(ax)$. Then, $\overrightarrow{y'b} \neq x$ or else, $|Paths(ax)| = |Paths(x)|$. Therefore, $y'b \notin [x]_{\equiv} = [ax]_{\equiv}$. □

A pseudo-code of the algorithm to compute a succinct representation of each non-degenerate equivalence class together with its size and frequency is shown as Algorithm 1. The above arguments lead to the following theorem.

Theorem 1. *Given $ST(w)$ and $SLT(w)$, Algorithm 1 computes succinct representations of all non-degenerate equivalence classes under $\equiv$, together with their sizes and frequencies in linear time.*

4 Computing Equivalence Classes Under $\equiv$ Using Suffix Array

In this section, we develop an algorithm that simulates Algorithm 1 using suffix arrays. Our algorithm is based on the algorithm by Kasai et al. [8] which simulates a post-order traversal on suffix trees with *SA*, *rank* and *lcp* arrays. A key feature of our algorithm is that it does not require any extra data structure other than the suffix, rank and lcp arrays, making it quite space economical.

For any string $x \in \mathit{Substr}(w)$, let

$$\mathit{Lbeg}(x) = \mathit{SA}[\min\{\mathit{rank}[i] \mid i \in \mathit{BegPos}(x)\}] \text{ and}$$
$$\mathit{Rbeg}(x) = \mathit{SA}[\max\{\mathit{rank}[i] \mid i \in \mathit{BegPos}(x)\}].$$

Recall that Algorithm 1 traverses $\mathit{SLT}(w)$. Our suffix array based algorithm simulates traversal on $\mathit{ST}(w)$, and when reaching any node x such that $x = \overleftrightarrow{x}$, it simulates suffix link tree traversal until reaching node $y \in \mathit{Suffix}(x)$ such that $y \not\equiv x$.

The next lemma states that for any node x of $\mathit{ST}(w)$, $\mathit{Freq}(x)$ is constant time computable using *rank* array.

Lemma 9. *For any node $x \in V$ of $\mathit{ST}(w)$,*

$$\mathit{Freq}(x) = \mathit{rank}[\mathit{Rbeg}(x)] - \mathit{rank}[\mathit{Lbeg}(x)] + 1.$$

When reaching any node x such that $x = \overleftrightarrow{x}$ in the post-order traversal on $\mathit{ST}(w)$, we compute a succinct representation of $[x]_\equiv$ due to Lemma 3. Examination of whether $x = \overleftrightarrow{x}$ can be done in constant time according to the following lemma.

Lemma 10. *For any node $x \in V$ of $\mathit{ST}(w)$, let $l = \mathit{Lbeg}(x)$ and $r = \mathit{Rbeg}(x)$. We have $x = \overleftrightarrow{x}$ if and only if at least one of the following holds: (1) $l-1 = 0$ or $r-1 = 0$, (2) $w[l-1] \neq w[r-1]$, or (3) $\mathit{rank}[r]-\mathit{rank}[l] \neq \mathit{rank}[r-1]-\mathit{rank}[l-1]$.*

Proof. ($\Rightarrow$) First, let us assume $x = \overleftrightarrow{x}$. If (1) and (2) do not hold, that is, $l-1 \neq 0$, $r-1 \neq 0$ and $w[l-1] = w[r-1] = a$, then $\mathit{Freq}(x) > \mathit{Freq}(ax)$ due to Lemma 3. This implies $\mathit{rank}[r] - \mathit{rank}[l] > \mathit{rank}[\mathit{Rbeg}(ax)] - \mathit{rank}[\mathit{Lbeg}(ax)] \geq \mathit{rank}[r-1] - \mathit{rank}[l-1]$ from Lemma 9, showing (3).

($\Leftarrow$) To show the reverse, we have only to show $\overleftarrow{x}$ since x is a node of $\mathit{ST}(w)$, and therefore $x = \overrightarrow{x}$. First, we show (1) $\Rightarrow x = \overleftrightarrow{x}$. If $l = 1$, then $|x| \in \mathit{EndPos}(x)$ while $|x| \notin \mathit{EndPos}(ax)$ for any $a \in \Sigma$, implying $x = \overleftarrow{x}$. The same applies for $r = 1$.

Next, we show (2) $\Rightarrow x = \overleftrightarrow{x}$ when (1) does not hold, that is, $l-1 \neq 0$ and $r-1 \neq 0$. Since $w[l-1] \neq w[r-1]$, we have that $l-1+|x| \in \mathit{EndPos}(x)$ while $l-1+|x| \notin \mathit{EndPos}(w[r-1]x)$ and $r-1+|x| \in \mathit{EndPos}(w[r-1]x)$. Therefore, $\mathit{Freq}(x) > \mathit{Freq}(w[r-1]x) > 0$, and since $\mathit{Freq}(x) \geq \sum_{a\in\Sigma} \mathit{Freq}(ax)$, we have $\mathit{Freq}(ax) < \mathit{Freq}(x)$ for all $a \in \Sigma$, thus implying $x = \overleftarrow{x}$.

Finally, we show (3) $\Rightarrow x = \overleftrightarrow{x}$ when (1) and (2) do not hold, that is, $l-1 \neq 0$, $r-1 \neq 0$ and $w[l-1] = w[r-1] = a$. (3) implies that $\mathit{rank}[r] - \mathit{rank}[l] > \mathit{rank}[\mathit{Rbeg}(ax)] - \mathit{rank}[\mathit{Lbeg}(ax)] \geq \mathit{rank}[r-1] - \mathit{rank}[l-1]$, and from Lemma 9 we have that $\mathit{Freq}(x) > \mathit{Freq}(ax) > 0$. Therefore $x \not\equiv ax$ from Lemma 3, implying $x = \overleftarrow{x}$.

Therefore, we have $x = \overleftrightarrow{x}$ if we assume at least one of (1)–(3). □

Now we consider to check whether or not $ax \equiv x$ for any nodes ax, x of $\mathit{ST}(w)$, where $a \in \Sigma$ and $x \in \Sigma^*$. By definition, it is clear that $\mathit{Lbeg}(ax)+1 \in \mathit{BegPos}(x)$

and $Rbeg(ax)+1 \in BegPos(x)$. However, note that $Lbeg(ax)+1 = Lbeg(x)$ does *not* always hold (same for $Rbeg$).

To check if $ax \equiv x$, we need to know whether or not $Lbeg(ax)+1 = Lbeg(x)$, and it can be done by the following lemma:

Lemma 11. *For any nodes $ax, x \in V$ of $ST(w)$ such that $a \in \Sigma$ and $x \in \Sigma^*$, let $l = Lbeg(ax)$. Then, $lcp[rank[l+1]] < |ax|-1$ if and only if $Lbeg(x) = l+1$.*

Proof. If $Lbeg(x) = l+1$, then clearly $lcp[rank[l+1]] < |x| = |ax|-1$.

Now, assume on the contrary that $Lbeg(x) \neq l+1$. Then, $rank[Lbeg(x)] < rank[l+1]$, and since $w[Lbeg(x):]$ and $w[l+1:]$ share x as a prefix, we have $lcp[rank[l+1]] \geq |x| = |ax|-1$ which is a contradiction. □

The following lemma can be shown in a similar way to the above lemma:

Lemma 12. *For any nodes $ax, x \in V$ of $ST(w)$ such that $a \in \Sigma$ and $x \in \Sigma^*$, let $r = Rbeg(ax)$. Then, $lcp[rank[r+1]+1] < |ax|-1$, if and only if $Rbeg(x) = r+1$.*

Now we have the following lemma on which our examination of equivalence relation is based.

Lemma 13. *For any nodes $ax, x \in V$ of $ST(w)$ such that $a \in \Sigma$ and $x \in \Sigma^*$, we have $ax \equiv x$ if and only if*

(1) $|ax|-1 > lcp[rank[l+1]]$,
(2) $|ax|-1 > lcp[rank[r+1]+1]$, and
(3) $rank[r] - rank[l] = rank[r+1] - rank[l+1]$,

where $l = Lbeg(ax)$ and $r = Rbeg(ax)$.

Proof. ($\Rightarrow$) Assume $ax \equiv x$. Then, $Freq(ax) = Freq(x)$ and thus we have $rank[Rbeg(ax)] - rank[Lbeg(ax)] = rank[Rbeg(x)] - rank[Lbeg(x)]$ from Lemma 9. From $Freq(ax) = Freq(x)$, we have $Rbeg(x) = Rbeg(ax)+1 = r+1$ and $Lbeg(x) = Lbeg(ax)+1 = l+1$. By Lemma 11 and Lemma 12 we get $|ax|-1 > lcp[rank[l+1]]$ and $|ax|-1 > lcp[rank[r+1]+1]$.

($\Leftarrow$) From Lemma 11, if $|ax|-1 > lcp[rank[l+1]]$, then $Lbeg(x) = l+1$. From Lemma 12, if $|ax|-1 > lcp[rank[r+1]+1]$, then $Rbeg(x) = r+1$. Therefore, if $rank[r] - rank[l] = rank[r+1] - rank[l+1]$, then $Freq(ax) = Freq(x)$ by Lemma 9. Consequently, we get $ax \equiv x$ from Lemma 3. □

Next, we consider how to compute $|Paths(x)| = |x| - |Parent(x)|$. When $x = \overleftrightarrow{x}$, we know $|x|$ and $|Parent(x)|$ which are computed in post-order traversal on $ST(w)$ simulated by the algorithm of [8]. When $x \neq \overleftrightarrow{x}$, namely, when x has been reached in suffix link traversal simulation, we have that $|x| = |ax|-1$ where ax is the node reached immediately before x in the suffix link tree traversal simulation. We have the following lemma for computation of $|Parent(x)|$.

Lemma 14. *For any node $x \in V$ of $ST(w)$, let $l = Lbeg(x)$ and $r = Rbeg(x)$. Then, $|Parent(x)| = \max\{lcp[rank[l]], lcp[rank[r]+1]\}$.*

Algorithm 2. Algorithm for computing a succinct representation, the size and frequency of each non-degenerate equivalence class using suffix, lcp and rank arrays

```
Input: $SA[1 : |w|]$, $lcp[1 : |w|]$, $rank[1 : |w|]$ : suffix, lcp and rank arrays of string w
Output: a succinct representation, the size and frequency of each
        non-degenerate equivalence class
1  Stack initialization (Left, Height) := $(-1, -1)$;
2  for $i = 1, \ldots, n$ do
3      Lnew := $i - 1$; Hnew := $lcp[i]$; Left := Stack.Left; Height := Stack.Height;
4      while Height > Hnew do
5          Pop Stack;
6          if Stack.Height > Hnew then parent := Stack.Height;
7          else parent = Hnew;
8          L := Left; R := $i - 1$; freq := R − L + 1; rlen := Height;
9          if $(SA[L] \neq 1) \& (SA[R] \neq 1)$ then
10             BL := $rank[SA[L] - 1]$; BR := $rank[SA[R] - 1]$;
11         if (BR−BL+1 ≠ freq) or (w[BL] ≠ w[BR]) or $(SA[L] = 1)$ or $(SA[R] = 1)$ then
12             Let x = w[$SA[L]$ : $SA[L]$ + rlen − 1];
13             report freq as the frequency of $[x]_{\equiv}$;
14             size := rlen − parent; mlen := rlen − parent; len := rlen; minimal := $\emptyset$;
15             FL := $rank[SA[L] + 1]$; FR := $rank[SA[R] + 1]$; BL := L; BR := R;
16             while (len−1 > $lcp$[FL]) & (len−1 > $lcp$[FR+1]) & (FR−FL+1 = freq) do
17                 if $lcp$[FL] ≥ $lcp$[FR + 1] then parent := $lcp$[FL];
18                 else parent := $lcp$[FR + 1];
19                 len := len − 1; size := size + len − parent;
20                 if mlen > len − parent then
21                     minimal := minimal ∪ {⟨$SA$[BL], $SA$[BL] + parent⟩};
22                 BL := FL; BR := FR;
23                 if ($SA$[FL] + 1 ≥ |w|) or ($SA$[FR] + 1 ≥ |w|) then break;
24                 FL := $rank[SA[FL]+1]$; FR := $rank[SA[FR]+1]$; mlen := len−parent;
25             report size as the size of $[x]_{\equiv}$;
26             minimal := minimal ∪ {⟨$SA$[BL], $SA$[BL] + parent⟩};
27             report(⟨$SA$[L], $SA$[L] + rlen − 1⟩, minimal) as a succinct
               representation of $[x]_{\equiv}$;
28         Lnew := Left; Left := Stack.Left; Height := Stack.Height;
29     if Height < Hnew then Push(Lnew, Hnew) to Stack;
30     Push($i$, |w| − $SA[i]$) to Stack;
31 end
```

Proof. For all $1 \leq i < rank[l]$, the length of the longest common prefix of $w[SA[i]:]$ and x is at most $lcp[rank[l]]$. Similarly for $rank[r] < j \leq |w|$, the length of the longest common prefix of $w[SA[j]:]$ and x is at most $lcp[rank[r]+1]$. Also, for all $rank[l] \leq k \leq rank[r]$, the longest common prefix of $w[SA[k]:]$ and x is $|x|$, and therefore $lcp[k] \geq |x|$ for all $rank[l] < k \leq rank[r]$. This implies that $|Parent(x)|$ is equal to either $lcp[rank[l]]$ or $lcp[rank[r]+1]$ and hence the lemma follows. □

A pseudo-code of the algorithm is shown in Algorithm 2. The **for** and **while** loops on line 2 and line 2 simulate a post-order traversal on $ST(w)$ using SA, *rank* and *lcp* arrays, and it takes linear time due to [8]. Checking whether or not $x = \overleftrightarrow{x}$ for any node x reached in the post-order traversal on $ST(w)$, is done in line 2 due to Lemma 10. Thus, we go into the **while** loop on line 2 only when $x = \overleftrightarrow{x}$, and this **while** loop continues until reaching $y \in \mathit{Suffix}(x)$ such that $y \not\equiv x$ due to Lemma 13. It is clear that all calculations in the **while** loop can be done in constant time.

Theorem 2. *Given SA, rank and lcp arrays of string w, Algorithm 2 computes succinct representations of all non-degenerate equivalence classes under $\equiv$, together with their sizes and frequencies in linear time.*

5 Experimental Results

We performed preliminary experiments on corpora [15,16], to compare practical time and space requirements of suffix tree, CDAWG, and suffix array based approaches to compute a succinct representation of for each non-degenerate equivalence class under $\equiv$, together with its size and frequency.

We constructed suffix trees using Ukkonen's algorithm [6], and ran Algorithm 1. CDAWGs were constructed using the CDAWG construction algorithm of [4]. We computed suffix arrays using the qsufsort program by [17]. All the experiments were conducted on a RedHat Linux desktop computer with a 2.8 GHz Pentium 4 processor and 1 GB of memory.

Table 1 shows the running time and memory usage of the algorithms for each data structure. The enumeration column shows the time efficiency of the algorithms computing succinct representations of all equivalence classes together with their sizes and frequencies. For all the corpora the suffix array approach

Table 1. The comparison of the computation time and memory space for suffix trees, CDAWGs and suffix arrays

corpora data name	data size (Mbytes)	data structure	Time (seconds)			memory (Mbytes)
			construction	enumeration	total	
cantrby/plrabn12	0.47	Suffix Tree	0.95	0.21	1.16	21.446
		CDAWG	0.97	0.18	1.15	9.278
		Suffix Array	**0.43**	**0.14**	**0.57**	**5.392**
ProteinCorpus/sc	2.8	Suffix Tree	12.08	1.43	13.51	121.877
		CDAWG	12.76	1.12	13.88	69.648
		Suffix Array	**3.08**	**0.63**	**3.71**	**33.192**
large/bible.txt	3.9	Suffix Tree	7.33	2.23	9.56	191.869
		CDAWG	6.68	1.62	8.30	56.255
		Suffix Array	**4.71**	**1.50**	**6.21**	**46.319**
large/E.coli	4.5	Suffix Tree	8.17	2.91	11.08	232.467
		CDAWG	8.58	2.31	10.89	139.802
		Suffix Array	**5.95**	**1.46**	**7.41**	**53.086**

was the fastest. In addition, the suffix array algorithm uses the least memory space for all the corpora.

References

1. Blumer, A., Blumer, J., Haussler, D., Mcconnell, R., Ehrenfeucht, A.: Complete inverted files for efficient text retrieval and analysis. J. ACM 34(3), 578–595 (1987)
2. Narisawa, K., Bannai, H., Hatano, K., Takeda, M.: Unsupervised spam detection based on string alienness measures. Technical report, Department of Informatics, Kyushu University (2007)
3. Takeda, M., Matsumoto, T., Fukuda, T., Nanri, I.: Discovering characteristic expressions in literary works. Theoretical Computer Science 292(2), 525–546 (2003)
4. Inenaga, S., Hoshinoa, H., Shinohara, A., Takeda, M., Arikawa, S., Mauri, G., Pavesi, G.: On-line construction of compact directed acyclic word graphs. Discrete Applied Mathematics 146(2), 156–179 (2005)
5. Weiner, P.: Linear pattern matching algorithms. In: Proc. 14th IEEE Annual Symp. on Switching and Automata Theory, pp. 1–11 (1973)
6. Ukkonen, E.: On-line construction of suffix trees. Algorithmica 14(3), 249–260 (1995)
7. Manber, U., Myers, G.: Suffix arrays: a new method for on-line string searches. SIAM J. Computing 22(5), 935–948 (1993)
8. Kasai, T., Lee, G., Arimura, H., Arikawa, S., Park, K.: Linear-time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications. In: Amir, A., Landau, G.M. (eds.) CPM 2001. LNCS, vol. 2089, pp. 181–192. Springer, Heidelberg (2001)
9. Abouelhoda, M.I., Kurtz, S., Ohlebusch, E.: Replacing suffix trees with enhanced suffix arrays. Journal of Discrete Algorithms 2(1), 53–86 (2004)
10. Blumer, A., Blumer, J., Haussler, D., Ehrenfeucht, A., Chen, M.T., Seiferas, J.: The smallest automaton recognizing the subwords of a text. Theoretical Computer Science 40, 31–55 (1985)
11. McCreight, E.M.: A space-economical suffix tree construction algorithm. J. ACM 23(2), 262–272 (1976)
12. Kärkkäinen, J., Sanders, P.: Simple linear work suffix array construction. In: Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.) ICALP 2003. LNCS, vol. 2719, pp. 943–955. Springer, Heidelberg (2003)
13. Kim, D.K., Sim, J.S., Park, H., Park, K.: Linear-time construction of suffix arrays. In: Baeza-Yates, R.A., Chávez, E., Crochemore, M. (eds.) CPM 2003. LNCS, vol. 2676, pp. 186–199. Springer, Heidelberg (2003)
14. Ko, P., Aluru, S.: Space efficient linear time construction of suffix arrays. In: Baeza-Yates, R.A., Chávez, E., Crochemore, M. (eds.) CPM 2003. LNCS, vol. 2676, pp. 200–210. Springer, Heidelberg (2003)
15. Arnold, R., Bell, T.: A corpus for the evaluation of lossless compression algorithms. In: Proc. DCC '97, pp. 201–210 (1997) `http://corpus.canterbury.ac.nz/`
16. Nevill-Manning, C., Witten, I.: Protein is incompressible. In: Proc. DCC '99, pp. 257–266 (1999), `http://www.data-compression.info/Corpora/ProteinCorpus/index.htm`
17. Larsson, N.J., Sadakane, K.: Faster suffix sorting. Technical Report LU-CS-TR:99-214, LUNDFD6/(NFCS-3140)/1–20/(1999) Department of Computer Science, Lund University, Sweden (1999) `http://www.larsson.dogma.net/qsufsort.c`

A Simple Construction of Two-Dimensional Suffix Trees in Linear Time
(Extended Abstract)

Dong Kyue Kim[1,⋆], Joong Chae Na[2], Jeong Seop Sim[3,⋆⋆], and Kunsoo Park[4,⋆⋆⋆]

[1] Division of Electronics and Computer Engineering, Hanyang University, Korea
`dqkim@hanyang.ac.kr`
[2] Department of Advanced Technology Fusion, Konkuk University, Korea
`jcna@theory.snu.ac.kr`
[3] School of Computer Science and Engineering, Inha University, Korea
`jssim@inha.ac.kr`
[4] School of Computer Science and Engineering, Seoul National University, Korea
`kpark@theory.snu.ac.kr`

Abstract. The two-dimensional suffix tree of a matrix A is a compacted trie that represents all square submatrices of A. There exists a linear-time construction of two-dimensional suffix trees using the *odd-even* scheme. This algorithm has the drawback that the merging step is quite complicated. In this paper, we propose a new and simple algorithm to construct two-dimensional suffix trees in linear time by applying the *skew* scheme to square matrices. To do this, we present a simple algorithm to merge two Isuffix trees in linear time.

1 Introduction

The suffix tree T_S of a string S is a compacted trie that represents all substrings of S. It has been a fundamental data structure not only for Computer Science, but also for Engineering and Bioinformatics applications [2,5,14]. The suffix tree was designed as a space-efficient alternative [24] to Weiner's position tree [29]. When the alphabet Σ of the given string S is of constant size, linear-time algorithms for constructing the suffix tree have been known for quite a while [3,24,28]. For integer alphabets, Farach [6] gave the first linear-time construction algorithm using a divide-and-conquer approach.

In two dimensions, Gonnet [13] first introduced a notion of suffix tree for a matrix, called the *PAT-tree*. Giancarlo [9] proposed the *Lsuffix tree*, compactly

⋆ Supported by the Korea Research Foundation Grant funded by the Korean Government(MOEHRD) (KRF-2006-311-D00762).
⋆⋆ Supported by Inha Research Grant 2006.
⋆⋆⋆ Corresponding author. Supported by FPR05A2-341 of 21C Frontier Functional Proteomics Project from Korean Ministry of Science & Technology.

storing all *square submatrices* of an $n \times n$ matrix A, together with an $O(n^2 \log n)$-time construction algorithm that uses $O(n^2)$ space. Giancarlo and Grossi [10,11] also introduced the general framework of two-dimensional suffix tree families and gave an expected linear-time construction algorithm. Kim and Park [20] proposed the first linear-time construction algorithm of two-dimensional suffix trees, called *Isuffix trees*, for polynomially bounded alphabet size using Farach's paradigm [6]. Cole and Hariharan [4] gave a randomized linear-time construction algorithm. Giancarlo and Guaiana [12] presented an $O(n^2 \log^2 n)$-time algorithm for *on-line* construction of the Lsuffix Tree. Na, Giancarlo and Park [25] improved their algorithm and gave an $O(n^2 \log n)$-time on-line construction algorithm. Two-dimensional suffix arrays (extension of Manber and Myers's suffix arrays [23]) are also constructed in $O(n^2 \log n)$ time [9,19].

A *divide-and-conquer* approach has been widely used to develop linear-time algorithms for constructing full-text index structures, i.e., suffix trees or suffix arrays. It consists of the following steps:

1. Partition the suffixes of S into two groups X and Y, and generate a string S' such that the suffixes in S' correspond to the suffixes in X. This step requires encoding several symbols in S into a new symbol in S'.
2. Construct the suffix tree of S' recursively.
3. Construct the suffix tree for X from the suffix tree of S' by decoding the encoded symbols.
4. Construct the suffix tree for Y using the suffix tree for X.
5. Merge the two suffix trees for X and Y to get the suffix tree of S.

In constructing suffix trees, this approach was applied to many sequential and parallel algorithms [6,8,16,26]. Farach [6] divided the suffixes of S into odd suffixes (group X) and even suffixes (group Y), called the *odd-even* scheme, to get the first linear-time algorithm for integer alphabets. Recently, almost at the same time, several algorithms [21,22,18,17] have been independently developed to directly construct suffix arrays in linear time. They are based on this divide-and-conquer approach. Kim et al. [21] and Hon et al. [17] followed the odd-even scheme. Kärkkäinen et al. [18] used the *skew* scheme, i.e., the suffixes of S are divided into suffixes beginning at positions $i \bmod 3 \neq 0$ (group X) and suffixes beginning at positions $i \bmod 3 = 0$ (group Y). Most of steps in the odd-even scheme are simple, but the merging step of this scheme is quite complicated. In the skew scheme, however, the merging step is simple and elegant, which makes the divide-and-conquer approach practical.

The linear-time algorithm for constructing two-dimensional suffix trees in [20] extended the odd-even scheme to an $n \times n (= N)$ square matrix. It partitions the Isuffixes of the matrix into four sets of size $\frac{1}{4}N$ each, and performs $\frac{3}{4}$-recursion, i.e., three sets of Isuffixes are regarded as group X and the remaining set as group Y. Since this construction used the odd-even scheme, the merging step was performed three times for each recursion.

In this paper, we present a new and simple algorithm for constructing two-dimensional suffix trees in linear time by applying the skew scheme to square matrices. Our contribution is twofold.

- We describe how to construct the Isuffix tree of group X instead of constructing three Isuffix trees of three sets of group X. Thus, the merging step is performed only once for each recursion.
- We present a simple algorithm to merge two Isuffix trees of group X and group Y in linear time. We also describe how to compare two Isuffixes from different groups in constant time.

This paper is organized as follows. In Section 2, we give some definitions and notations. We present outlines of recursive steps of the proposed algorithm in Section 3. In Section 4, we give detailed description of merging steps. Additionally we describe the decoding algorithm in the appendix, whose details were not described in [20].

2 Preliminaries

In this section we give some preliminaries including a linearization method of square matrices and the definition of two-dimensional suffix trees (Isuffix trees) introduced in [20].

2.1 Linear Representation of Square Matrices

Given an $n \times m$ ($n \leq m$) matrix A, we denote by $A[i : k, j : l]$ the submatrix of A with corners (i, j), (k, j), (i, l), and (k, l). When $i = k$ or $j = l$, we omit one of the repeated indices. An entry of matrix A is an integer in the range $[0, m^c]$ for some constant c. The integers in A can be mapped into integers in the range $[1, nm]$ by linear-time sorting. Hence we assume that $\Sigma = \{1, 2, \ldots, nm\}$.

Let $I\Sigma = \bigcup_{i=1}^{\infty} \Sigma^i$. We refer to the strings of $I\Sigma$ as *Icharacters* and we consider each of them as an *atomic* item. Two Icharacters are *equal* if and only if they are equal as strings over Σ. Given two Icharacters Ia and Ib of equal length, $Ia \prec Ib$ if and only if Ia as a string is lexicographically smaller than Ib as a string.

We describe a linearization method for a square matrix $C[1 : n, 1 : n]$ [1,9,19]. When we cut a matrix along the main diagonal, it is divided into an upper right half and a lower left half. (See Fig. 1 (b).) Let $a(i) = C[i + 1, 1 : i]$ and $b(i) = C[1 : i + 1, i + 1]$ for $1 \leq i < n$, i.e., $a(i)$ is a row of the lower left half and $b(i)$ is a column of the upper right half. Then $a(i)$'s and $b(i)$'s can be regarded as Icharacters. We call $a(i)$ a *row-type* Icharacter and $b(i)$ a *column-type* Icharacter. We consider the initial Icharacter $C[1, 1]$ as both row-type and column-type.

The linearized string IC of square matrix $C[1 : n, 1 : n]$, called the *Istring* of matrix C, is the concatenation of Icharacters $IC[1], \ldots, IC[2n - 1]$ that are defined as follows: (i) $IC[1] = C[1, 1]$; (ii) $IC[2i] = a(i)$, $1 \leq i < n$; (iii) $IC[2i + 1] = b(i)$, $1 \leq i < n$. (See Fig. 1 (b).) The *Ilength* of IC is the number of Icharacters in IC, denoted by $|IC|$. Note that $|IC| = 2n - 1$. Let $IC[j..k]$ denote the *Isubstring* of IC that is the concatenation of Icharacters $IC[j], \ldots, IC[k]$. The kth *Iprefix* of an Istring IC is the Isubstring $IC[1..k]$.

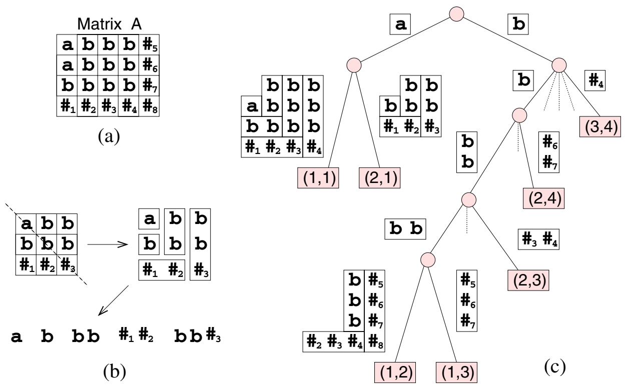


Fig. 1. (a) A matrix A, (b) the suffix A_{21} and its Istring, and (c) the Isuffix tree of A

2.2 Isuffix Trees

For $1 \leq i \leq n$ and $1 \leq j \leq m$, the *suffix* A_{ij} of matrix A is the largest square submatrix of A that starts at position (i, j) in A. That is, $A_{ij} = A[i : i + k, j : j + k]$, where $k = min(n - i, m - j)$. We assume that the entries of the last row and the last column in A are special symbols #'s, which make all suffixes of A distinct. The *Isuffix* IA_{ij} of A is the Istring of A_{ij}. The position (i, j) is called the *index* of Isuffix IA_{ij}.

The *Isuffix tree* $IST(A)$ of matrix A is a compacted trie that represents all Isuffixes of A. Each edge (u, v) is labeled with $label(u, v)$ that is a nonempty Isubstring $label(u, v) = IA_{ij}[\ell_1..\ell_2]$ of an Isuffix of A. To maintain the Isuffix tree in linear space, $O(nm)$, we store a 4-tuple (i, j, ℓ_1, ℓ_2) for $label(u, v)$. Each internal node v has at least two children. No two sibling edges have the same first Icharacters in their labels. For each node v, let $L(v)$ denote the Istring obtained by concatenating the labels on the path from the root to v. The leaf corresponding to Isuffix IA_{ij} will be denoted by l_{ij}. The *level* of a node v in an Isuffix tree is $|L(v)|$. See Fig. 1 for an example of an Isuffix tree.

The *least common ancestor* of two nodes v and w in a tree is denoted by $\texttt{lca}(v, w)$. By the results of [15,27] the computation of lca of two nodes can be done in constant time after linear-time preprocessing on a tree. The length of the *longest common prefix* of two strings α and β is denoted by $\texttt{lcp}(\alpha, \beta)$. Similarly, the Ilength of the *longest common Iprefix* of two Istrings $I\alpha$ and $I\beta$ is denoted by $\texttt{Ilcp}(I\alpha, I\beta)$. For all nodes u, v in Isuffix tree $IST(A)$, the following property is satisfied between Ilcp and lca: $\texttt{Ilcp}(L(u), L(v)) = |L(\texttt{lca}(u, v))|$.

3 Construction of Two-Dimensional Suffix Trees

We first give some definitions related to partial Isuffix trees and encoded matrices, which are used for recursion.

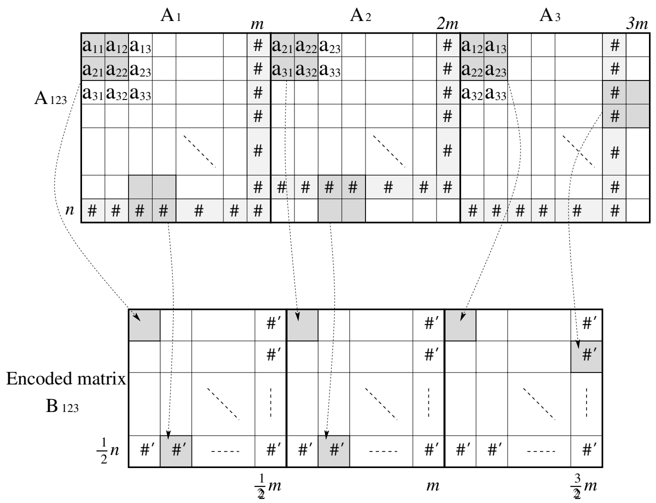


Fig. 2. A matrix A_{123} and its encoded matrix B_{123}

Given an $n \times m$ matrix A, we divide all Isuffixes IA_{ij} of matrix A into four types.

– IA_{ij} is a type-1 Isuffix if both of i and j are odd.
– IA_{ij} is a type-2 Isuffix if i is even and j is odd.
– IA_{ij} is a type-3 Isuffix if i is odd and j is even.
– IA_{ij} is a type-4 Isuffix if both of i and j are even.

The *partial* Isuffix tree $pIST(A)$ of A is a compacted trie that represents all type-1 Isuffixes of A.

We define the *encoded matrix* B of A. Without loss of generality, we assume that n and m are even. For all start positions $(2i-1, 2j-1)$, $1 \leq i \leq n/2$ and $1 \leq j \leq m/2$, of type-1 Isuffixes of A, we make 4-tuples $\langle A[2i-1, 2j-1], A[2i, 2j-1], A[2i-1, 2j], A[2i, 2j]\rangle$. Radix-sort all the tuples in linear time, and map each tuple into an integer in the range $[1, nm/4]$. We assign the integer of each tuple $\langle A[2i-1, 2j-1], A[2i, 2j-1], A[2i-1, 2j], A[2i, 2j]\rangle$ to $B[i, j]$ in the encoded matrix B. Note that Isuffixes of B correspond to type-1 Isuffixes of A. We call an encoded Isuffix IB_{ij} the *corresponding Isuffix* of $IA_{2i-1,2j-1}$.

Let $A_1 = A$, $A_2 = A[2 : n, 1 : m]$, $A_3 = A[1 : n, 2 : m]$, and $A_4 = A[2 : n, 2 : m]$. We append a dummy row (resp. a dummy column) at the bottom of A_2 and A_4 (resp. on the right of A_3 and A_4). Let A_{123} be the concatenation of A_1, A_2, and A_3, i.e., its size is $n \times 3m$. We denote the encoded matrix of A_{123} by B_{123}. (See Fig. 2.) Type-1 Isuffixes of matrix A_{123} correspond to type-1, type-2, and type-3 Isuffixes of A. Moreover, special characters in the rightmost column in A_1 and A_2 play a role of delimiters in A_{123}. So all type-1 Isuffixes of matrix A_{123} are distinct. Thus, all Isuffixes of matrix B_{123} are also distinct because Isuffixes of B_{123} correspond to type-1 Isuffixes of A_{123}.

In order to compare two Icharacters in constant time, we define one-dimensional suffix trees [9]. Let *rows* (resp. *cols*) be the string obtained by concatenating the rows (resp. columns) of A in row (resp. column) major order. We construct one-dimensional suffix trees T_{rows} and T_{cols} for strings *rows* and *cols*, respectively, such that all leaves in T_{rows} and T_{cols} are in lexicographic order of corresponding suffixes. This construction can be done in $O(nm)$ time for integer alphabets by Farach's algorithm [6].

The construction algorithm for the Isuffix tree $IST(A)$ consists of the following five steps.

1. Construct T_{rows} and T_{cols} of matrix A.
   After construction of T_{rows} and T_{cols} using Farach's algorithm, we perform the preprocessing for constant-time `lca` operations on T_{rows} and T_{cols}.
2. Recursively compute $IST(B_{123})$.
3. Construct $pIST(A_{123})$ from $IST(B_{123})$.
   We construct $pIST(A_{123})$ by decoding the encoded Isuffix tree $IST(B_{123})$. This decoding procedure can be done in linear time by bucket partitioning using T_{rows} and T_{cols} as in [20]. The details are described in Appendix.
4. Construct $pIST(A_4)$ using $pIST(A_{123})$.
   We use the result in [20], which constructs $pIST(A_4)$ using $pIST(A_r)$, $1 \leq r \leq 3$. We can extract $pIST(A_r)$, $1 \leq r \leq 3$, from $pIST(A_{123})$ in linear time by a tree traversal.
5. Merge $pIST(A_{123})$ and $pIST(A_4)$ into $IST(A)$.
   This *merging procedure* is described in the next section.

Our algorithm uses $\frac{3}{4}$-recursion and all steps except the recursion take linear time. If $n = 1$, matrix A is a string and so we can construct the Isuffix tree in $O(m)$ time using Farach's algorithm [6]. Hence, the worst-case running time $T(n, m)$ of our algorithm can be described by the recurrence

$$T(n,m) = \begin{cases} O(m) & \text{if } \ n = 1, \\ T(\frac{1}{2}n, \frac{3}{2}m) + O(nm) & \text{otherwise,} \end{cases}$$

whose solution is $T(n, m) = O(nm)$.

Theorem 1. *Given an $n \times m$ matrix A over integer alphabets, the Isuffix tree of A can be constructed in $O(nm)$ time.*

4 Merging Procedure

Now we describe and analyze our merging procedure that merges two partial Isuffix trees $pIST(A_{123})$ and $pIST(A_4)$ into the final Isuffix tree $IST(A)$ in linear-time.

Our merging procedure is performed by a way similar to generic merge sorting. Since we have $pIST(A_{123})$, we can find out the lexicographical order of all type-1, type-2, and type-3 Isuffixes of A in $O(nm)$ time. Also, we can find out the lexicographical order of all type-4 Isuffixes of A in $O(nm)$ time using $pIST(A_4)$. Let Lst_{123} (resp. Lst_4) be the list of Isuffixes from $pIST(A_{123})$ (resp. $pIST(A_4)$) in lexicographically sorted order. Firstly, we preprocess $pIST(A_{123})$ for constant time `lca` operation. Secondly, we choose the first Isuffixes (elements) IA_{ij} and IA_{kl} from Lst_{123} and Lst_4, respectively. Then we determine the lexicographical order of IA_{ij} and IA_{kl} and compute `Ilcp` between them. After this, we remove the smaller one from its list and add it into a new list. We do this step until one of the two lists is exhausted. Finally, we make $IST(A)$ using the merged list of all Isuffixes of A and computed `Ilcp`'s.

Consider the time complexity of our merging procedure. The first step of our merging procedure can be performed in linear time. The final step can be performed in linear time by [7]. The remaining parts are how to determine the order of IA_{ij} and IA_{kl}, and how to compute their `Ilcp`. If we can do these in constant time, we can perform all steps in $O(nm)$ time.

Consider the first different characters from IA_{ij} and IA_{kl}. The order between IA_{ij} and IA_{kl} is determined by the order of these two characters. We call these two characters the *d-characters* of IA_{ij} and IA_{kl}. (d stands for distinguishing.) Thus, if we find the d-characters, we can determine the order of IA_{ij} and IA_{kl} in constant time by a single character comparison, and also we can compute $\texttt{Ilcp}(IA_{ij}, IA_{kl})$ in constant time by a simple computation.

Now we explain how to find the d-characters. There are three cases according to the type of IA_{ij}. The first case is when IA_{ij} is a type-1 Isuffix of A, the second case is when IA_{ij} is a type-2 Isuffix of A, and the third case is when IA_{ij} is a type-3 Isuffix of A. Since IA_{ij} and IA_{kl} are in different partial Isuffix trees, it is not easy to compare them directly. Thus, instead, we compare either $IA_{i+1,j}$ and $IA_{k+1,l}$ or $IA_{i,j+1}$ and $IA_{k,l+1}$, which are in the same tree. In the second case, we compare type-1 Isuffix $IA_{i+1,j}$ and type-3 Isuffix $IA_{k+1,l}$. In the third case, we compare type-1 Isuffix $IA_{i,j+1}$ and type-2 Isuffix $IA_{k,l+1}$. In the first case, we can compare either $IA_{i+1,j}$ and $IA_{k+1,l}$ or $IA_{i,j+1}$ and $IA_{k,l+1}$. Then, since all the Isuffixes are in the same partial Isuffix tree $pIST(A_{123})$, we can compute their `Ilcp` in constant time.

1. IA_{ij} is a type-1 Isuffix of A

Assume we compare $IA_{i+1,j}$ and $IA_{k+1,l}$. Consider two suffixes α and β of *rows* of A such that α starts at (i,j) and β starts at (k,l). Assume $\texttt{Ilcp}(IA_{i+1,j}, IA_{k+1,l}) = p$ and $\texttt{lcp}(\alpha,\beta) = r$. Set $q = \lceil p/2 \rceil$. (Note that q is the number of column-type Icharacters of the longest common Iprefix between $IA_{i+1,j}$ and $IA_{k+1,l}$. See Figure 3.) Let γ and δ be suffixes of *cols* of A such that γ starts at $(i+1, j+q)$ and δ starts at $(k+1, l+q)$, respectively. Assume $\texttt{lcp}(\gamma,\delta) = s$. We can compute p, r, and s in constant time using `lca` operations on $pIST(A_{123})$, T_{rows}, and T_{cols}, respectively. There are five cases when IA_{ij} is a type-1 Isuffix of A.

(a) When $r \leq q$ (Figure 3-(a)): In this case, the d-characters are simply $A[i, j+r]$ and $A[k, l+r]$. It is easy to see that $\texttt{Ilcp}(IA_{ij}, IA_{kl}) = 2r$.

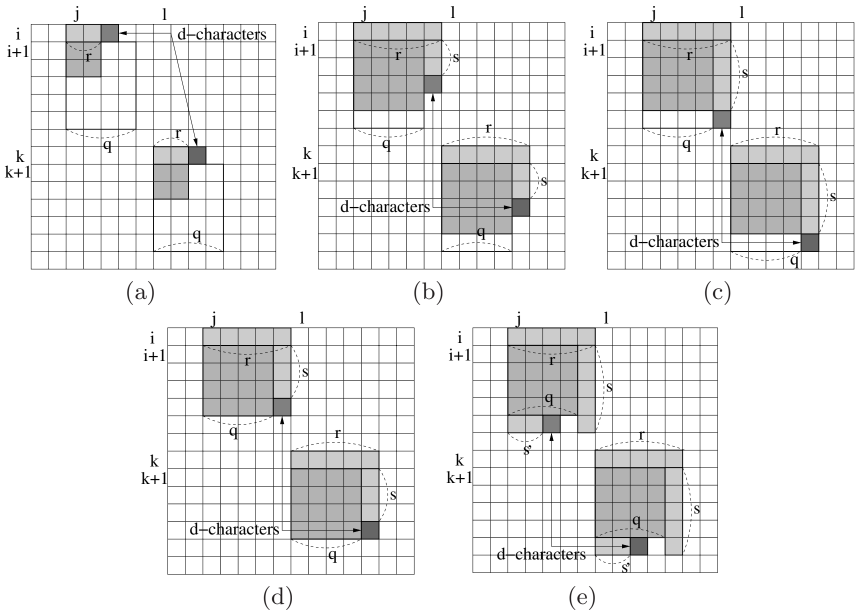


Fig. 3. Five cases when IA_{ij} is a type-1 Isuffix of A

(b) When $r > q$, p is even, and $s < q$ (Figure 3-(b)): In this case, the d-characters are $A[i+s+1, j+q]$ and $A[k+s+1, l+q]$. Since $IA_{ij}[p+1] \neq IA_{kl}[p+1]$ due to the d-characters, $\mathtt{Ilcp}(IA_{ij}, IA_{kl}) = p$.

(c) When $r > q$, p is even, and $s = q$ (Figure 3-(c)): Note that s cannot exceed q because $\mathtt{Ilcp}(IA_{i+1,j}, IA_{k+1,l}) = p$. In this case, the d-characters are $A[i+q+1, j+q]$ and $A[k+q+1, l+q]$ and $\mathtt{Ilcp}(IA_{ij}, IA_{kl}) = p+1$.

(d) When $r > q$, p is odd, and $s < q$ (Figure 3-(d)): In this case, the d-characters are $A[i+s+1, j+q]$ and $A[k+s+1, l+q]$. Also, $\mathtt{Ilcp}(IA_{ij}, IA_{kl}) = p+1$, because $IA_{ij}[p+2] \neq IA_{kl}[p+2]$ due to the d-characters.

(e) When $r > q$, p is odd, and $s \geq q$ (Figure 3-(e)): Let γ' and δ' be suffixes of *rows* such that γ' starts at $(i+q+1, j)$ and δ' starts at $(k+q+1, l)$, respectively. Assume $\mathtt{lcp}(\gamma', \delta') = s'$. Then, the d-characters are $A[i+q+1, j+s']$ and $A[k+q+1, l+s']$. Also, $\mathtt{Ilcp}(IA_{ij}, IA_{kl}) = p+2$.

2. IA_{ij} **is a type-2 Isuffix of** A
We compare type-1 Isuffix $IA_{i+1,j}$ and type-3 Isuffix $IA_{k+1,l}$ instead of comparing IA_{ij} and IA_{kl}. There are also five cases when IA_{ij} is a type-2 Isuffix of A. Details are omitted since all the cases are the same as Case 1.

3. IA_{ij} **is a type-3 Isuffix of** A
As mentioned above, we compare type-1 Isuffix $IA_{i,j+1}$ and type-2 Isuffix $IA_{k,l+1}$ instead of comparing IA_{ij} and IA_{kl}. Let α and β be suffixes of *cols* such that α starts at (i, j) and β starts at (k, l), respectively. Assume that $\mathtt{Ilcp}(IA_{i,j+1},$

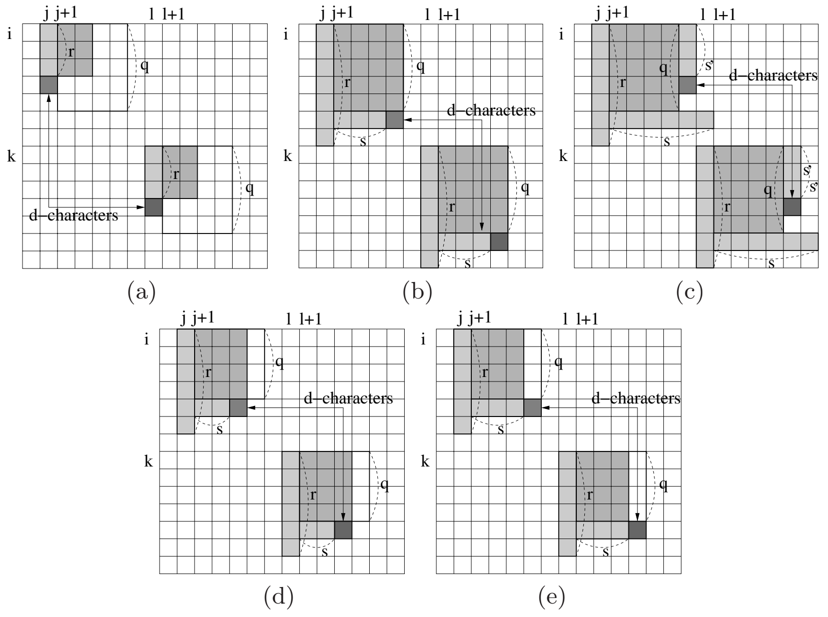


Fig. 4. Five cases when IA_{ij} is a type-3 Isuffix of A

$IA_{k,l+1}) = p$ and $\mathtt{lcp}(\alpha, \beta) = r$. Set $q = \lceil (p+1)/2 \rceil$. (Note that q is the number of *row*-type Icharacters of the longest common Iprefix between $IA_{i,j+1}$ and $IA_{k,l+1}$. See Figure 4.) Let γ and δ be suffixes of *rows* such that γ starts at $(i+q, j+1)$ and δ starts at $(k+q, l+1)$, respectively. Assume $\mathtt{lcp}(\gamma, \delta) = s$. We can compute p, r, and s in constant time using $\mathtt{lca}$ operations on $pIST(A_{123})$, T_{cols}, and T_{rows}, respectively. There are also five cases when IA_{ij} is a type-3 Isuffix of A.

(a) When $r \leq q$ (Figure 4-(a)): In this case, the d-characters are simply $A[i+r, j]$ and $A[k+r, l]$. Also, it is easy to see that $\mathtt{Ilcp}(IA_{ij}, IA_{kl}) = 2r - 1$.
(b) When $r > q$, p is even, and $s < q - 1$ (Figure 4-(b)): In this case, the d-characters are $A[i+q, j+s+1]$ and $A[k+q, l+s+1]$. Also, due to the d-characters, $IA_{ij}[p+2] \neq IA_{kl}[p+2]$, and thus $\mathtt{Ilcp}(IA_{ij}, IA_{kl}) = p+1$.
(c) When $r > q$, p is even, and $s \geq q - 1$ (Figure 4-(c)): Let γ' and δ' be suffixes of *cols* such that γ' starts at $(i, j+q)$ and δ' starts at $(k, l+q)$, respectively. Assume $\mathtt{lcp}(\gamma', \delta') = s'$. Then, the d-characters are $A[i+s', j+q]$ and $A[k+s', l+q]$. Also, $\mathtt{Ilcp}(IA_{ij}, IA_{kl}) = p+2$, because $IA_{ij}[p+3] \neq IA_{kl}[p+3]$ due to the d-characters.
(d) When $r > q$, p is odd, and $s < q - 1$ (Figure 4-(d)): In this case, the d-characters are $A[i+q, j+s+1]$ and $A[k+q, l+s+1]$. Also, $\mathtt{Ilcp}(IA_{ij}, IA_{kl}) = p$, because $IA_{ij}[p+1] \neq IA_{kl}[p+1]$ due to the d-characters.
(e) When $r > q, p$ is odd, and $s = q-1$ (Figure 4-(e)): In this case, the d-characters ar $A[i+q, j+q]$ and $A[k+q, l+q]$. Also, $\mathtt{Ilcp}(IA_{ij}, IA_{kl}) = p+1$, because

$IA_{ij}[p+1] = IA_{kl}[p+1]$ and $IA_{ij}[p+2] \neq IA_{kl}[p+2]$ due to the d-characters. Note that s cannot exceed $q-1$ because $\mathtt{Ilcp}(IA_{i,j+1}, IA_{k,l+1}) = p$.

Lemma 1. *Given T_{rows}, T_{cols}, and $pIST(A_{123})$ with constant time* `lca` *operation, we can decide the order of a type-123 Isuffix and a type-4 Isuffix, and can compute their* `Ilcp` *in constant time.*

Theorem 2. *Two partial Isuffix trees $pIST(A_{123})$ and $pIST(A_4)$ can be merged into the Isuffix tree $IST(A)$ in $O(nm)$ time.*

References

1. Amir, A., Farach, M.: Two-dimensional dictionary matching. IPL 21, 233–239 (1992)
2. Apostolico, A.: The myriad virtues of subword trees. In: Apostolico, A., Galil, Z. (eds.) Combinatorial Algorithms on Words, pp. 85–95. Springer, Heidelberg (1985)
3. Chen, M.T., Seiferas, J.: Efficient and elegant subword tree construction. In: Apostolico, A., Galil, Z. (eds.) Combinatorial Algorithms on Words. NATO Advanced Science Institutes. F, vol. 12, pp. 97–107. Springer, Heidelberg (1985)
4. Cole, R., Hariharan, R.: Faster suffix tree construction with missing suffix links. In: Proc. of the 30th STOC, pp. 407–415 (2000)
5. Crochemore, M., Rytter, W.: Jewels of Stringology. World Scientific Publishing, Singapore (2002)
6. Farach, M.: Optimal suffix tree construction with large alphabets. In: Proc. of the 38th FOCS, pp. 137–143 (1997)
7. Farach, M., Muthukrishnan, S.: Optimal logarithmic time randomized suffix tree construction. In: Proc. of the 23rd ICALP, pp. 550–561 (1996)
8. Farach-Colton, M., Ferragina, P., Muthukrishnan, S.: On the sorting-complexity of suffix tree construction. J. ACM 47(6), 987–1011 (2000)
9. Giancarlo, R.: A generalization of the suffix tree to square matrices, with application. SIAM J. on Comp. 24(3), 520–562 (1995)
10. Giancarlo, R., Grossi, R.: On the construction of classes of suffix trees for square matrices: Algorithms and applications. Information and Computation 130(2), 151–182 (1996)
11. Giancarlo, R., Grossi, R.: Suffix tree data structures for matrices. In: Apostolico, A., Galil, Z. (eds.) Pattern Matching Algorithms, chapter 11, pp. 293–340. Oxford University Press, Oxford (1997)
12. Giancarlo, R., Guaiana, D.: On-line construction of two-dimensional suffix trees. Journal of Complexity 15(1), 72–127 (1999)
13. Gonnet, G.H.: Efficient searching of text and pictures. Technical Report OED-88-02, University of Waterloo (1988)
14. Gusfield, D.: Algorithms on Strings, Tree, and Sequences. Cambridge University Press, Cambridge (1997)
15. Harel, D., Tarjan, R.: Fast algorithms for finding nearest common ancestors. SIAM J. on Comp. 13(2), 338–355 (1984)
16. Hariharan, R.: Optimal parallel suffix tree construction. In: Proc. of the 26th FOCS, pp. 290–299 (1994)
17. Hon, W.K., Sadakane, K., Sung, W.K.: Breaking a time-and-space barrier in constructing full-text indices. In: Proc. of the 44th FOCS, pp. 251–260 (2003)

18. Kärkkäinen, J., Sanders, P., Burkhardt, S.: Linear work suffix array construction. J. ACM (in press)
19. Kim, D.K., Kim, Y.A., Park, K.: Generalizations of suffix arrays to multi-dimensional matrices. TCS 302(1-3), 401–416 (2003)
20. Kim, D.K., Park, K.: Linear-time construction of two-dimensional suffix trees. In: Proc. of the 26th ICALP, pp. 463–372 (1999)
21. Kim, D.K., Sim, J.S., Park, H., Park, K.: Constructing suffix arrays in linear time. J. Disc. Alg. 3(2-4), 126–142 (2005)
22. Ko, P., Aluru, S.: Space efficient linear time construction of suffix arrays. J. Disc. Alg. 3(2-4), 143–156 (2005)
23. Manber, U., Myers, G.: Suffix arrays: A new method for on-line string searches. SIAM J. on Comp. 22(5), 935–948 (1993)
24. McCreight, E.M.: A space-economical suffix tree construction algorithms. J. ACM 23(2), 262–272 (1976)
25. Na, J.C., Giancarlo, R., Park, K.: O(n^{2}logn) time on-line construction of two-dimensional suffix trees. In: Proc. of the 11th COCOON, pp. 273–282 (2005)
26. Sahinalp, S.C., Vishkin, U.: Symmetry breaking for suffix tree construction. In: Proc. of the 26th FOCS, pp. 300–309 (1994)
27. Schieber, B., Vishkin, U.: On finding lowest common ancestors: Simplification and parallelization. SIAM J. on Comp. 17(6), 1253–1262 (1988)
28. Ukkonen, E.: On-line construction of suffix trees. Algorithmica 14(3), 249–260 (1995)
29. Weiner, P.: Linear pattern matching algorithms. In: Proc. of the 14th IEEE Symp. on Switching and Automata Theory, pp. 1–11 (1973)

Appendix: Decoding the Encoded Isuffix Trees

Given an $n \times m$ matrix V and its encoded matrix W, we describe how to construct $pIST(V)$ from the encoded Isuffix tree $IST(W)$.

First, let us consider some relations between a (type-1) Isuffix $IV_{2i-1,2j-1}$ and its corresponding Isuffix IW_{ij}. For simplicity, we denote $IV_{2i-1,2j-1}$ and IW_{ij} by IC and ID, respectively. As shown in Figure 5, two Icharacters $ID[2k..2k+1]$ correspond to four Icharacters $IC[4k..4k+3]$ For example, $ID[2..3]$ correspond to $IC[4..7]$ (when $k = 1$). We call $IC[4k]$, $IC[4k+1]$, $IC[4k+2]$, and $IC[4k+3]$ *type*-I, *type*-II, *type*-III, and *type*-IV Icharacters, respectively.

A difficulty that arises in decoding $IST(W)$ is that the lexicographic order of Isuffixes of V may not be that of corresponding Isuffixes of W. As shown in Figure 5, $IC[4..7]$ represents different linearized string from which $ID[2..3]$ does. We solve this difficulty by combining two levels in $IST(W)$ and then splitting into four levels in $pIST(V)$ using T_{rows} and T_{cols} of V. (Intuitively, this means that we combine two Icharacters of W and split it to four Icharacters of V.)

Our decoding procedure consists of four stages.

1. Construct the *Lsuffix tree* $LST(W)$ by combining two levels of $IST(W)$.
2. Construct the *decoded Lsuffix tree* $DLST(V)$ from $LST(W)$ by modifying the labels of $LST(W)$ to those of V.
3. Obtain some information on orders of Icharacters at nodes of $DLST(V)$.
4. Construct $pIST(V)$ by sorting and partitioning children at nodes in $DLST(V)$.

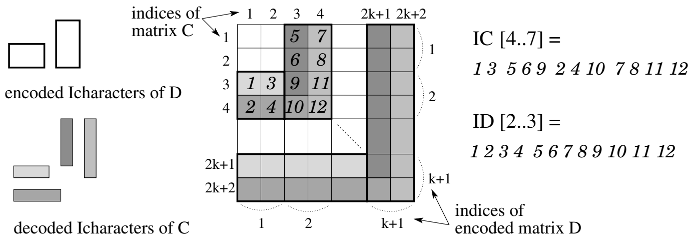


Fig. 5. IC and its corresponding Isuffix ID

Stage 1. Construct $LST(W)$ by combining two levels $(2k, 2k+1)$ of $IST(W)$.

While traversing $IST(W)$, we do the following for an internal node u except the root such that $L(u)$ is even (say $2k$). Let w be the parent of u. We have two cases.

- If $L(w)$ is $2k-1$: Let Ib be an encoded Icharacter $label(w, u)$. For every edge e of u, we add Ib in front of the label of e and add e to the edge list of w (we do not care the rank of e in w). Then, we remove u from the tree.
- If $L(w)$ is less than $2k-1$: Let Ib be the last encoded Icharacter of $label(w, u)$. We raise the level of u to $2k-1$ by relabeling edge (w, u) to $label(w, u) - Ib$ and adding Ib in front of the labels of all edges of u.

Stage 2. Construct $DLST(V)$ by modifying labels in $LST(W)$ into those of V.

For each edge (u, v) in $LST(W)$, let $label(u, v)$ be $(i, j, 2\ell_1, 2\ell_2+1)$, i.e., $IW_{ij}[2\ell_1..2\ell_2+1]$. Because an encoded Lcharacter $IW_{ij}[2k..2k+1]$ corresponds to an Lcharacter $IV_{2i-1,2j-1}$ $[4k..4k+3]$, we change $(i, j, 2\ell_1, 2\ell_2+1)$ into $(2i-1, 2j-1, 4\ell_1, 4\ell_2+3)$.

Stage 3. Compute the orders of Icharacters needed for splitting levels of $DLST(V)$.

For an internal node u of $DLST(V)$, let $v_1, \ldots, v_r$ be the children of u. Let $E_{\mathrm{I}}(u)$ (resp. $E_{\mathrm{II}}(u)$, $E_{\mathrm{III}}(u)$, and $E_{\mathrm{IV}}(u)$) denote the edge list of u sorted by the first (resp. second, third, and fourth) Icharacters of labels of edges $(u, v_1), \ldots, (u, v_r)$.

We explain how to make sorted lists E_{I}'s of all nodes in linear time after constructing T_{rows} of V. We will use the leaves of T_{rows} as buckets. Recall that all leaves in T_{rows} are in lexicographic order. For $1 \le k \le r$, let Ia_k be the first Icharacter of $label(u, v_k)$ and (i_k, j_k) be a position in V where Ia_k starts. Note that Ia_k is a type-I Icharacter. We call the index (i_k, j_k) the *type*-I *position* of edge (u, v_k). Notice that positions $(i_1, j_1), \ldots, (i_r, j_r)$ must be distinct. The lexicographic order of $Ia_1, \ldots, Ia_r$ is the lexicographic order of the leaves $l_{i_1 j_1}, \ldots, l_{i_r j_r}$ in T_{rows}. Initially, E_{I}'s are empty.

1. Insert every edge (u, v_k) into bucket $l_{i_k j_k}$ in T_{rows} for all internal nodes u.

2. Scan all leaves in T_{rows} in lexicographic order. For each leaf l in T_{rows}, we extract every edge (u, v) from bucket l and determine the rank of v among the children of u by adding (u, v) to the end of $E_{\mathrm{I}}(u)$.

Similarly, we can construct E_{III} (resp. E_{II} and E_{IV}) using T_{rows} (resp. T_{cols}).

Stage 4. Construct $pIST(V)$ by refining edges in $DLST(V)$.

We describe how to partition edges of an internal node u of $DLST(V)$ in $O(r)$ time, where r is the number of edges of u. Let $|L(u)| = 4k - 1$.

1. Reorder and partition by the first Icharacters.
   (a) Sort the children of u in lexicographical order of their first Icharacters using E_{I}. Let $e_1, \ldots, e_r$ be the edges of u in the order of their first Icharacters.
   (b) Partition the children of u. Let $(i_1, j_1), \ldots, (i_r, j_r)$ be the type-I positions of edges $e_1, \ldots, e_r$, respectively. Initially, we create a new node v_1 as a child of u and insert e_1 into bucket v_1. For every adjacent indices (i_c, j_c) and (i_d, j_d) (i.e., $d = c + 1$), do the following. Find the node $w = \mathtt{lca}(l_{i_c j_c}, l_{i_d j_d})$ in T_{row}. Then, $|L(w)|$ is the $\mathtt{lcp}$ between $V[i_c, j_c : m]$ and $V[i_d, j_d : m]$. Let v_x be the node (bucket) where e_c is stored. If $|L(w)| \geq 2k$ (i.e., two type-I Icharacter is the same), then e_d is inserted into v_x. Otherwise, we create a new node v_y as a child of u and insert e_d into bucket v_y.
2. Reorder and partition by the second Icharacters. Let $v_1, \ldots, v_q$ be the children of u created in Step 1.
   (a) Sort edges in each node v_i in lexicographical order of type-II Icharacters simultaneously. It can be done in $O(r)$ time by scanning $E_{\mathrm{II}}(u)$.
   (b) Partition the edges in each node v_i by type-II Icharacter using the same method as partitioning the edges of u in Step 1 except that we use T_{cols} at this time.
3. 4. Similarly, reorder and partition by the third and the fourth Icharacters.
5. Eliminate nodes with only one child. Some nodes (including u) can have only one child if edges are partitioned into one bucket.

* In special case, the partitioning at the root is started by type-II Icharacters.

Author Index