Lecture Notes in Computer Science 6129

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

Amihood Amir Laxmi Parida (Eds.)

Combinatorial Pattern Matching

21st Annual Symposium, CPM 2010
New York, NY, USA, June 21-23, 2010
Proceedings

Springer

Volume Editors

Amihood Amir
Johns Hopkins University
Baltimore, MD, USA
and
Bar-Ilan University
Department of Computer Science
52900 Ramat-Gan, Israel
E-mail: amir@macs.biu.ac.il

Laxmi Parida
IBM T.J. Watson Research Center
Yorktown Heights, NY, USA
E-mail: parida@us.ibm.com

Library of Congress Control Number: 2010927801

CR Subject Classification (1998): F.2, I.5, H.3.3, J.3, I.4.2, E.4, G.2.1, E.1

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

ISSN 0302-9743
ISBN-10 3-642-13508-0 Springer Berlin Heidelberg New York
ISBN-13 978-3-642-13508-8 Springer Berlin Heidelberg New York

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

Preface

The papers contained in this volume were presented at the 21st Annual Symposium on Combinatorial Pattern Matching (CPM 2010) held at NYU-Poly, Brooklyn, New York during June 21–23, 2010.

All the papers presented at the conference are original research contributions. We received 53 submissions from 21 countries. Each paper was reviewed by at least three reviewers. The committee decided to accept 28 papers. The program also includes three invited talks by Zvi Galil from Tel Aviv University, Israel, Richard M. Karp from University of California at Berkeley, USA, and Jeffrey S. Vitter from Texas A&M University, USA.

The objective of the annual CPM meetings is to provide an international forum for research in combinatorial pattern matching and related applications. It 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 meeting also deals with problems in computational biology, data compression and data mining, coding, information retrieval, natural language processing and pattern recognition.

The Annual Symposium on Combinatorial Pattern Matching started in 1990, and has since taken place every year. Previous CPM meetings were held in Paris, London, Tucson, Padova, Asilomar, Helsinki, Laguna Beach, Aarhus, Piscataway, Warwick, Montreal, Jerusalem, Fukuoka, Morelia, Istanbul, Jeju Island, Barcelona, London, Ontario, Pisa, and Lille.

Starting from the third meeting, 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, 4580, 5029, 5577, and 6129.

Selected papers from the first meeting appeared in volume 92 of *Theoretical Computer Science*, from the 11th meeting in volume 2 of *Journal of Discrete Algorithms*, from the 12th meeting in volume 146 of *Discrete Applied Mathematics*, from the 14th meeting in volume 3 of *Journal of Discrete Algorithms*, from the 15th meeting in volume 368 of *Theoretical Computer Science*, from the 16th meeting in volume 5 of *Journal of Discrete Algorithms*, and from the 19th meeting in volume 410 of *Theoretical Computer Science*.

The whole submission and review process was carried out with the help of the EasyChair conference system. The conference was sponsored by the NYU-Poly, Brooklyn, and by IBM Research. Special thanks are due to the members of the Program Committee who worked very hard to ensure the timely review of all

the submitted manuscripts, and participated in stimulating discussions that led to the selection of the papers for the conference.

April 2010

Amihood Amir
Laxmi Parida

Organization

Program Committee

Amihood Amir	Johns Hopkins University, USA, and Bar-Ilan University, Israel (Co-chair)
Rolf Backofen	Albert-Ludwigs-Universität Freiburg, Germany
Ayelet Butman	Holon Academic Institute of Technology, Holon, Israel
Matteo Comin	University of Padova, Italy
Miklós Csurös	Université de Montréal, Canada
Petros Drineas	Rensselaer Polytechnic Institute, USA
Leszek Gasieniec	University of Liverpool, UK
Steffen Heber	North Carolina State University, USA
John Iacono	Polytechnic Institute of New York University, USA
Shunsuke Inenaga	Kyushu University, Japan
Rao Kosaraju	Johns Hopkins University, USA
Gregory Kucherov	Laboratoire d'Informatique Fondamentale de Lille, France
Gad Landau	NYU-Poly, USA, and University of Haifa, Israel
Thierry Lecroq	University of Rouen, France
Avivit Levy	Shenkar College and CRI, University of Haifa, Israel
Ion Mandoiu	University of Connecticut, USA
Avi Ma'ayan	Mount Sinai, USA
Gonzalo Navarro	University of Chile, Chile
Laxmi Parida	IBM T.J. Watson Research Center, USA (Co-chair)
Heejin Park	Hanyang University, Korea
Nadia Pisanti	University of Pisa, Italy
Ely Porat	Bar-Ilan University, Israel
Naren Ramakrishnan	Virginia Tech, USA
Marie-France Sagot	INRIA, France
Rahul Shah	Louisiana State University, USA
Dennis Shasha	New York University, USA
Dina Sokol	City University of New York, USA
Torsten Suel	Polytechnic Institute of NYU, USA
Jens Stoye	Universität Bielefeld, Germany
Oren Weimann	Weizmann Institute of Science, Israel

Yufeng Wu	University of Connecticut, USA
Dekel Tsur	Ben Gurion University of the Negev, Israel
Michal Ziv-Ukelson	Ben Gurion University of the Negev, Israel

Organizing Committee

Gad Landau	NYU-Poly, USA, and University of Haifa, Israel
Laxmi Parida	IBM T.J. Watson Research Center, USA

Steering Committee

Alberto Apostolico	University of Padova, Italy, and Georgia Institute of Technology, USA
Maxime Crochemore	Université Paris-Est, France, and King's College London, UK
Zvi Galil	Columbia University, USA, and Tel Aviv University, Israel

Web and Publications Committee

Asif Javed	IBM T.J. Watson Research Center, USA

External Referees

Hideo Bannai
Michaël Cadilhac
Sabrina Chandrasekaran
Francisco Claude
Maxime Crochemore
Danny Hermelin
Wing Kai Hon
Brian Howard
Peter Husemann
Asif Javed
Erez Katzenelson
Takuya Kida
Sung-Ryul Kim
Tsvi Kopelowitz
Alexander Lachmann
Taehyung Lee
Arnaud Lefebvre
Zsuzsanna Liptak
Nimrod Milo
Mathias Möhl
Joong Chae Na
Shoshana Neuburger
Marius Nicolae
Ge Nong
Pierre Peterlongo
Tamar Pinhas
Yoan Pinzon
Boris Pismenny
Igor Potapov
Sven Rahmann
Paolo Ribeca
Luis M.S. Russo
Jeong Seop Sim
Tatiana Starikovskaya
Sharma Thankachan
Alex Tiskin
Charalampos Tsourakakis
Fabio Vandin

Rossano Venturini
Davide Verzotto
Isana Vexler-Lublinsky
Sebastian Will
Prudence W.H. Wong
Shay Zakov

Sponsoring Institutions

IBM Research
NYU-Poly, Brooklyn

Table of Contents

Algorithms for Forest Pattern Matching

Kaizhong Zhang and Yunkun Zhu

Dept. of Computer Science, University of Western Ontario,
London, Ontario N6A 5B7, Canada
kzhang@csd.uwo.ca, yzhu233@csd.uwo.ca

Abstract. Ordered labelled trees are trees where the left-to-right order among siblings is significant. An ordered labelled forest is a sequence of ordered labelled trees. Given an ordered labelled forest F ("the target forest") and an ordered labelled forest G ("the pattern forest"), the *forest pattern matching problem* is to find a sub-forest F' of F such that F' and G are the most similar over all possible F'. In this paper, we present efficient algorithms for the forest pattern matching problem for two types of sub-forests: closed subforests and closed substructures. As RNA molecules' secondary structures could be represented as ordered labelled forests, our algorithms can be used to locate the structural or functional regions in RNA secondary structures.

1 Introduction

An ordered labelled tree is a tree where the left-to-right order among siblings is significant and each node is labelled by a symbol from a given alphabet. An ordered labelled forest is a sequence of ordered labelled trees. Ordered labelled trees and forests are very useful data structures for hierarchical data representation. In this paper, we refer to ordered labelled trees and ordered labelled forests as trees and forests, respectively.

Among numerous applications where trees and forests are useful representations of objects, the need for comparing trees and forests frequently arises. As a typical example, consider the secondary structure comparison problem for RNA. Since RNA is a single strand of nucleotides, it folds back onto itself into a shape that is topologically a forest [14,3,6,10], which we call its secondary structure. Figure 1 which is adapted from [5] shows an example of the RNA GI:2347024 structure, where (a) is a segment of the RNA sequence, (b) is its secondary structure and (c) is the forest representation. Algorithms for the edit distance between forests (tree) [15,2] could be used to measure the global similarity between forests (trees). Motivated mainly by the problem of locating structural or functional regions in RNA secondary structures, the forest (tree) pattern matching (FPM) problem became interesting and attracted some attention [3,4,5,6].

In this paper, the forest pattern matching (FPM) problem is defined as the following: Given a target forest F and a pattern forest G, find a sub-forest F' of

A. Amir and L. Parida (Eds.): CPM 2010, LNCS 6129, pp. 1–12, 2010.

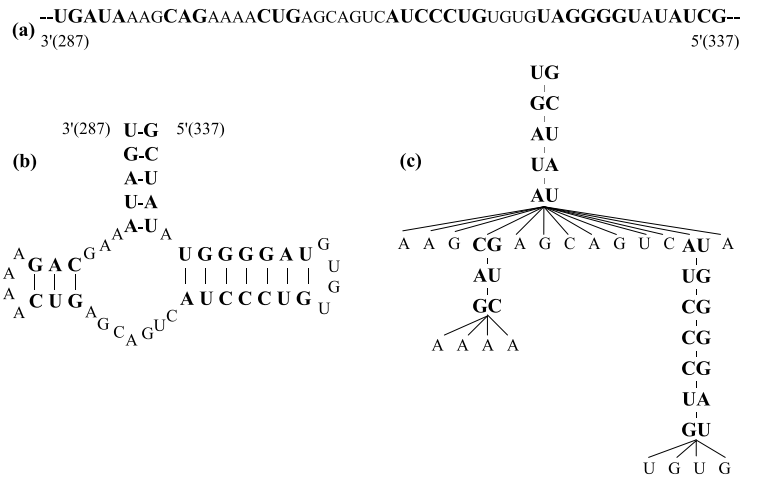


Fig. 1. (a) A segment of the RNA GI: 2347024 primary structure [8], (b) its secondary structure, (c) its forest representation

F which is the most similar to G over all possible F'. There are various ways to define the term "sub-forest". For the definition of "sub-forest" as substructures or simple substructures [5], algorithms have been developed [15,5]. We consider two alternative definitions of "sub-forest": closed subforests and closed substructures.

For the closed subforests definition, we present an efficient algorithm which improved the complexity results given in [5] of CPM 2006. For the new closed substructures definition, we present an efficient algorithm.

2 Preliminaries

Throughout this paper, we use the following definitions and notations.

Let F be any given forest, we use a left-to-right postorder numbering of the nodes in F. $|F|$ denotes the number of nodes in F. In the postorder numbering, $F[i..j]$ will generally be an ordered subforest of F induced by the nodes numbered from i to j inclusive. Let $f[i]$ be the ith node in F and $F[i]$ be the subtree rooted at $f[i]$. A subforest of F is an ordered sequence of subtrees of F. A substructure of F is any connected sub-graph of F. $l(i)$ denotes the postorder number of the leftmost leaf descendant of $f[i]$. We say $f[i_1]$ and $f[i_2]$ (or just i_1 and i_2) are siblings if they have the same parent. D_F and L_F denote the depth and the number of leaves of F respectively. To simplify the presentation, we assume that the forest F has an imaginary parent node, denoted by $p(F)$. Finally we define the key roots of F as the set $K(F) = \{p(F)\} \cup \{i \mid i \in F \text{ and } i \text{ has a left sibling}\}$ and from [15] we have $|K(F)| \leq L_F$.

2.1 Forest Edit Distance

Our algorithms are based on forest edit distance. For forest edit distance, there are three edit operations on a forest F. (1) Change: to change one node label to another in F. (2) Delete: to delete a node i from F (making the children of i become the children of the parent of i and then removing i). (3) Insert: to insert a node i into F (the complement of delete). An edit operation can be represented as (a, b) where a and b are labels of forest nodes or a null symbol indicating insertion or deletion. Let γ be a cost function that assigns to each edit operation (a, b) a nonnegative real number $\gamma(a, b)$. We constrain γ to be a distance metric.

Let S be a sequence $s_1, \cdots, s_k$ of edit operations. An S-derivation from A to B is a sequence of forests $A_0, \cdots, A_k$ such that $A = A_0$, $B = A_k$, and $A_{i-1} \rightarrow A_i$ via s_i for $1 \leq i \leq k$. We extend γ to the sequence S by letting $\gamma(S) = \sum_{i=1}^{i=|S|} \gamma(s_i)$. Formally the distance between F and G is defined as follows:

$$\delta(F, G) = \min\{\gamma(S) \mid S \text{ is an edit operation sequence taking } F \text{ to } G\}.$$

The definition of γ makes δ a distance metric also. Equivalently forest edit distance can also be defined using the minimum cost mapping between two forests [11,15].

Tai [11] gave the first algorithm for computing the tree edit distance between two given trees F and G (one can easily extend this algorithm to the forest edit distance problem). Zhang and Shasha [15] gave a more efficient algorithm for this problem running in $O(|F| \cdot |G| \cdot \min\{D_F, L_F\} \cdot \min\{D_G, L_G\})$ time and $O(|F| \cdot |G|)$ space. More recently, Klein [7], Touzet [12], and Demaine *et al.* [2] developed faster algorithms which have better time complexity in the worst case. The algorithm of Demaine *et al.* [2] runs in $O(|F| \cdot |G|^2 \cdot (1 + \log \frac{|F|}{|G|}), |G| < |F|$, time and $O(|F| \cdot |G|)$ space.

2.2 Sub-forest Definitions

Let F be a forest. In this paper, we consider two types of sub-forests of F: (1) a *closed subforest*: a sequence of subtrees of F such that their roots are consecutive siblings. (2) a *closed substructure*: a sequence of substructures of F such that their roots are consecutive siblings. Figure 2 shows these two types of sub-forests of the forest F in Figure 1(c). Here, F_1 is a substructure of F, F_2 is a closed subforest of F, and F_3 is a closed substructure of F.

Therefore, we can now define the forest pattern matching (FPM) problem more formally: given a target forest F and a pattern forest G, find a *sub-forest* (using any one of the above definitions for "sub-forest") F' of F which minimizes the forest edit distance to G over all possible F'.

Forest pattern matching problem for substructures have been studied in [15,5]. Forest pattern matching problem for closed subforests has been studied in [5]. We propose the forest pattern matching problem for closed substructures. The motivation is from the forest representation of RNA secondary structures. In

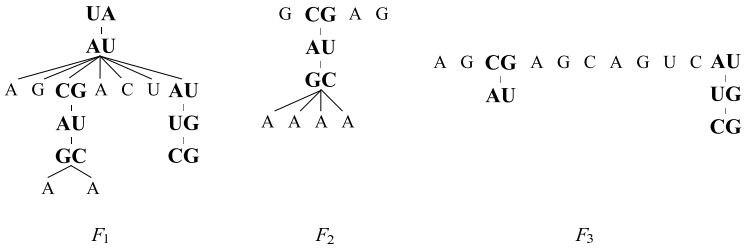


Fig. 2. Examples of variant sub-forest of the forest F in Figure 1(c)

this representation, see Figure 1, sibling nodes are in fact connected through the backbone of RNA primary structure. Therefore, it is natural to assume that, in addition to the parent child connection, a node is also connected to its left and right siblings. Hence a closed substructure we defined is just a connected sub-graph in this representation. A closed substructure can be used to represent a pattern local to a multiple loop, although it does not imply a physically connected RNA fragment in a tree representatation of RNA [1].

2.3 Previous Work and Our Results

The problem of finding a most similar "closed subforest" was discussed by Jansson and Peng in their CPM 2006 paper [5] and their algorithm runs in $O(|F| \cdot |G| \cdot L_F \cdot \min\{D_G, L_G\})$ time and $O(|F| \cdot |G| + L_F \cdot D_F \cdot |G| + |F| \cdot L_G \cdot D_G)$ space.

In this paper, we show how to solve the forest pattern matching (FPM) problem efficiently based on [15,2] for two types of sub-forests, "closed subforest" and "closed substructure". The time complexity of our algorithms are summarized in Table 1 and the space complexity of our algorithm is $O(|F| \cdot |G|)$.

Table 1. Our results

FPM	Time complexity	Section
Closed subforest	$\frac{O(\lvert F\rvert \cdot \lvert G\rvert \cdot \min\{D_F, L_F\} \cdot \min\{D_G, L_G\})}{O(\lvert F\rvert \cdot \lvert G\rvert \cdot (\lvert G\rvert \cdot (1 + \log \frac{\lvert F\rvert}{\lvert G\rvert}) + \min\{D_F, L_F\}))}$	3.1
Closed substructure	$\frac{O(\lvert F\rvert \cdot \lvert G\rvert \cdot \min\{D_F, L_F\} \cdot \min\{D_G, L_G\})}{O(\lvert F\rvert \cdot \lvert G\rvert \cdot (\lvert G\rvert \cdot (1 + \log \frac{\lvert F\rvert}{\lvert G\rvert}) + \min\{D_F, L_F\}))}$	3.2

Compared with the algorithm of Jansson and Peng [5], our first algorithm solving the same problem is faster and uses less space. Our second algorithm solves the problem of finding a most similar closed substructure which could be used to search an RNA structural pattern local to a multiple loop.

3 Algorithms for the Forest Pattern Matching Problem

In this section, we present efficient algorithms for forest pattern matching problem for two types of sub-forests, closed subforest and closed substructure, respectively. We also refer the forest pattern matching problem as the problem of *finding a most similar sub-forest.*

3.1 An Algorithm for Finding a Most Similar Closed Subforest

Given a target forest F and a pattern forest G, our goal is to compute

$$\min\{\delta(F[l(i_1)..i_2], G) \mid i_1 \text{ and } i_2 \text{ are siblings}\}.$$

Jansson and Peng [5] gave an algorithm for this problem in $O(|F| \cdot |G| \cdot L_F \cdot \min\{D_G, L_G\})$ time and $O(|F| \cdot |G| + L_F \cdot D_F \cdot |G| + |F| \cdot L_G \cdot D_G)$ space. We present an algorithm which is more efficient in both time and space. Our algorithm combines the idea of [15] and the method of approximate pattern matching for sequences [9,13].

We first examine sequence pattern matching method and then give a natural extension from sequence pattern matching to forest pattern matching for closed subforest.

Given a pattern sequence $P[1..m]$ and a text sequence $T[1..n]$, the problem is to find a segment $T[k..l]$ in the text which yields a minimum edit distance to $P[1..m]$. The idea is that in calculating the score for $T[1..i]$ and $P[1..j]$, any prefix of $T[1..i]$ could be deleted without any penalty. In other words, the score is $\min\{\delta(T[i_1..i], P[1..j]) \mid 1 \le i_1 \le i+1\}$.

Now consider a node i in a forest F and let the degree of node i be d_i and its children be $i_1, i_2, \ldots, i_{d_i}$. Given a pattern forest G, we would like to find u and v such that $F[l(i_u)..i_v]$ yields the minimum distance to G. Let $k \in F[i_s]$ where $1 \le s \le d_i$, how could we define a score $\Delta(F[l(i_1)..k], G[1..j])$ for $F[l(i_1)..k]$ and $G[1..j]$ in order to extend the definition from sequences to forests?

If $k \in \{i_1, i_2, \ldots, i_{d_i}\}$, then $F[l(i_1)..k]$ is a sequence of sibling trees, i.e. $T[i_1], \ldots, T[i_s]$. $\Delta(F[l(i_1)..k], G[1..j])$ is therefore defined as $\min\{\delta(F[l(i_t)..i_s], G[1..j]) \mid 1 \le t \le s+1\}$ where $\delta(,)$ is the forest edit distance. In particular, if $s = 1$, then the score is $\min\{\delta(F[l(i_1)..i_1], G[1..j]), \delta(\emptyset, G[1..j])\}$ which can be obtained directly using the forest edit distance algorithm.

If $k \notin \{i_1, i_2, \ldots, i_{d_i}\}$, then $\Delta(F[l(i_1)..k], G[1..j])$ is defined as $\min\{\delta(F[l(i_t)..k], G[1..j]) \mid 1 \le t \le s\}$ since $F[l(i_s)..k]$ is a proper part of $F[i_s]$ that can not be deleted without penalty.

With this definition, $\min\{\Delta(F[l(i_1)\ ..i_t], G[1..|G|]) \mid 1 \le t \le d_i\}$ is what we want to compute for node i. We have the following two lemmas for the the calculation of $\Delta(F[l(i_1)..k], G[1..j])$.

Lemma 1. *Let i, F and G be defined as the above, $i_1 \le k \le i_{d_i}$ and $1 \le j \le |G|$, then*

$$\Delta(\emptyset,\emptyset) = 0;$$
$$\Delta(F[l(i_1)..k],\emptyset) = \begin{cases} 0 & \text{if } k \in \{i_1,\ldots,i_{d_i}\} \\ \Delta(F[l(i_1)..k-1],\emptyset) + \gamma(f[k],-); & \text{otherwise} \end{cases}$$
$$\Delta(F[l(i_1)..i_1],G[1..j]) = \min \begin{cases} \delta(F[l(i_1)..i_1],G[1..j]) \\ \delta(\emptyset,G[1..j]). \end{cases}$$

Proof. This is directly from the above definition. □

Lemma 2. *Let i, F and G be defined as the above, $i_1 < k \le i_{d_i}$ and $1 \le j \le |G|$, then*

$$\begin{aligned} &\Delta(F[l(i_1)..k],G[1..j]) \\ &= \min \begin{cases} \Delta(F[l(i_1)..k-1],G[1..j]) + \gamma(f[k],-), \\ \Delta(F[l(i_1)..k],G[1..j-1]) + \gamma(-,g[j]), \\ \Delta(F[l(i_1)..l(k)-1],G[1..l(j)-1]) + \delta(F[l(k)..k],G[l(j)..j]). \end{cases} \end{aligned}$$

Proof. We prove this lemma inductively. The base case is $k = i_1 + 1$ where we need the fact that $\Delta(F[l(i_1)..i_1],G[1..j]) = \min\{\delta(F[l(i_1)..i_1],G[1..j]), \delta(\emptyset,G[1..j])\}$.

For $k > i_1$ and $k \in \{i_2,i_3,\ldots,i_{d_i}\}$,

$$\begin{aligned} &\min \begin{cases} \Delta(F[l(i_1)..k-1],G[1..j]) + \gamma(f[k],-) \\ \Delta(F[l(i_1)..k],G[1..j-1]) + \gamma(-,g[j]) \\ \Delta(F[l(i_1)..l(k)-1],G[1..l(j)-1]) + \delta(F[l(k)..k],G[l(j)..j]) \end{cases} \\ &= \min \begin{cases} \min\{\delta(F[l(i_t)..k-1],G[1..j]) \mid 1 \le t \le s\} + \gamma(f[k],-) \\ \min\{\delta(F[l(i_t)..k],G[1..j-1]) \mid 1 \le t \le s+1\} + \gamma(-,g[j]) \\ \min\{\delta(F[l(i_t)..l(k)-1],G[1..l(j)-1]) \mid 1 \le t \le s\} \\ \qquad + \delta(F[l(k)..k],G[l(j)..j]) \end{cases} \\ &= \min \begin{cases} \min\{\delta(F[l(i_t)..k],G[1..j]) \mid 1 \le t \le s\} \\ \delta(\emptyset,G[1..j]). \end{cases} \\ &= \min\{\delta(F[l(i_t)..k],G[1..j]) \mid 1 \le t \le s+1\} \\ &= \Delta(F[l(i_1)..k],G[1..j]). \end{aligned}$$

For $k > i_1$ and $k \notin \{i_2,i_3,\ldots,i_{d_i}\}$,

$$\begin{aligned} &\min \begin{cases} \Delta(F[l(i_1)..k-1],G[1..j]) + \gamma(f[k],-) \\ \Delta(F[l(i_1)..k],G[1..j-1]) + \gamma(-,g[j]) \\ \Delta(F[l(i_1)..l(k)-1],G[1..l(j)-1]) + \delta(F[l(k)..k],G[l(j)..j]) \end{cases} \\ &= \min \begin{cases} \min\{\delta(F[l(i_t)..k-1],G[1..j]) \mid 1 \le t \le s\} + \gamma(f[k],-) \\ \min\{\delta(F[l(i_t)..k],G[1..j-1]) \mid 1 \le t \le s\} + \gamma(-,g[j]) \\ \min\{\delta(F[l(i_t)..l(k)-1],G[1..l(j)-1]) \mid 1 \le t \le s\} \\ \qquad + \delta(F[l(k)..k],G[l(j)..j]) \end{cases} \\ &= \min\{\delta(F[l(i_t)..k],G[1..j]) \mid 1 \le t \le s\} \\ &= \Delta(F[l(i_1)..k],G[1..j]). \end{aligned}$$

□

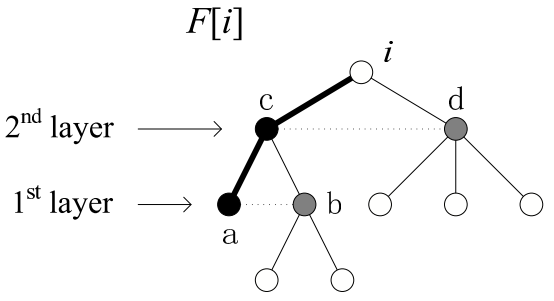


Fig. 3. The bold line is the leftmost path of $F[i]$. The black nodes (a,c) belong to $lp(i)$ and the black and gray nodes (a,b,c,d) belong to $layer(i)$.

With these two lemmas, we can calculate $\min\{\Delta(F[l(i_1)\cdots i_t], G[1..|G|) \mid 1 \leq t \leq d_i\}$ using dynamic programming. However, we have to do this for every node i of F. Because for each child subtree of $F[i]$ the calculation starts at i_1 instead of $l(i_1)$ and $\delta(F[l(i_1)..i_1], G[1..j])$ is needed in the calculation, the best way is to do the calculations for all the nodes on the path from a leaf to its nearest ancestor key root together. In this way, we do the computation layer by layer, see Figure 3. Lemma 3 and 4 extend Lemma 1 and 2 from a node to the leftmost path of a key root. Due to the page limitation, we omit the proofs.

We will need the following definitions: $lp(i)$: a set which contains the nodes on the leftmost path of $F[i]$ except the root i; $layer(i)$: a set which contains all of the sibling nodes of nodes in $lp(i)$ including $lp(i)$. In Figure 3, $lp(i) = \{a, c\}$ and $layer(i) = \{a, b, c, d\}$. With these definitions, we have the following two lemmas. In Lemma 4, for convenience, $forestdist(F[l(i)..i_1], G[1..j_1])$ represents $\delta(F[l(i)..i_1], G[1..j_1])$ and $treedist(i_1, j_1)$ represents $\delta(F[l(i_1)..i_1], G[l(j_1)..j_1])$.

Lemma 3. *Let i be a key root of F, $l(i) \leq i_1 < i$ and $1 \leq j_1 \leq |G|$, then*

$$\Delta(\emptyset, \emptyset) = 0;$$
$$\Delta(F[l(i)..i_1], \emptyset) = \begin{cases} 0 & \text{if } i_1 \in layer(i) \\ \Delta(F[l(i)..i_1 - 1], \emptyset) + \gamma(f[i_1], -); & \text{if } i_1 \notin layer(i) \end{cases}$$
$$\Delta(F[l(i)..i_1], G[1..j_1]) = \min \begin{cases} forestdist(F[l(i)..i_1], G[1..j_1]) \\ \delta(\emptyset, G[1..j_1]). \end{cases} \text{ if } i_1 \in lp(i)$$

Lemma 4. *Let i be a key root of F, $l(i) \leq i_1 < i$, $i_1 \notin lp(i)$, and $1 \leq j_1 \leq |G|$, then*

$$\begin{aligned} &\Delta(F[l(i)..i_1], G[1..j_1]) \\ &= \min \begin{cases} \Delta(F[l(i)..i_1 - 1], G[1..j_1]) + \gamma(f[i_1], -), \\ \Delta(F[l(i)..i_1], G[1..j_1 - 1]) + \gamma(-, g[j_1]), \\ \Delta(F[l(i)..l(i_1) - 1], G[1..l(j_1) - 1]) + treedist(i_1, j_1). \end{cases} \end{aligned}$$

Our algorithm is a dynamic programming algorithm. In the first stage of our algorithm, we call forest edit distance algorithm [15,2] for F and G to get $treedist(i, j)$ needed in Lemma 4. In the second stage, the key roots of F are sorted in an increasing order and put in an array K_F. And for any key root k of F, we first call forest edit distance algorithm of [15] for $F[k]$ and G to get $forestdist(,)$ needed in Lemma 3 and then call the procedure for $\Delta(,)$ computation for $F[k]$ and G. We are now ready to give our algorithm for finding a most similar closed subforest of F to G:

Theorem 1. *Our algorithm correctly computes the cost of an optimal solution.*

Proof. Because of step 1 in Algorithm 1, all the $treedist(,)$ used in step 7 in Procedure $Delta(F[i], G)$ are available. Because of step 4 in Algorithm 1, all the $forestdist(,)$ used in step 7 in Procedure $Delta(F[i], G)$ are available. □

Input: A target forest F and a pattern forest G.
Output: $\min\{\Delta(F[l(x_1)..x_2], G) \mid x_1$ is x_2's leftmost sibling$\}$.
Algorithm:
1 Call $TreeDistance(F, G)$ according to [15] or [2];
2 **for** $i' := 1$ *to* $|K_F|$ **do**
3 $i := K_F[i']$;
4 Call $ForestDistance(F[i], G)$ according to [15];
5 Call Procedure $Delta(F[i], G)$;
6 **end**

Algorithm 1. Finding a most similar closed subforest of F to G

Procedure $Delta(F[i], G)$:
1 $\Delta(\emptyset, \emptyset) = 0$;
2 **for** $i_1 := l(i)$ *to* $i - 1$ **do**
3 Compute $\Delta(F[l(i)..i_1], \emptyset)$ according to Lemma 3.
4 **end**
5 **for** $i_1 := l(i)$ *to* $i - 1$ **do**
6 **for** $j_1 := 1$ *to* $|G|$ **do**
7 Compute $\Delta(F[l(i)..i_1], G[1..j_1])$ according to Lemma 3 and Lemma 4.
8 **end**
9 **end**

Theorem 2. *Our algorithm can be implemented to run in* $O(|F| \cdot |G| \cdot min\{D_F, L_F\} \cdot min\{D_G, L_G\})$ *time and* $O(|F| \cdot |G|)$ *space.*

Proof. The time and space complexity of the computation for the edit distance of all subtree pairs of F and G is $O(|F| \cdot |G| \cdot \min\{D_F, L_F\} \cdot \min\{D_G, L_G\})$ and $O(|F| \cdot |G|)$ due to [15]. For one key root i, the time and space for

$ForestDistance(F[i], G)$ and $Delta(F[i], G)$ are the same: $O(|F[i]| \cdot |G|)$. Therefore the total time and space for all key roots are $O(|G||F| \cdot \min\{D_F, L_F\})$ and $O(|G||F|)$ due to Lamma 7 in [15]. Hence the time and space complexity of our algorithm are $O(|F| \cdot |G| \cdot \min\{D_F, L_F\} \cdot \min\{D_G, L_G\})$ and $O(|F| \cdot |G|)$ respectively.

If we use the algorithm [2] to compute the edit distance, the time complexity is $O(|F| \cdot |G|^2 \cdot (1 + \log(|F|/|G|)))$ and the total time complexity is $O(|F| \cdot |G| \cdot (|G| \cdot (1 + \log \frac{|F|}{|G|}) + \min\{D_F, L_F\}))$. □

3.2 An Algorithm for Finding a Most Similar Closed Substructure

In this section we consider the problem of finding a most similar closed substructure. Recall that, for a given forest F, a subtree of F is one of $F[i]$ where $1 \leq i \leq |F|$ and a subforest of F is an ordered sequence of subtrees of F.

Giving a target forest F and a pattern forest G, the forest removing distance from F to G, $\delta_r(F, G)$, is defined as the following where $subf(F)$ is the set of subforests of F and $F \setminus f$ represents the forest resulting from the deletion of subforest f from F.

$$\delta_r(F, G) = \min_{f \in subf(F)} \{\delta(F \setminus f, G)\}$$

Zhang and Shasha's algorithm [15] for approximate tree pattern matching with removing solves this problem. This can also be solved using the technique of Demaine *et al.* [2].

We again consider a node i in forest F and let the degree of node i be d_i and its children be $i_1, i_2, \ldots, i_{d_i}$. Let $k \in F[i_s]$ where $1 \leq s \leq d_i$, we now define another removing distance $\delta_R(F[l(i_1)..k], G[1..j])$ as follows where $subf(F, node\_set)$ is the set of subforests of F such that nodes in $node\_set$ are not in any of the subforests.

$$\delta_R(F[l(i_1)..k], G[1..j]) = \min_{f \in subf(F[l(i_1)..k], \{i_1, \ldots i_{d_i}\})} \delta(F[l(i_1)..k] \setminus f, G[1..j])$$

From this definition and the algorithm in [15], we have the following formula for $\delta_R(F[l(i_t)..k], G[1..j])$, where $1 \leq t \leq s$.

$$\begin{aligned}
&\delta_R(F[l(i_t)..k], G[1..j]) = \\
&\min \begin{cases}
\delta_R(F[l(i_t)..l(k)-1], G[1..j]), & \text{if } k \notin \{i_1, i_2, \ldots, i_{d_i}\} \\
\delta_R(F[l(i_t)..k-1], G[1..j]) + \gamma(f[k], -), \\
\delta_R(F[l(i_t)..k], G[1..j-1]) + \gamma(-, g[j]), \\
\delta_R(F[l(i_t)..l(k)-1], G[1..l(j)-1]) + \gamma(f[k], g[j]) \\
\quad + \delta_r(F[l(k)..k-1], G[l(j)..j-1]).
\end{cases}
\end{aligned}$$

We can now define $\Psi(F[l(i_1)..k], G[1..j])$ for $F[l(i_1)..k]$ and $G[1..j]$ using $\delta_R(,)$ for closed substructures. This is exactly the same way as we define $\Delta(F[l(i_1)..k], G[1..j])$ using $\delta(,)$ for closed subforests.

If $k \in \{i_1, i_2, \ldots, i_{d_i}\}$, then $\Psi(F[l(i_1)..k], G[1..j])$ is defined as $\min\{\delta_R(F[l(i_t)..i_s],\ G[1..j]) \mid 1 \le t \le s+1\}$.

In particular, if $s = 1$, $\Psi(F[l(i_1)..i_1], G[1..j])$ is $\min\{\delta_R(\emptyset, G[1..j]),\ \delta_R(F[l(i_1)..i_1], G[1..j])\} = \delta_r(F[l(i_1)..i_1], G[1..j])$ which can be obtained directly using the forest removing distance algorithm.

If $k \notin \{i_1, i_2, \ldots, i_{d_i}\}$, then $\Psi(F[l(i_1)..k], G[1..j])$ is defined as $\min\{\delta_R(F[l(i_t)..k],\ G[1..j]) \mid 1 \le t \le s\}$.

For the calculation of $\Psi(F[l(i_1)..k], G[1..j])$, we have the following two lemmas. The proofs are similar to Lemma 1 and Lemma 2.

Lemma 5. *Let i, F and G be defined as the above, $i_1 \le k \le i_{d_i}$ and $1 \le j \le |G|$, then*

$$\begin{aligned}&\Psi(\emptyset, \emptyset) = 0;\\ &\Psi(F[l(i_1)..k], \emptyset) = 0;\\ &\Psi(F[l(i_1)..i_1], G[1..j]) = \delta_r(F[l(i_1)..i_1], G[1..j]).\end{aligned}$$

Lemma 6. *Let i, F and G be defined as the above, $i_1 < k \le i_{d_i}$ and $1 \le j \le |G|$, then*

$$\begin{aligned}&\Psi(F[l(i_1)..k], G[1..j])\\ &= \min\begin{cases}\Psi(F[l(i_1)..l(k)-1], G[1..j]), & \text{if } k \notin \{i_2, \ldots, i_{d_i}\}\\ \Psi(F[l(i_1)..k-1], G[1..j]) + \gamma(f[k], -),\\ \Psi(F[l(i_1)..k], G[1..j-1]) + \gamma(-, g[j]),\\ \Psi(F[l(i_1)..l(k)-1], G[1..l(j)-1]) + \gamma(f[k], g[j])\\ \quad + \delta_r(F[l(k)..k-1], G[l(j)..j-1]).\end{cases}\end{aligned}$$

Lemma 7 and 8 extend Lemma 5 and 6 from a node to the leftmost path of a key root. Due to the page limitation, we omit the proofs.

Lemma 7. *Let i be a key root of F, $l(i) \le i_1 < i$ and $1 \le j_1 \le |G|$, then*

$$\begin{aligned}&\Psi(\emptyset, \emptyset) = 0;\\ &\Psi(F[l(i)..i_1], \emptyset) = 0;\\ &\Psi(F[l(i)..i_1], G[1..j_1]) = \delta_r(F[l(i)..i_1], G[1..j_1]). \quad \text{if } i_1 \in lp(i)\end{aligned}$$

Lemma 8. *Let i be a key root of F, $l(i) < i_1 < i$, $i_1 \notin lp(i)$, and $1 \le j_1 \le |G|$, then*

$$\begin{aligned}&\Psi(F[l(i)..i_1], G[1..j_1]) =\\ &\min\begin{cases}\Psi(F[l(i)..l(i_1)-1], G[1..j_1]), & \text{if } i_1 \notin layer(i)\\ \Psi(F[l(i)..i_1-1], G[1..j_1]) + \gamma(f[i_1], -),\\ \Psi(F[l(i)..i_1], G[1..j_1-1]) + \gamma(-, g[j_1]),\\ \Psi(F[l(i)..l(i_1)-1], G[1..l(j_1)-1]) + \gamma(f[i_1], g[j_1])\\ \quad + \delta_r(F[l(i_1)..i_1-1], G[l(j_1)..j_1-1]).\end{cases}\end{aligned}$$

We can now show our algorithm for closed substructures.

Input: A target forest F and a pattern forest G.
Output: $\min\{\Psi(F[l(x_1)..x_2], G) \mid x_1$ is x_2's leftmost sibling$\}$.
Algorithm:
1 Call $Tree\_RemoveDistance(F, G)$ according to [15];
2 **for** i':=1 *to* $|K(F)|$ **do**
3 $\quad i := K(F)[i']$;
4 $\quad$ Call $Forest\_RemoveDistance(F[i], |G|)$ according to [15];
5 $\quad$ Call Procedure $Psi(F[i], G)$;
6 **end**

Algorithm 2. Finding most similar closed substructure of F to G

Procedure $Psi(F[i], G)$:
1 $\Psi(\emptyset, \emptyset) = 0$;
2 **for** $i_1 := l(i)$ *to* $i - 1$ **do**
3 $\quad \Psi(F[l(i)..i_1], \emptyset) = 0$;
4 **end**
5 **for** $i_1 := l(i)$ *to* $i - 1$ **do**
6 $\quad$ **for** $j_1 := 1$ *to* $|G|$ **do**
7 $\quad\quad$ Compute $\Psi(F[l(i)..i_1], G[1..j_1])$ according to Lemma 7 and Lemma 8.
8 $\quad$ **end**
9 **end**

4 Conclusion

We have presented two algorithms for the forest pattern matching problem for two types of sub-forest. Our first algorithm for finding a most similar closed subforest is better than that of [5]. Our second algorithm for finding a most similar closed substructure can be used to search for local forest patterns.

When the input are two sequences represented as forests, both our algorithms reduce to the sequence approximate pattern matching algorithm [9]. When the input are two sequences represented as linear trees, our second algorithm reduces to the sequence approximate pattern matching algorithm [9].

References

1. Backofen, R., Will, S.: Local Sequence-structure Motifs in RNA. Journal of Bioinformatics and Computational Biology 2(4), 681–698 (2004)
2. Demaine, E.D., Mozes, S., Rossman, B., Weimann, O.: An optimal decomposition algorithm for tree edit distance. In: Arge, L., Cachin, C., Jurdziński, T., Tarlecki, A. (eds.) ICALP 2007. LNCS, vol. 4596, pp. 146–157. Springer, Heidelberg (2007)
3. Höchsmann, M., Töller, T., Giegerich, R., Kurtz, S.: Local similarity in RNA secondary structures. In: Proceedings of the IEEE Computational Systems Bioinformatics Conference, pp. 159–168 (2003)
4. Jansson, J., Hieu, N.T., Sung, W.-K.: Local gapped subforest alignment and its application in finding RNA structural motifs. In: Fleischer, R., Trippen, G. (eds.) ISAAC 2004. LNCS, vol. 3341, pp. 569–580. Springer, Heidelberg (2004)

5. Jansson, J., Peng, Z.: Algorithms for Finding a Most Similar Subforest. In: Lewenstein, M., Valiente, G. (eds.) CPM 2006. LNCS, vol. 4009, pp. 377–388. Springer, Heidelberg (2006)
6. Jiang, T., Wang, L., Zhang, K.: Alignment of trees - an alternative to tree edit. Theoretical Computer Science 143, 137–148 (1995)
7. 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)
8. Motifs database, `http://subviral.med.uottawa.ca/cgi-bin/motifs.cgi`
9. Sellers, P.H.: The theory and computation of evolutionary distances: pattern recognition. Journal of Algorithms 1(4), 359–373 (1980)
10. Shapiro, B.A., Zhang, K.: Comparing multiple RNA secondary structures using tree comparisons. Computer Applications in the Biosciences 6(4), 309–318 (1990)
11. Tai, K.-C.: The tree-to-tree correction problem. Journal of the Association for Computing Machinery (JACM) 26(3), 422–433 (1979)
12. Touzet, H.: A linear time edit distance algorithm for similar ordered trees. In: Apostolico, A., Crochemore, M., Park, K. (eds.) CPM 2005. LNCS, vol. 3537, pp. 334–345. Springer, Heidelberg (2005)
13. Ukkonen, E.: Algorithms for approximate string matching. Information and Control 64(1–3), 100–118 (1985)
14. Zhang, K.: Computing similarity between RNA secondary structures. In: Proceedings of IEEE International Joint Symposia on Intelligence and Systems, Rockville, Maryland, May 1998, pp. 126–132 (1998)
15. 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)

Affine Image Matching Is Uniform TC^0-Complete

Christian Hundt

Institut für Informatik, Universität Rostock, Germany
christian.hundt@uni-rostock.de

Abstract. Affine image matching is a computational problem to determine for two given images A and B how much an affine transformated A can resemble B. The research in combinatorial pattern matching led to a polynomial time algorithm which solves this problem by a sophisticated search in the set $\mathcal{D}(A)$ of all affine transformations of A. This paper shows that polynomial time is not the lowest complexity class containing this problem by providing its TC^0-completeness. This result means not only that there are extremely efficient parallel solutions but also reveals further insight into the structural properties of image matching. The completeness in TC^0 relates affine image matching to a number of most basic problems in computer science, like integer multiplication and division.

Keywords: digital image matching, combinatorial pattern matching, design and analysis of parallel algorithms.

1 Introduction

The affine image matching problem (AIMP, for short) is to determine for two given images A and B how much an affine transformation of A can resemble B. Affine image matching (AIM) has a wide range of applications in various image processing settings, e.g., in computer vision [16], medical imaging [5,18,19], pattern recognition, digital watermarking [7], etc.

Recently, discretization techniques developed in the combinatorial pattern matching research (CPM, for short) have been used successfully for AIM. Apart from algorithmic achievements, this led to improved techniques for the analysis of the problem. Essentially, all algorithms developed in CPM for computing a best match $f(A)$ with B share the same plane idea, to perform exhaustive search of the entire set $\mathcal{D}(A)$, which contains all affine transformations of A. Surprisingly, the fastest known methods which determine the provably best affine image match come from this simple approach. In fact, the main challenge of computing $\mathcal{D}(A)$, is to find a discretization of the set $\mathcal{F}$ of all affine transformations. A convenient starting point for the research in this direction is given by the discretization techniques developed in CPM, although the problem in the focus of CPM consists in locating an *exact* match of an affine transformation of A in B, rather than on computing the *best* one like in AIM. See e.g. [17,11,10,1,4,3,2].

In [13,14] affine transformations are characterized by six real parameters and, based on this, a new generic discretization of $\mathcal{F}$ is developed which is basically

A. Amir and L. Parida (Eds.): CPM 2010, LNCS 6129, pp. 13–25, 2010.

a partition of the parameter space $\mathbb{R}^6$ into subspaces $\varphi_1, \ldots, \varphi_{\tau(n)}$, where $\tau(n)$ depends on the size $n \times n$ of image A. Every subspace φ_i represents one possible transformation of A and consequently the cardinality of $\mathcal{D}(A)$ is shown to be in $O(n^{18})$ by estimating an upper bound on the number $\tau(n)$ of subspaces. The discretization motivates an algorithm that first constructs a data structure $\mathcal{I}_n$ representing the partition and then, to solve the AIMP, it searches all images in $\mathcal{D}(A)$ by traversing $\mathcal{I}_n$. Its running time is linear in $\tau(n)$ and thus, in $O(n^{18})$ for images A and B of size $n \times n$.

However, the exact time complexity remains unknown. It is also an open question whether the decision version of the AIMP is included in a complexity class that is "easier" than P – the class of problems decidable in polynomial time. Particularly, it is open whether the problem belongs to low complexity classes of the hierarchy inside P:

$$\mathrm{AC}^0 \subset \mathrm{TC}^0 \subseteq \mathrm{NC}^1 \subseteq \mathrm{L} \subseteq \mathrm{P}.$$

Every class in the hierarchy implies a structural computational advantage against the hardness in P. This paper continues the research on AIM in the combinatorial setting using the algebraic approach introduced in [13] and refined in [14] to give a new, surprisingly low complexity for AIM by showing that the affine image matching problem is TC^0-complete. TC^0 is a very natural complexity class because it exactly expresses the complexity of a variety of basic problems in computation, such as integer addition, multiplication, comparison and division.

The containment of the AIMP in $\mathrm{TC}^0 \subseteq \mathrm{NC}^1$ means first that AIM can be solved in logarithmic parallel time on multi-processor systems which have bounded fan-in architecture. However, theoretically it can even be solved in constant time if the processors are assumed to have unbounded fan-in. Secondly, since $\mathrm{TC}^0 \subseteq \mathrm{L}$ AIM can also be solved on deterministic sequential machines using only a logarithmic amount of memory. Finally, the completeness of the AIMP in TC^0 means that there is no polynomially sized, uniformly shaped family of Boolean formulas expressing AIM since this captures the computational power of $\mathrm{AC}^0 \neq \mathrm{TC}^0$.

Anyway, the new results have no immediate impact on practical settings of AIM. In fact, they have to be seen more as an ambition to uncover the structural properties of AIM and related problems. Particularly, the novel TC^0 approach to AIM is based on a characterization of the parameter space partition $\varphi_1, \ldots, \varphi_{\tau(n)}$ for affine transformations. Every subspace φ_i is shown to exhibit a positive volume such that an algorithm can simply sample a certain subregion of $\mathbb{R}^6$ to hit every element of the partition. Thus $\mathcal{D}(A)$ can be computed without the data structure $\mathcal{I}_n$ that implicitely represents the space partition. Interestingly, this is not an hereditary property. The parameter space partition of linear transformations, i.e., the subset of $\mathcal{F}$ without translations, contains subspaces with zero volume [14]. This means that, although linear image matching is weaker than AIM, the low-complexity approach of this paper cannot be applied to this problem in a straight forward manner. The author leaves the estimation of this problem's complexity as an open challenge.

This paper presents results which heavily build on previous work [13,14]. After a short presentation of technical preliminaries Section 3 briefly provides

the basics of the AIM approach introduced in [13,14] which are necessary to understand the new results of this paper. Then Section 4 provides the new TC^0 approach to AIM and next Section 5 proves that the AIMP belongs to the hardest problems of TC^0. Finally, the paper concludes by drawing a wider picture of the finding's impact. All proofs are removed due to space limitations.

2 Technical Preliminaries

Digital Images and their Affine Transformations. Through the whole paper, an image is a two-dimensional array of pixels, i.e., of unit squares partitioning a certain square area of the real plane $\mathbb{R}^2$. The pixels of an image A are indexed over a set $\mathcal{N} = \{(i,j) \mid -n \leq i,j \leq n\}$, where n is called the size of A. The geometric center point of the pixel with index (i,j) can be found at coordinates (i,j). Each pixel (i,j) has a color $A\langle i,j\rangle$ that is an element from a finite set $\Sigma = \{0,1,\ldots,\sigma\}$ of color values. To simplify the dealing with A's borders let $A\langle i,j\rangle = 0$ if $(i,j) \notin \mathcal{N}$. The distortion between two given images A and B of size n is measured by $\Delta(A,B) = \sum_{(i,j)\in\mathcal{N}} \delta(A\langle i,j\rangle, B\langle i,j\rangle)$ where $\delta : \Sigma \times \Sigma \to \mathbb{N}$ is a function charging color mismatches, for example, $\delta(c_1,c_2) = |c_1 - c_2|$.

The set $\mathcal{F}$ of affine transformations contains exactly all injective functions $f : \mathbb{R}^2 \to \mathbb{R}^2$ which can be described by

$$f(x,y) = \left(\begin{smallmatrix} a_1 & a_2 \\ a_4 & a_5 \end{smallmatrix}\right) \cdot \left(\begin{smallmatrix} x \\ y \end{smallmatrix}\right) + \left(\begin{smallmatrix} a_3 \\ a_6 \end{smallmatrix}\right) \tag{1}$$

for some constants $a_1,\ldots,a_6 \in \mathbb{R}$, with the additional property of $a_1a_5 \neq a_2a_4$.

Applying an affine transformation f to A gives a transformed image $f(A)$ of size n. To define a color value for any pixel (i,j) in $f(A)$, let f^{-1} be the inverse function of f. Notice that f^{-1} is always an affine transformation, too. Then define the color value $f(A)\langle i,j\rangle$ as the color $A\langle I,J\rangle$ of the pixel $(I,J) = [f^{-1}(i,j)]$, where $[(x,y)] := ([x],[y])$ denotes rounding both components of a vector $(x,y) \in \mathbb{R}^2$. Hence, determining $f(A)\langle i,j\rangle$ means to choose the pixel (I,J) of A which geometrically contains the point $f^{-1}(i,j)$ in its square area. This setting models *nearest-neighbor* interpolation, commonly used in image processing. Now, any image A defines the set $\mathcal{D}(A) = \{f(A) \mid f \in \mathcal{F}\}$ that contains all possible affine transformations of A.

Based on this, the following defines the affine image matching problem:

For given images A and B of size n find the minimal distortion $\Delta(f(A),B)$ over all transformations $f \in \mathcal{F}$.

For the analysis of complexity aspects consider the decision variant of this problem which asks if there is a transformation $f \in \mathcal{F}$ which yields $\Delta(f(A),B) \leq t$ for some given threshold $t \in \mathbb{N}$.

Circuit Complexity. This paper discusses the complexity of certain functions $f : \{0,1\}^* \to \{0,1\}^*$ mapping binary strings to binary strings. Let $|s|$ be the length of a binary string $s \in \{0,1\}^*$ and for all $i \in \{0\ldots,|s|-1\}$ let $s\langle i\rangle$ denote the ith character of s. Moreover, let 1^n be the string of n sequent characters 1 and for all strings s and s' let $s|s'$ be their concatenation.

Circuits C can be imagined as directed acyclic graphs were vertices, also called gates, compute Boolean functions. Gates gain input truth values from predecessor gates and distribute computation results to all their successor gates. If C has n sources and m sinks, then it computes a function $f : \{0,1\}^n \to \{0,1\}^m$, i.e., C computes for every input string of length n an output string of length m. This makes circuits weaker than other computational models, which can compute functions $f : \{0,1\}^* \to \{0,1\}^*$. Consequently one considers families $\mathcal{C} = \{C_1, C_2, \ldots\}$ of circuits to compute f for every input length n with an individual circuit C_n. On the other hand, such families can be surprisingly powerful because they may not necessarily be finitely describable in the traditional sense. A usual workaround is a uniformity constraint which demands that every circuit C_n of $\mathcal{C}$ can be described by a Turing machine $M_{\mathcal{C}}$ with resource bounds related to n. Usually $M_{\mathcal{C}}$ is chosen much weaker than the computational power of $\mathcal{C}$ to avoid the obscuration of $\mathcal{C}$'s complexity. This paper considers only DLOGTIME-uniform families $\mathcal{C}$ where $M_{\mathcal{C}}$ has to verify in $O(\log n)$ time whether C_n fulfills a given structural property like, e.g., "*Gate i computes the $\wedge$-function*" or "*Gate i is a predecessor of gate j*".

The class DLOGTIME-uniform FAC^0 contains all functions $f : \{0,1\}^* \to \{0,1\}^*$ which can be computed by constant-depth, polynomial-size families $\mathcal{C}$ of DLOGTIME-uniform circuits, i.e., where (a) every gate computes a function "$\wedge$", "$\vee$" or "$\neg$", (b) all circuits C_n can be verified by a Turing machine $M_{\mathcal{C}}$ that runs in $O(\log n)$-time (c) the number of gates in C_n grows only polynomially in n and (d) regardless of n, the length of any path in C_n from input to output is not longer than a constant. For convenience denote this class also by U_D-FAC^0. A prominent member of U_D-FAC^0 is the addition function of two integer numbers.

If the gates can also compute threshold-functions T_k, a generalization of "$\wedge$" and "$\vee$" which is true if at least k inputs are true, then the generated function class is called DLOGTIME-uniform FTC^0 (U_D-FTC^0), a class that contains a big variety of integer arithmetic functions.

A decision problem is a set $\Pi \subseteq \{0,1\}^*$, i.e., a set of strings. By U_D-AC^0 denote the class of all decision problems which can be decided by a function $f \in U_D$-FAC^0, i.e., $f : \{0,1\}^* \to \{0,1\}$ is a function with $f(s) = 1 \Leftrightarrow s \in \Pi$. Accordingly, U_D-TC^0 is the class of decision problems decidable by a function in U_D-FTC^0.

This paper uses special decision problems Π_f for any function $f : \{0,1\}^* \to \{0,1\}^*$. The set Π_f contains all binary strings s which encode pairs $(i, s') \in \mathbb{N} \times \{0,1\}^*$ using a unary encoding for integer i and a binary encoding for s' such that $(i, s') \in \Pi_f$ if and only if $f(s')\langle i \rangle = 1$, i.e., if the ith character of $f(s')$ is a 1. Clearly, f is in U_D-FAC^0 if the output length of f is bounded polynomially in the input length and Π_f is in U_D-AC^0. A circuit family $\mathcal{C}$ deciding Π_f can be used to compute also f simply by spending one circuit of $\mathcal{C}$ for every output bit. The same holds for functions in U_D-FTC^0.

By definition, U_D-(F)AC^0 is a subset of U_D-(F)TC^0 and thus, U_D-FAC^0-reductions are suited well to define completeness in the class U_D-TC^0. A problem Π is U_D-TC^0-complete if Π belongs to U_D-TC^0 and if for all $\Pi' \in U_D$-TC^0 there

is a function r_{Π} in U_D-FAC^0 such that for all $s \in \{0,1\}^*$ it is true $s \in \Pi' \Leftrightarrow r_{\Pi}(s) \in \Pi$. Clearly, since U_D-FAC^0-reductions are transitive, it is also sufficient for U_D-TC^0-completeness to find one other U_D-TC^0-complete problem Π' and then provide a function r in U_D-FAC^0 such that for all $s \in \{0,1\}^*$ it is true $s \in \Pi' \Leftrightarrow r(s) \in \Pi$. A canonical U_D-TC^0-complete set is MAJ, containing strings over $\{0,1\}^*$ with a majority of 1-characters [6].

For convenience the uniformity statement U_D is mostly omitted in the rest of the paper since all considered circuit families apply DLOGTIME-uniformity. Due to space limitations this section cannot go into further details of this rich theory and the author refers the reader to the text book [21].

First Order Logic. First order formulas are an important concept from logic. A comprehensive introduction to first order logic and in particular the connection to circuit families is given in [21].

In this paper a first order formula F is build recursively over a unary predicate $s(\cdot)$ and two binary predicates $bit(\cdot,\cdot)$ and $<(\cdot,\cdot)$ by the standard use of "$\wedge$", "$\vee$", "$\neg$", "$\forall$" and "$\exists$". Without loss of generality F is of the form $F = Q_1 v_1 \dots Q_m v_m F'$ where $Q_1, \dots, Q_m$ are quantifiers, "$\forall$" or "$\exists$", for the variables $v_1, \dots, v_m$ and F' is a quantifier free formula. The variables $v_1, \dots, v_m$ are called bounded and every other variable in F' is free. If there are no free variables then F is called a sentence.

The assertion of a formula F is either true or false, which is defined over the recursive construction of F and relative to (1) a universe, that is a finite subset $\{0, \dots, n-1\}$ of $\mathbb{N}$, (2) a specification of $s(\cdot)$ and (3) an assignment of values for free variables. The meaning of the binary predicates is fixed, thus, $bit(a,i)$ is true if and only if the ith bit in the n-bit binary representation of a is one, and $<(a,b)$ is true if and only if $a < b$.

This paper applies first order logic to describe sets of strings $\Pi \subseteq \{0,1\}^*$, i.e., decision problems. Particularly, any string $s \in \{0,1\}^*$ defines a universe $\{0, \dots, |s|-1\}$ and a specification of $s(\cdot)$ by giving for all $i \in \{0, \dots, |s|-1\}$ that the predicate $s(i)$ is true if and only if $s\langle i \rangle = 1$. Consequently, a string s alone determines the truth value of a sentence F because it has no free variables. Then a string s is said to model F, which is denoted by $s \models F$, if s satisfies F. Thereby a sentence F describes a set $\Pi_F = \{s \in \{0,1\}^* \mid s \models F\}$. For example $F = \exists v\ s(v)$ gives the set of strings which contain at least one character 1.

If F has free variables $v_1, \dots, v_m$, then a string s alone is not enough to determine the truth value. However, in this case the formula $F[v_1 \leftarrow i_1, \dots, v_m \leftarrow i_m]$, where the free variables are assigned certain values $i_1, \dots, i_m$ from the universe, defines a proper truth value. This paper applies the concept of free variables in terms of a modular design principle. The variables $v_1, \dots, v_m$ can be understood as parameters of F which influence the formula's assertion. This means that F can be applied as a subformula in a sentence F' which uses $v_1, \dots, v_m$ to pass auxiliary arguments $i_1, \dots, i_m$ to F. Such "subformula-calls" are denoted by $F[i_1, \dots, i_m]$.

The set of all problems Π which can be expressed by a first order sentence F, i.e., for which $\Pi = \Pi_F$, is denoted by FO. It turns out that $FO = U_D\text{-AC}^0$. Consequently integer addition is first-order-expressible and therefore this paper utilizes the subformula $ADD[x_1, x_2, y]$ which is satisfied if and only if x_1, x_2 and y are assigned values satisfying $x_1 + x_2 = y$.

TC^0 being a generalization of AC^0 implies that a characterization of TC^0 in terms of first order logic needs a language extension. Therefore consider beside "$\forall$" and "$\exists$" the additional majority quantifier "M", which is defined as follows: The sentence $F = Mv\ F'$ is true for given strings s if and only if the formulas $F'[v \leftarrow i]$ are true for a majority of assignments of $i \in \{0, \ldots, |s| - 1\}$ to the free variable v. Then $U_D\text{-TC}^0 = FO[M]$, the set of problems expressible by first order sentences with additional quantifier "M".

In some cases it is difficult to express certain relations in FO or $FO[M]$-sentences just because the values of variables are restricted to $\{0, \ldots, |s| - 1\}$. However, one can simply assume that there are long variables v which are able to take values in the range $\{-|s|^k - 1, \ldots, |s|^k - 1\}$ for some arbitrary constant k. The value of v can simply be represented in $k+1$ ordinary variables $v_0, \ldots, v_{k-1}$ and sgn by $v = (-1)^{sgn} \cdot \sum_{i=0}^{k-1} |s|^i \cdot v_i$. A sentence F using a long variable v realizes the quantification and the predicates $<(\cdot,\cdot)$ and $bit(\cdot,\cdot)$ over v by reducing them to their ordinary counterparts.

Beside long variables first order logic can be extended also with a $\leq(\cdot,\cdot)$ predicate because it easily reduces to $<(\cdot,\cdot)$. This paper applies these predicates in infix notation.

3 Previous Results

Previous work [13,14] presented a new algorithmic approach to solve affine image matching in linear time with respect to the cardinality $|\mathcal{D}(A)|$. Moreover, it provided an upper bound of $O(n^{18})$ for this cardinality which means that AIM can be solved in polynomial time. This section briefly discusses some basics of this approach which are used in this paper.

By equation (1) in the previous section, all transformations in $\mathcal{F}$ can be characterized by the six parameters a_1 to a_6. Hence, each affine transformation f can be described by a point $(a_1, \ldots, a_6)^T$ in the six-dimensional parameter space $\mathbb{R}^6$. Reversely, every such point in $\mathbb{R}^6$ which fulfills $a_1 a_5 \neq a_2 a_4$ characterizes an affine transformation. Now, a discrete characterization of $\mathcal{F}$ can be obtained by a subdivision of the parameter space $\mathbb{R}^6$ into a finite number of subspaces $\varphi_1, \ldots, \varphi_{\tau(n)}$ with the following property: Any pair of transformations $f, f' \in \mathcal{F}$ gives the same transformation $f(A) = f'(A)$ of an image A of size n if their inverses f^{-1} and f'^{-1} are represented by points $(a_1, \ldots, a_6)^T$, resp. $(a'_1, \ldots, a'_6)^T$, contained in the same subspace φ_i for some $i \in \{1, \ldots, \tau(n)\}$. This means that each of the $\tau(n)$ subspaces represents one transformed image in $\mathcal{D}(A)$.

The principle of the polynomial time algorithm is searching the whole set $\mathcal{D}(A)$ which is a common practice in the CPM. Using the discrete characterization of $\mathcal{F}$ the algorithm traverses all the subspaces φ_1 to $\varphi_{\tau(n)}$ of the parameter

space. With each subspace it finds one of the possible transformed images A' in $\mathcal{D}(A)$. Subsequently, the distortion between such images A' and B is evaluated to eventually find the best match.

For images of size n the subdivision of the parameter space into the spaces φ_1 to $\varphi_{\tau(n)}$ is determined by the following set H_n of functions $\mathbb{R}^6 \to \mathbb{R}$:

$$H_n = \{I_{ijk}(a_1,\ldots,a_6) = ia_1 + ja_2 + a_3 - (k-0.5) \mid (i,j) \in \mathcal{N}, k \in \{-n,\ldots,n+1\}\}$$
$$\cup\ \{J_{ijk}(a_1,\ldots,a_6) = ia_4 + ja_5 + a_6 - (k-0.5) \mid (i,j) \in \mathcal{N}, k \in \{-n,\ldots,n+1\}\}$$

Hence, $H_n = \{\ell_1,\ldots,\ell_{r(n)}\}$ is a set of $r(n) = (2n+1)^2(2n+2)$ linear functions where every ℓ_w, either $\ell_w = I_{ijk}$ or $\ell_w = J_{ijk}$ for some $(i,j) \in \mathcal{N}$ and $k \in \{-n,\ldots,n+1\}$, describes the following two subspaces of $\mathbb{R}^6$:

$$h^+(\ell_w) = \{(a_1,\ldots,a_6)^T \in \mathbb{R}^6 \mid \ell_w(a_1,\ldots,a_6) \geq 0\},$$
$$h^-(\ell_w) = \{(a_1,\ldots,a_6)^T \in \mathbb{R}^6 \mid \ell_w(a_1,\ldots,a_6) < 0\}.$$

The meaning of the sets $h^+(\ell_w)$ and $h^-(\ell_w)$ can be understood as follows: All the points $(a_1,\ldots,a_6)^T$ in $h^+(I_{ijk})$ describe inverse affine transformations $f^{-1}(x,y) = \left(\begin{smallmatrix} a_1 & a_2 \\ a_4 & a_5 \end{smallmatrix}\right) \cdot \left(\begin{smallmatrix} x \\ y \end{smallmatrix}\right) + \left(\begin{smallmatrix} a_3 \\ a_6 \end{smallmatrix}\right)$ which have one thing in common: It is always true that $[f^{-1}(i,j)] = (I,J)$ with $I \geq k$. Accordingly, all points $(a_1,\ldots,a_6)^T$ in $h^-(I_{ijk})$ give transformations f^{-1} which uniquely fulfill $[f^{-1}(i,j)] = (I,J)$ with $I < k$. Finally, a similar property is true for the J-coordinate of $[f^{-1}(i,j)] = (I,J)$ depending on the situation of the point describing f^{-1} with respect to $h^+(J_{ijk})$ and $h^-(J_{ijk})$.

Now, the partition of the parameter space into the pieces φ_1 to $\varphi_{\tau(n)}$ is defined by the intersection of the subspaces $h^+(\ell)$ and $h^-(\ell)$ given by the lines ℓ in H_n. Particularly, for $H_n = \{\ell_1,\ldots,\ell_{r(n)}\}$ define

$$\mathcal{A}(H_n) = \left\{ \varphi \subseteq \mathbb{R}^6 \;\middle|\; \varphi = \bigcap_{w=1}^{r(n)} h^{s_w}(\ell_w) \text{ for some } s_1,\ldots,s_{r(n)} \in \{+,-\}, \varphi \neq \emptyset \right\}.$$

In literature the set $\mathcal{A}(H_n)$ is called the (hyperplane) arrangement given by H_n. For detailed information on such arrangements see [8]. In this paper the elements of $\mathcal{A}(H_n)$ are called faces.

The relation between $\mathcal{A}(H_n)$ and $\mathcal{D}(A)$ is the most important property formulated in [14]:

Theorem 1 ([14]). *For all n and every image A of size n there exists a surjective mapping*

$$\Gamma_n : \mathcal{A}(H_n) \to \mathcal{D}(A).$$

Thus, Theorem 1 reduces the enumeration of $\mathcal{D}(A)$, a set with no obvious structure, to the enumeration of $\mathcal{A}(H_n)$. In turn, the efficient enumeration of all faces in $\mathcal{A}(H_n)$ can be realized easily. The algorithm conveniently constructs a graph $\mathcal{I}_n$, which contains a node $v(\varphi)$ for each face $\varphi \in \mathcal{A}(H_n)$ and which encodes the incidence of faces by edges, i.e., two nodes $v(\varphi)$ and $v(\varphi')$ are connected by an edge if the faces φ and φ' are neighbors in $\mathbb{R}^6$. For a detailed description of

incidence graphs for arrangements and the complexity of computing them see [8] and [9]. The affine image matching algorithm proposed in [14] works as follows

The AIM Algorithm

1. Construct the incidence graph $\mathcal{I}_n$;
2. Perform *depth first searching* to traverse all nodes $v(\varphi)$ in $\mathcal{I}_n$;
3. For each enumerated face φ apply $\Gamma_n(\varphi)$ to compute $f(A)$;
4. Return the image $f(A)$ that induces the minimum distortion $\Delta(f(A), B)$.

This algorithm finds the best affine image match in $O(|\mathcal{A}(H_n)|)$ time plus the time needed to compute the incidence graph which is linear with respect to $|\mathcal{A}(H_n)|$, too. The following estimation bounds the algorithm's running time:

Theorem 2 ([14]). *The cardinality of $\mathcal{A}(H_n)$ is $O(n^{18})$. As a consequence AIM can be done in time bounded by $O(n^{18})$.*

The rest of this paper shows how to avoid the sequential manner of computation and introduces how to get a TC^0 circuit family to solve affine image matching. Moreover, it provides a simple FAC^0-reduction of the majority function to the affine image matching problem.

4 Membership in TC^0

Define $\Pi \subseteq \{0,1\}^*$ the set of strings $s = n|a|b|t$ which encode (1) a number $n \in \mathbb{N}$ in zero-terminated unary 1^n0, (2) two images A and B of size n by binary strings a and b each of $(2n+1)\lceil \log_2(\sigma+1) \rceil$ bits and (3) a number $t \in \mathbb{N}$ in binary representation such that the minimum of $\Delta(f(A), B)$ over all transformations f in $\mathcal{F}$ is at most t. Hence, the set Π is a concrete realization of the AIMP's decision version. This section develops an $FO[M]$-sentence F to express Π, i.e., such that $\Pi_F = \Pi$, which implies that the decision version of AIM is in TC^0. Subsequently it argues that also the optimization version is in FTC^0.

Basically the new $FO[M]$ approach to AIM is somehow a relaxation of the old one. To compute $\mathcal{D}(A)$ it is sufficient to find one point from every face in $\mathcal{A}(H_n)$ in order to describe a representative inverse affine transformation f^{-1}. By Theorem 1 one can find all images $f(A)$ in $\mathcal{D}(A)$ in this way. The graph $\mathcal{I}_n$ makes sure that every face in $\mathcal{A}(H_n)$ is processed only once. However, it may be possible to drop the computation of $\mathcal{I}_n$ if one does not insist on this exact processing of $\mathcal{A}(H_n)$.

Consequently, the $FO[M]$ approach works as follows: To find at least one point from every face in $\mathcal{A}(H_n)$ a sentence F can sample a hypercube region of $\mathbb{R}^6$ in such a way that avoids points $(a_1, \ldots, a_6)^T$ with the property $a_1a_5 = a_2a_4$. In this way all images $f(A)$ of $\mathcal{D}(A)$ can be computed in a parallel fashion. Then F expresses $\Delta(f(A), B)$ and subsequently the minimum over all $f \in \mathcal{F}$.

For this new technique consider $\mathcal{G}_n$, the grid of points

$$\begin{pmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \end{pmatrix} = 10^{-7}n^{-7} \cdot \begin{pmatrix} t_1{+}0.5 \\ t_2 \\ t_3 \\ t_4 \\ t_5{+}0.5 \\ t_6 \end{pmatrix}$$

where $t_1, \ldots, t_6$ are integers in the set $\{-10^{12}n^{13}, \ldots, 10^{12}n^{13}\}$. The central property of $\mathcal{G}_n$ applied in F is given in the following theorem:

Theorem 3. *1.* $|\mathcal{G}_n| \in O(n^{78})$.
2. Every point $p = (a_1, \ldots, a_6)^T \in \mathcal{G}_n$ *fulfills* $a_1a_5 \neq a_2a_4$.
3. For all faces φ *in* $\mathcal{A}(H_n)$ *there is a point* p *in the grid* $\mathcal{G}_n$ *such that* $p \in \varphi$.

The proof of the theorem is somewhat technical. However, it first shows that there is a hypercube that intersects all faces of $\mathcal{A}(H_n)$. Then it establishes a lower bound on the volume of all faces. Consequently, if the hypercube is sampled with points of adequately small distance, every face of $\mathcal{A}(H_n)$ gets a hit. Because all points $(a_1, \ldots, a_6)^T$ avoid the condition $a_1a_5 = a_2a_4$ it is satisfactory to process the grid $\mathcal{G}_n$ to find all elements of $\mathcal{D}(A)$ for any given image A of size n.

The advantage of $\mathcal{G}_n$ against $\mathcal{I}_n$ is the simple structure which can be easily generated on the fly. The disadvantage is the enormous growth of size, which, nevertheless, remains polynomial in n.

The following develops a rough idea of the sentence F that expresses Π. Particularly,

$$F = \exists t_1 \ldots \exists t_6 \quad \text{DELTA}[t_1, \ldots, t_6] \wedge (-10^{12}n^{13} \leq t_1) \wedge (t_1 \leq 10^{12}n^{13}) \wedge \ldots \\ \wedge (-10^{12}n^{13} \leq t_6) \wedge (t_6 \leq 10^{12}n^{13})$$

is build by a subformula DELTA$[t_1, \ldots, t_6]$ which is true for given parameters $t_1, \ldots, t_6$ if the string s encodes numbers and images that fulfill $\Delta(f(A), B) \leq t$ where f^{-1} is the transformation given by the grid point $(t_1, \ldots, t_6)^T$. In this fashion the sentence samples all points of the grid $\mathcal{G}_n$ and accepts if and only if at least one of them represents a transformation of A which resembles B enough in terms of t. Theorem 3 guarantees the correctness of this approach. Obviously t_1 to t_6 are variables representing long integers such that they can hold values polynomially in n.

The formula DELTA can be expressed in first order logic with majority quantifiers as follows: Basically, DELTA$[t_1, \ldots, t_6]$ has to (1) find the transformation f^{-1} represented by $(t_1, \ldots, t_6)^T$, (2) compute the sum $\Delta(f(A), B) = \sum_{(i,j)\in\mathcal{N}} \delta(A\langle f^{-1}(i,j)\rangle, B\langle i,j\rangle)$ and (3) compare this to t. The computation of f^{-1} by the grid point $(t_1, \ldots, t_6)^T$ means to determine

$$I = \left\lfloor \frac{(2t_1+1)i + 2t_2j + 2t_3}{2 \cdot 10^7 n^7} \right\rfloor \quad \text{and} \quad J = \left\lfloor \frac{2t_4i + (2t_5+1)j + 2t_6}{2 \cdot 10^7 n^7} \right\rfloor .$$

for all $(i,j) \in \mathcal{N}$. This is easily first-order-expressible with majority because it involves only a constant number of integer additions, multiplications, divisions and roundings, all functions in TC^0 [6,12]. Now since iterated addition is also in TC^0 [6] DELTA can easily compute the sum of $\delta(A(I,J), B(i,j))$ over all $(i,j) \in \mathcal{N}$. Consequently, the descriptiveness of F in first order logic with majority depends on the function $\delta : \Sigma \times \Sigma \to \mathbb{N}$. However, since Σ is finite it follows that δ is even first order-expressible. The expression equivalence between $FO[M]$ and TC^0 implies:

Lemma 1. *The decision version of Affine Image Matching is in* U_D-TC^0.

Consequently there exists a uniform family $\mathcal{C}$ of constant-depth, polynomial-size threshold circuits which decide Π. The optimization version of the AIMP can be computed by similar means using $\mathcal{C}$. Basically this can be done by constructing another family $\mathcal{C}_f$ of threshold circuits which try all possible values of $\Delta(f(A), B)$ in separate parallel copies of $\mathcal{C}$'s circuits. Since $\Delta(f(A), B) \leq m \cdot (2n+1)^2$, where $m = \max\{\delta(c_1, c_2) \mid c_1, c_2 \in \Sigma)$ is a constant, it follows that this approach induces at most a polynomial growth in size. Then, since the minimum of all satisfying distortion trials can be computed in U_D-AC^0 [6], the depth remains constant, too.

Theorem 4. *Affine Image Matching is in* U_D-FTC^0.

5 Completeness in TC^0

This section shows the decision version of the AIMP to be TC^0-complete. Consider the TC^0-complete majority problem, i.e., the set $\mathrm{MAJ} \subseteq \{0,1\}^*$ of strings which contain at least $\lceil 0.5|s| \rceil$ characters 1. This section gives an FAC^0-reduction r of MAJ to Π, the set of strings s encoding A, B and t such that the minimum $\Delta(f(A), B)$ over all affine transformations f is at most t. Hence, r is a function which maps strings $s \in \{0,1\}^*$ to a binary encoding of images A and B and an integer t such that $s \in \mathrm{MAJ}$ if and only if $\Delta(A, B) \leq t$. Remember, a function r is in FAC^0 if the set Π_r is in AC^0. Consequently, this section argues the existence of a first order sentence F which expresses Π_r, i.e., such that $\Pi_F = \Pi_r$.

The basic idea for the reduction is in fact very simple: Consider any string $s \in \{0,1\}^*$. Then imagine images A_s and B_s of size $n = 4|s|$ where $A_s\langle i, j\rangle = B_s\langle i, j\rangle = 0$ for all pixels $(i, j) \in \mathcal{N}$ with $j \neq 0$. Additionally set

$$A_s\langle i, 0\rangle = \begin{cases} s\langle i\rangle, & \text{if } 0 \leq i < |s| \\ 1, & \text{otherwise} \end{cases} \quad \text{and} \quad B_s\langle i, 0\rangle = 1$$

for all $i \in \{-n, \ldots, n\}$ and let $t_s = \lfloor 0.5|s| \rfloor$. Obviously, A_s contains a copy of s and B_s a row of 1-characters. Moreover t_s describes the maximum number of 0-characters in s to be in MAJ. Then the majority of characters in s is 1 if and only if $\Delta(A_s, B_s) \leq t_s$ for the distortion measure $\delta(c_1, c_2) = |c_1 - c_2|$. Hence, if transformations were not allowed on A_s this approach would already be successful.

However, the AIMP allows any affine transformation on A_s and thus, the above relation is not enough. To use a similar approach A_s and B_s are extended in such a way that the transformations that lead to the optimal match are close to identity and thus, still count the number of zeros in s. For this end leave most of A_s and B_s as before but for all $k \in \{-n, \ldots, n\}$ let

$$\begin{array}{llll} A_s\langle k, -n+2\rangle = A_s\langle k, n-2\rangle = A_s\langle -n+2, k\rangle = A_s\langle n-2, k\rangle = \\ B_s\langle k, -n\rangle \quad = B_s\langle k, n\rangle \quad = B_s\langle -n, k\rangle \quad = B_s\langle n, k\rangle \quad = 2 \end{array}$$

i.e., draw a frame in A_s and a little bigger frame in B_s. Now consider the following lemma:

Lemma 2. *Let $s \in \{0,1\}^*$ and A_s, B_s and t_s as defined above. Moreover, consider the transformation*

$$f_{opt}(x,y) = \left(\begin{smallmatrix} a & 0 \\ 0 & a \end{smallmatrix}\right) \cdot \left(\begin{smallmatrix} x \\ y \end{smallmatrix}\right) + \left(\begin{smallmatrix} 0 \\ 0 \end{smallmatrix}\right)$$

where $a = \frac{16|s|}{16|s|-7}$. Then s contains a majority of characters 1 if and only if $\Delta(f_{opt}(A_s), B_s) \leq t_s$ for $\delta(c_1, c_2) = |c_1 - c_2|$.

The transformation f_{opt} guarantees that (1) the string s remains unaltered and (2) the rest of $f_{opt}(A_s)$ looks like B_s. This means $\Delta(f_{opt}(A_s), B_s)$ equals the number of 0 in s and thus, is at most t_s if and only if the majority of characters in s is 1. However, that is not enough. It remains to show that there is no transformation $f' \in \mathcal{F}$ that performs a better match of $f'(A)$ to B because otherwise the solution of the AIMP would rather go with f' and not f_{opt}:

Lemma 3. *For all $s \in \{0,1\}^*$ it is true that $\Delta(f_{opt}(A_s), B_s)$ under $\delta(c_1, c_2) = |c_1 - c_2|$ is minimum over all affine transformations in $\mathcal{F}$.*

The basic idea behind the lemma's proof is that the frames in $f(A_s)$ and B_s have to be aligned. Together with the alignment of the row $j = 0$ this leaves all in all only one true transformation $f(A_s)$. Moreover, the small frame in A guarantees that $f(A_s)$ has to be scaled up to match B's frame. This results in the effect that every pixel (I, J) in the center of A is represented by a pixel (i, j) in $f(A_s)$, i.e., $(I, J) = [f^{-1}(i, j)]$. Consequently no s-character 0 represented in A is forgotten in $f(A_s)$. Thus, every transformation $f(A_s)$ has to count at least all characters 0 in the string s.

The rest is to argue that the computation $r(s) = (A_s, B_s, t_s)$ of the images A_s and B_s as well as the threshold t_s from the string s can be accomplished by a first order sentence expressing Π_r. However, a big deal of that is simply copying and filling in constants. In particular, both A_s and B_s can be computed by these simple operations, namely, inserting the string s in A_s and preparing the frames in both images. The most complex work is the computation of t_s because it contains a division. But whereas general division is TC^0-complete the division by the constant two can be established using only addition by

$$\mathrm{DIV2}[x,y] = \exists z \ \mathrm{ADD}[y,y,x] \vee (\mathrm{ADD}[y,y,z] \wedge \mathrm{ADD}[z,1,x]),$$

i.e., there is a first order subformula $\mathrm{DIV2}[x, y]$ that expresses $\lfloor 0.5x \rfloor = y$. The following theorem states the completeness result:

Theorem 5. *The decision version of the AIMP is U_D-TC^0-complete under U_D-FAC^0-reductions.*

6 Conclusions

This paper analyzes the complexity of affine image matching. It argues the existence of a first order sentence using the majority quantifier that expresses this problem, thus, showing that affine image matching is contained in U_D-FTC^0.

Moreover, it gives a U_D-FAC^0-reduction from majority to affine image matching and therefore provides that the problem is even complete in U_D-TC^0.

This work concentrates on affine image matching and neglects the superset of projective transformations and the subset of linear transformations considered in [14]. It is a natural conjecture that linear image matching can also be solved in TC^0 and that projective image matching is at least hard in TC^0.

In particular the whole approach of this paper can be easily transfered to the case of projective transformations, i.e., even projective image matching is U_D-TC^0-complete under U_D-FAC^0-reductions. This paper sticks to affine image matching only for convenience because some results become more technical for projective transformations. Beside the pure complication of introducing previous work, especially the formulation of an analogue of Theorem 3 requires handling a multiplicity of cases which makes the basic ideas less perspicuous.

For linear transformations the case is more complicated. Although a proper subclass, the structure of linear transformations is geometrically harder than in the affine case. This results basically from the fact that the arrangement $\mathcal{A}(H_n)$ under linear transformations contains faces which have no volume. Consequently, it is likely that they are missed during a sampling process as described in this paper. The same holds for several of the small subclasses of affine transformations like scaling and rotation which were analyzed in [15]. Although the author believes that image matching under each of these classes can be done in TC^0, it remains open whether this is true. At least the problem's hardness for TC^0 is evident even for scalings because the reduction builds mainly on Δ and benefits from a restriction on the class of transformations.

Regarding the practicability of this paper's results notice that problems in TC^0 can be solved very efficiently in time. However, whereas TC^0 restricts size only polynomially, it is practically impossible to create circuits of n^{78} processors even for small n. But the containment in TC^0 has to be seen more in a structural context that gains insight into the problem's properties. Particularly, the immense growth in size is partially caused by the weak uniformity constraint, which is a natural choice for complexity analysis. But more powerful models of construction produce much smaller circuits. Consider e.g. P-uniform TC^0 families, i.e., where circuits C_n must be constructible by a Turing machine in polynomial time. This is a very natural model for practical settings because it allows resources consumption during the planning of C_n but saves them when C_n comes into operation. Under this setting the graph $\mathcal{I}_n$ can be generated and then used to construct C_n, thus, to reduce the size consumption of the circuit.

According to the above note, this paper is another step towards image matching in real applications. The author hopes that it helps to initiate future work on the practical aspects of image matching like for example the first impressions that were revealed from [20].

Acknowledgment

The author thanks Maciej Liśkiewicz and Ragnar Nevries for helpful ideas and proof reading as well as the anonymous reviewers for improvement suggestions.

References

1. Amir, A., Butman, A., Crochemore, M., Landau, G., Schaps, M.: Two-dimensional pattern matching with rotations. Theor. Comput. Sci. 314(1-2), 173–187 (2004)
2. Amir, A., Butman, A., Lewenstein, M., Porat, E.: Real two dimensional scaled matching. Algorithmica 53(3), 314–336 (2009)
3. Amir, A., Chencinski, E.: Faster two-dimensional scaled matching. In: Lewenstein, M., Valiente, G. (eds.) CPM 2006. LNCS, vol. 4009, pp. 200–210. Springer, Heidelberg (2006)
4. Amir, A., Kapah, O., Tsur, D.: Faster two-dimensional pattern matching with rotations. In: Sahinalp, S.C., Muthukrishnan, S.M., Dogrusoz, U. (eds.) CPM 2004. LNCS, vol. 3109, pp. 409–419. Springer, Heidelberg (2004)
5. Brown, L.G.: A survey of image registration techniques. ACM Computing Surveys 24(4), 325–376 (1992)
6. Chandra, A.K., Stockmeyer, L., Vishkin, U.: Constant depth reducibility. SIAM J. Comput. 13(2), 423–439 (1984)
7. Cox, I.J., Bloom, J.A., Miller, M.L.: Digital Watermarking, Principles and Practice. Morgan Kaufmann, San Francisco (2001)
8. Edelsbrunner, H.: Algorithms in Combinatorial Geometry. Springer, Berlin (1987)
9. Edelsbrunner, H., O'Rourke, J., Seidel, R.: Constructing arrangements of lines and hyperplanes with applications. SIAM J. Comput. 15, 341–363 (1986)
10. Fredriksson, K., Navarro, G., Ukkonen, E.: Optimal exact and fast approximate two-dimensional pattern matching allowing rotations. In: Apostolico, A., Takeda, M. (eds.) CPM 2002. LNCS, vol. 2373, pp. 235–248. Springer, Heidelberg (2002)
11. Fredriksson, K., Ukkonen, E.: A rotation invariant filter for two-dimensional string matching. In: Farach-Colton, M. (ed.) CPM 1998. LNCS, vol. 1448, pp. 118–125. Springer, Heidelberg (1998)
12. Hesse, W.: Division is in uniform TC^0. In: Orejas, F., Spirakis, P.G., van Leeuwen, J. (eds.) ICALP 2001. LNCS, vol. 2076, pp. 104–114. Springer, Heidelberg (2001)
13. Hundt, C., Liśkiewicz, M.: On the complexity of affine image matching. In: Thomas, W., Weil, P. (eds.) STACS 2007. LNCS, vol. 4393, pp. 284–295. Springer, Heidelberg (2007)
14. Hundt, C., Liśkiewicz, M.: Combinatorial bounds and algorithmic aspects of image matching under projective transformations. In: Ochmański, E., Tyszkiewicz, J. (eds.) MFCS 2008. LNCS, vol. 5162, pp. 395–406. Springer, Heidelberg (2008)
15. Hundt, C., Liśkiewicz, M., Nevries, R.: A combinatorial geometric approach to two-dimensional robustly pattern matching with scaling and rotation. Theor. Comput. Sci. 51(410), 5317–5333 (2009)
16. Kasturi, R., Jain, R.C.: Computer Vision: Principles. IEEE Computer Society Press, Los Alamitos (1991)
17. Landau, G.M., Vishkin, U.: Pattern matching in a digitized image. Algorithmica 12(3/4), 375–408 (1994)
18. Maintz, J.B.A., Viergever, M.A.: A survey of medical image registration. Medical Image Analysis 2(1), 1–36 (1998)
19. Modersitzki, J.: Numerical Methods for Image Registration. Oxford University Press, Oxford (2004)
20. Nevries, R.: Entwicklung und Analyse eines beschleunigten Image Matching-Algorithmus für natürliche Bilder, Diplomarbeit, Universität Rostock (2008)
21. Vollmer, H.: Introduction to circuit complexity. Springer, Berlin (1999)

Old and New in Stringology

Zvi Galil

Blavatnik School of Computer Science
Tel Aviv University

Twenty five years ago in a paper titled "Open Problems in Stringology" I listed thirteen open problems in a field I called Stringology. The first part of the talk will revisit the list. Some problems were solved, others were partially solved and some resisted any progress.

The second part of the talk will review some recent results in Stringology, namely algorithms in the streaming model. In this model, the algorithms cannot store the entire input string(s) and can use only very limited space. Surprisingly, efficient algorithms were discovered for a number of string problems.

The talk will conclude with new open problems that are raised by these new results.

A. Amir and L. Parida (Eds.): CPM 2010, LNCS 6129, p. 26, 2010.

Small-Space 2D Compressed Dictionary Matching

Shoshana Neuburger[1,⋆] and Dina Sokol[2,⋆⋆]

[1] Department of Computer Science, The Graduate Center of the City University of New York, New York, NY, 10016
shoshana@sci.brooklyn.cuny.edu

[2] Department of Computer and Information Science, Brooklyn College of the City University of New York, Brooklyn, NY, 11210
sokol@sci.brooklyn.cuny.edu

Abstract. The dictionary matching problem seeks all locations in a text that match any of the patterns in a dictionary. In the *compressed dictionary matching* problem, the input is in compressed form. In this paper we introduce the 2-dimensional compressed dictionary matching problem in Lempel-Ziv compressed images, and present an efficient solution for patterns whose rows are all *periodic*. Given k patterns, each of (uncompressed) size $m \times m$, and a text of (uncompressed) size $n \times n$, all in 2D-LZ compressed form, our algorithm finds all occurrences of the patterns in the text. The algorithm is *strongly inplace*, i.e., the extra space it uses is proportional to the optimal compression of the dictionary, which is $O(km)$. The preprocessing time of the algorithm is $O(km^2)$, linear in the uncompressed dictionary size, and the time for performing the search is linear in the uncompressed text size, independent of the dictionary size. Our algorithm is general in the sense that it can be used for any 2D compression scheme which can be sequentially decompressed in small space.

1 Introduction

The compressed matching problem is the problem of finding all occurrences of a pattern in a compressed text. Various algorithms have been devised to solve the 2D compressed matching problem, e.g., [6, 3, 4]. The dictionary matching problem is that of locating all occurrences of a set of patterns in a given text. In this paper we introduce the compressed dictionary matching problem in 2-dimensions. Compressed dictionary matching can be trivially solved using any compressed pattern matching algorithm and searching for each pattern separately. Preferably, an algorithm should scan the text once so that its search time depends only on the size of the text and not on the size of the dictionary of patterns. Aho and Corasick achieved this goal for uncompressed patterns and text.

⋆ This work has been supported in part by the National Science Foundation Grant DB&I 0542751.

⋆⋆ This work has been supported in part by the National Science Foundation Grant DB&I 0542751 and the PSC-CU Research Award 62280-0040.

A. Amir and L. Parida (Eds.): CPM 2010, LNCS 6129, pp. 27–39, 2010.

We address 2D compressed dictionary matching when the patterns and text are in LZ78 compressed form. Space is an important concern of a compressed pattern matching algorithm. An algorithm is *strongly inplace* if the amount of extra space it uses is proportional to the optimal compression of the data. The algorithm we present is both linear time and strongly inplace. The problem we are addressing is of practical significance. Many images are stored in Lempel-Ziv compressed form. Facial recognition is a direct application of 2D compressed dictionary matching. The goal of such software is to identify individuals in a larger image based on a dictionary of previously identified faces. An efficient algorithm does not depend on the size of the database of known images.

Pattern matching cannot be performed directly on compressed data since compression is context-sensitive. The same uncompressed string can be compressed differently in different files, depending on the data that precedes the matching content. The *key property* of LZ78 is the ability to perform decompression using constant space in time linear in the uncompressed string. We follow the assumption of Amir et. al. [3] and consider the row-by-row linearization of 2D data.

Existing algorithms for 2D dictionary matching are not sequential. Thus, they are not easily adapted to form strongly-inplace algorithms. Amir and Farach contributed a 2D dictionary matching algorithm that can be used for square patterns [2]. Its time complexity is linear in the size of the text with preprocessing time linear in the size of the dictionary. Their algorithm linearizes the patterns by considering subrow/subcolumn pairs around the diagonal, which is not conducive to row-by-row decompression. Idury and Schaffer discuss multiple pattern matching in two dimensions for rectangular patterns [9]. Although their algorithm is efficient, the data structures require more space than we allow.

We do not know of a small-space dictionary matching algorithm for even one-dimensional data. Multiple pattern matching in LZW compressed text is addressed by Kida et. al. [10]. They present an algorithm that simulates the Aho-Corasick search mechanism for compressed, one-dimensional texts and an uncompressed dictionary of patterns. Their approach uses space proportional to both the compressed text and uncompressed dictionary sizes, which is more space than we allow.

In this paper we present an algorithm that solves the *2D LZ-Compressed Dictionary Matching Problem* where all pattern rows are periodic and the periods are no greater than $m/4$. Given a dictionary of 2D LZ-compressed patterns, $P_1, P_2, \ldots, P_k$, each of uncompressed size $m \times m$, and a compressed text of uncompressed size $n \times n$, we find all occurrences of patterns in the text. Our algorithm is strongly inplace since it uses $O(km)$ space. The best compression that LZ78 can achieve on the dictionary is $O(km)$ [14]. The time complexity of our algorithm is $O(km^2 + n^2 \log \sigma)$, where $\sigma = min(km, |\Sigma|)$ and Σ is the alphabet. After preprocessing the dictionary, the time complexity is independent of the dictionary size.

Amir et. al. present an algorithm for strongly-inplace single pattern matching in 2D LZ78-compressed data [3]. Their algorithm requires $O(m^3)$ time to preprocess the pattern of uncompressed size $m \times m$ and search time proportional

to the uncompressed text size. Our preprocessing scheme can be used to reduce the preprocessing time of their algorithm to $O(m^2)$, linear in the size of the uncompressed pattern, resulting in an overall time complexity of $O(m^2 + n^2)$.

2 Overview

We overcome the space requirement of traditional 2D dictionary matching algorithms with an innovative preprocessing scheme that converts 2D patterns to a linear representation. The pattern rows are initially classified into groups, with each group having a single representative. We store a *witness*, or position of mismatch, between the group representatives. A 2D pattern is named by the group representative for each of its rows. This is a generalization of the naming technique used by Bird [7] and Baker [5] to linearize 2D data. The preprocessing is performed in a single pass over the patterns with no need to decompress more than two pattern rows at a time. $O(1)$ information is stored per pattern row, resulting in a total of $O(km)$ information. Details of the preprocessing stage can be found in Section 3.

In the text scanning phase, we name the rows of the text to form a 1D representation of the 2D text. Then, we use an Aho-Corasick (AC) automaton [1] to mark candidates of possible pattern occurrences in the 1D text in $O(n^2 \log \sigma)$ time. Since similar pattern rows were grouped together, we need a verification stage to determine if the candidates are actual pattern occurrences. With additional preprocessing of the 1D pattern representations, a single pass suffices to verify potential pattern occurrences in the text. The details of the text scanning stage are described in Section 4.

The algorithm of Amir et. al. [3] is divided into two cases. A pattern can (i) have only periodic rows with all periods $\leq m/4$ or (ii) have at least one aperiodic row or a row with a period $> m/4$. We focus on the more difficult case, (i). In such an instance, the number of pattern occurrences is potentially larger than the amount of working space we allow. Our algorithm performs linear-time strongly-inplace 2D LZ-compressed dictionary matching of patterns in which all rows are periodic with periods $\leq m/4$.

A known technique for minimizing space is to work with small overlapping text blocks of uncompressed size $3m/2 \times 3m/2$. The potential starts all lie in the upper-left $m/2 \times m/2$ square. If $O(km^2)$ space were allowed, then the 2D-LZ dictionary matching problem would easily be solved by decompressing small text blocks and using any known 2D dictionary matching algorithm within each text block. However, a strongly-inplace algorithm, such as ours, uses only $O(km)$ extra space.

We follow the framework of [3] to sequentially decompress small blocks of 2D-LZ data in time linear in the uncompressed text and in constant space. $O(m)$ pointers are used to keep track of the current location in the compressed text.

3 Pattern Preprocessing

Definition 1. *A string p is* primitive *if it cannot be expressed in the form $p = u^k$, for $k > 1$ and a prefix u of p.*

Definition 2. *A string p is* periodic *in u if $p = u'u^k$ where u' is a suffix of u, u is primitive, and $k \geq 2$.*

A periodic string p can be expressed as $u'u^k$ for one unique primitive u. We refer to u as "the period" of p. Depending on the context, u can refer to either the string u or the period size $|u|$.

Definition 3. *[8] A 2D $m \times m$ pattern is* h-periodic*, or horizontally periodic, if two copies of the pattern can be aligned in the top row so that there is no mismatch in the region of overlap and the length of overlap in each row is $\geq m/2$.*

Observation 1. *A 2D pattern is h-periodic iff each of its rows is periodic.*

A dictionary of h-periodic patterns can occur $\Omega(km)$ times in a text block. It is difficult to search for periodic patterns in small space since the output can be larger than the amount of extra space we allow. We take advantage of the periodicity of pattern rows to succinctly represent pattern occurrences. The distance between any two overlapping occurrences of P_i in the same row is the Least Common Multiple (LCM) of the periods of all rows of P_i. We precompute the LCM of each pattern so that $O(1)$ space suffices to store all occurrences of a pattern in a row, and $O(km)$ space suffices to store all occurrences of h-periodic patterns.

We introduce two new data structures that allow our algorithm to achieve a small space yet linear time complexity. They are the witness tree and the offset tree. The witness tree facilitates the linear-time preprocessing of pattern rows. The offset tree allows the text scanning stage to achieve linear time complexity, independent of the number of patterns in the dictionary.

3.1 Lyndon Word Naming

Definition 4. *Two words x, y are* conjugate *if $x = uv$, $y = vu$ for some words u, v [12].*

Definition 5. *A* Lyndon word *is a primitive string which is lexicographically smaller than any of its conjugates [12].*

We partition the pattern rows into disjoint groups. Each group is given a different name and a representative is chosen for each group. Pattern rows whose periods are conjugates of each other are grouped together. Conjugacy is an equivalence relation. Every primitive word has a conjugate which is a Lyndon word; namely, its least conjugate. Computing the smallest conjugate of a word is a practical way of obtaining a standard representation of a word's conjugacy class. This process is called *canonization* and can be done in linear time and space [12]. We will use the same 1D name to represent all patterns whose periods are conjugates of each other. This enables us to linearize the 2D patterns in a meaningful manner.

We decompress and name the pattern rows, one at a time. After decompressing a row, its period is found and canonized. If a new Lyndon word or a new period size is encountered, the row is given a new name. Otherwise, the row adopts the name already given to another member of its conjugacy class. A 2D pattern is

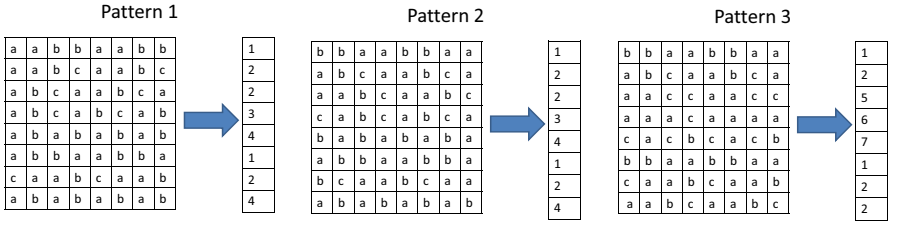


Fig. 1. Three 2D patterns with their 1D representations. Patterns 1 and 2 are not the same, yet their 1D representations are the same.

transformed to a 1D representation by naming its rows. Thus, an $m \times m$ pattern can be represented in $O(m)$ space and a 2D dictionary can be represented in $O(km)$ space.

Three 2D patterns and their 1D representations are shown in Figure 1. To understand the naming process we will look at Pattern 1. The period of the first row is *aabb*, which is four characters long. It is given the name *1*. When the second row is examined, its period is found to be *aabc*, which is also four characters long. *aabb* and *aabc* are both Lyndon words of size four, but they are different, so the second row is named *2*. The period of the third row is *abca*, which is represented by the Lyndon word *aabc*. Thus, the second and third rows are given the same name even though they are not identical.

Pattern preprocessing is performed on one row at a time to conserve space. We decompress one row at a time and gather the necessary information. An LZ78 compressed string can be decompressed in time and space linear to the size of the uncompressed string [3]. After decompressing a pattern row, its period is identified using known techniques in linear time and space, i.e., using a KMP automaton [11] of the string. Then, we compute and store $O(1)$ information per row[1]: period size, name, and position of the first Lyndon word occurrence in the row ($LYpos$).

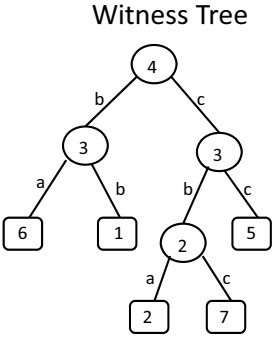
We use the *witness tree* to name the pattern rows. Since we know which Lyndon word represents a row, the same name is given to pattern rows whose periods are conjugates of each other. Rows that have already been named are stored in a witness tree. We only compare a new string to previously named strings of the same size. The witness tree keeps track of failures in Lyndon word character comparisons. With the witness tree, we compare at most one named row to the new row.

3.2 Witness Tree

Components of witness tree:

– *Internal node:* position of a character mismatch. The position is an integer $\in [1, m]$.

[1] This is under the assumption that the word size is large enough to store $\log m$ bits in one word.



Name	Period size	Lyndon word
1	4	aabb
2	4	aabc
3	3	abc
4	2	ab
5	4	aacc
6	4	aaab
7	4	acbc

Fig. 2. A witness tree for the Lyndon words of length 4

- *Edge:* labeled with a character in Σ. Two edges emanating from a node must have different labels.
- *Leaf:* an equivalence class representing one or more pattern rows.

When a new row is examined, we need to determine if the Lyndon word of its period has already been named. An Aho-Corasick [1] automaton completes this task in $O(km^2)$ time and space, but we allow only $O(km)$ space. The witness tree allows us to identify the only named string of the same size that has no recorded position of mismatch with the new string, if there is one. A witness tree for Lyndon words of length four is depicted in Figure 2.

The witness tree is used as it is constructed in the pattern preprocessing stage. As strings of the same size are compared, points of distinction between the representatives of 1D names are identified and stored in a tree structure. When a mismatch is found between strings that have no recorded distinction, comparison halts, and the point of failure is added to the tree. Characters of a new string are examined in the order dictated by traversal of the witness tree, possibly out of sequence. If traversal halts at an internal node, the string receives a new name. Otherwise, traversal halts at a leaf, and the new string is sequentially compared to the string represented by the leaf. Depending on whether comparison completes successfully, the new string receives either the name of the leaf or a new name.

As an example, we explain how the name *7* becomes a leaf in the witness tree of Figure 2. We seek to classify the Lyndon word *acbc*, using the witness tree for Lyndon words of size four. Since the root represents position 4, the first comparison finds that *c*, the fourth character in *acbc*, matches the edge connecting the root to its right child. This brings us to the right child of the root, which tells us to look at position 3. Since there is a *b* at the third position of *acbc*, we reach the leaf labeled *2*. Thus, we compare the Lyndon words *acbc* and *aabc*. They differ at the second position, so we create an internal node for position 2, with leaves labeled *2* and *7* as its children, and their edges labeled *a* and *c*, respectively.

Lemma 1. *Of the named strings that are the same size as a new string, i, there is at most one equivalence class, j, that has no recorded mismatch against i.*

Proof. The proof is by contradiction. Suppose we have two such classes, l and j. Both l and j have the same size as i and neither has a recorded mismatch with i. By transitivity of the equivalence relation, we have not recorded a mismatch between l and j. This means that l and j should have received the same name. This contradicts the assumption that l and j are different classes. □

Lemma 2. *The witness trees for the rows of k patterns, each of size $m \times m$, is $O(km)$ in size.*

Proof. The proof is by induction. The first time a string of size u is encountered, initialize the tree for strings of size u to a single leaf. Subsequent examination of a string of size u contributes either zero or one new node (with an accompanying edge) to the tree. Either the string is given a name that has already been used or it is given a new name. If the string is given a name already used, the tree remains unchanged. If the string is given a new name, it mismatched another string of the same size. There are two possibilities to consider.

(i) A leaf is replaced with an internal node to represent the position of mismatch. The new internal node has two leaves as its children. One leaf represents the new name, and the other represents the string to which it was compared. The new edges are labeled with the characters that mismatched.

(ii) A new leaf is created by adding an edge to an existing internal node. The new edge represents the character that mismatched and the new leaf represents the new name. □

Corollary 1. *The witness tree for Lyndon words of length u has depth $\leq u$.*

Lemma 3. *A pattern row of size $O(m)$ is named in $O(m)$ time using the appropriate witness tree.*

Proof. By Lemma 1, a new string is compared to at most one other string, j. A witness tree is traversed from the root to identify j. Traversal of a witness tree ceases either at an internal node or at a leaf. The time spent traversing a tree is bounded by its depth. By Corollary 1, the tree-depth is $O(m)$, so the tree is traversed in $O(m)$ comparisons. Thus, a new string is classified with $O(m)$ comparisons. □

The patterns are named in $O(km^2)$ time using only $O(km)$ extra space. This time complexity is optimal since each pattern row must be decompressed and examined at least once. Since we require only $O(1)$ rows to be decompressed at a time, naming is done within $O(m)$ extra space.

3.3 Preprocessing the 1D Patterns

Once the pattern rows are named, an Aho-Corasick (AC) automaton is constructed for the 1D patterns of names. (See Figure 1 for the 1D names of three

patterns.) Several different patterns have the same 1D name if their rows belong to the same equivalence classes. This is easily detected in the AC automaton since the patterns occur at the same terminal state.

The next preprocessing step computes the Least Common Multiple (LCM) of each distinct 1D pattern. This can be done incrementally, one row at a time, in time proportional to the number of pattern rows. The LCM of an h-periodic pattern reveals the horizontal distance between its candidates in a text block. This conserves space as there are fewer candidates to maintain. In effect, this will also conserve verification time.

If several patterns share a 1D name, an *offset tree* is constructed of the Lyndon word positions in these patterns. We defer the description of the offset tree to Section 4.1 where it is used in the verification phase.

In summary, pattern preprocessing in $O(km^2)$ time and $O(m)$ space:

1. For each pattern row, (i) decompress (ii) compute period and canonize (iii) store period size, name, first Lyndon word occurrence (*LYpos*).
2. Construct AC automaton of 1D patterns.
3. Find LCM of each 1D pattern.
4. For multiple patterns of same 1D name, build offset tree.

4 Text Scanning

Our algorithm processes the text once and searches for all patterns simultaneously. The text is broken into overlapping blocks of uncompressed size $3m/2 \times 3m/2$. Each text row is decompressed $O(1)$ times with 1 or 2 pointers to mark the end of each row in the block of text. One pointer indicates the position in the compressed text. When the endpoint of a row in the text block occurs in middle of a compressed character, a second pointer indicates its position within the compressed character. In total, $O(m)$ pointers are used to keep track of the current location in the compressed text.

The text scanning stage has three steps:

1. Name rows of text.
2. Identify candidates with a 1D dictionary matching algorithm, e.g., AC.
3. Verify candidates separately for each text row using the offset tree of the 1D pattern.

Step 1. Name Text Rows
We search a 2D text for a 1D dictionary patterns using a 1D Aho-Corasick (AC) automaton. A 1D pattern can begin at any of the first $m/2$ positions of a text block row. The AC automaton can branch to one of several characters; we can't afford the time or space to search for each of them in the text row. Thus, we name the rows of a text block before searching for patterns. The divide-and-conquer algorithm of Main and Lorentz [13] finds all maximal repetitions that cross a given point in linear time. Repetitions of length $\geq m$ that cross the midpoint and have a period size $\leq m/4$ are the only ones that are of interest to our algorithm.

Lemma 4. *At most one maximal periodic substring of length* $\geq m$ *with period* $\leq m/4$ *can occur in a text block row of size* $3m/2$.

Proof. The proof is by contradiction. Suppose that two maximal periodic substrings of length m, with period $\leq m/4$ occur in a row. Call the periods of these strings u and v. Since we are looking at periodic substrings that begin within an $m/2 \times m/2$ square, the two substrings overlap by at least $m/2$ characters. Since u and v are no larger than $m/4$, at least two adjacent copies of both u and v occur in the overlap. This contradicts the fact that both u and v are primitive. □

After finding the only maximal periodic substring of length $\geq m$ with period $\leq m/4$, the text rows are named in much the same way as the pattern rows are named. The period of the maximal run is found and canonized. Then, the appropriate witness tree is used to name the text row. We use the witness tree constructed during pattern preprocessing since we are only interested in identifying text rows that correspond to Lyndon words found in the pattern rows. At most one pattern row will be decompressed to classify the conjugacy class of a text row. In addition to the name, period size, and *LYpos*, we maintain a *left* and *right* pointer for each row of a text block. *left* and *right* mark the endpoints of the periodic substring in the text. The *LYpos* (position of first Lyndon word occurrence) is computed relative to the *left* pointer of the row. This process is repeated for each row, and $O(m)$ information is obtained for the text block.

Complexity of Step 1: The largest periodic substring of a row of width $3m/2$, if it exists, can be found in $O(m)$ time and space [13]. Its period can be found and canonized in linear time and space [12]. The row is named in $O(m)$ time and space using the appropriate witness tree (Lemma 3). Overall, $O(m^2)$ time and $O(m)$ space is needed to name the rows of a text block.

Step 2. Identify Candidates
After Step 1 completes, a 1D text remains, each row labeled with a name, period size, *LYpos*, and *left*/*right* boundaries. A 1D dictionary matching algorithm, such as AC, is used to mark occurrences of the 1D patterns of names. The occurrence of a 1D pattern indicates the potential occurrence of 2D pattern(s) since several 2D dictionary patterns can have the same 1D name. All *candidates*, or possible pattern starts, are in rows marked with occurrences of the 1D pattern. The occurrence of a 1D pattern is not sufficient evidence that a 2D pattern actually occurs. Thus, a separate verification step is necessary. The *left* pointer with the LYpos identify the first occurrence of a pattern row. Since the patterns are h-periodic, pattern occurrences are at multiples of the first row's period size that leave enough space for the pattern width before *right*.

Complexity of Step 2: 1D dictionary matching in a string of size m can be done in $O(m \log \sigma)$ time and $O(mk)$ space using an AC automaton [1].

Step 3. Verify Candidates
The verification process considers each row of text that contains candidates separately. Recall that a text row contains candidates iff a 1D pattern begins

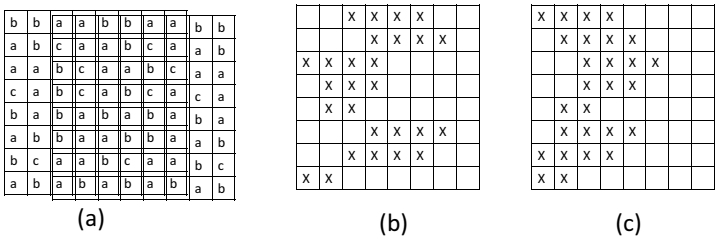


Fig. 3. (a) Two consistent patterns are shown. Each pattern is a horizontal cyclic shift of the other. (b) The first Lyndon word occurrence on each row of the pattern is represented by a sequence of Xs. (c) The representative of this consistency class. The class representative is the shift in which the Lyndon word of the first row begins at the first position.

there. Several patterns can share a 1D representation. We need to verify the overall width of the 1D names, as well as the alignment of the periods among rows.

After identifying a text row as containing candidates for a pattern occurrence, we need to ensure that the labeled periodic string extends over at least m columns in each of the next m rows. We are interested in the minimum of all *right* pointers, *minRight*, as well as the maximum of all *left* pointers, *maxLeft*, as this is the range of positions in which the pattern(s) can occur. If the pattern will not fit between *minRight* and *maxLeft*, i.e., $minRight - maxLeft < m$, candidates in the row are eliminated.

The verification stage must also ascertain that the Lyndon word positions in the text align with the Lyndon word positions in the pattern rows. Naively, this can be done in $O(m^3)$ time. We verify a candidate row in $O(m)$ time using the offset tree of a 1D pattern.

Several different patterns that have the same 1D representation can occur at overlapping positions on the same text row. We call such a set of patterns *consistent*. Consistent patterns can be obtained from one another by performing a horizontal cyclic permutation of the characters, i.e., by moving several columns to the opposite end of the matrix. Figure 3 depicts a pair of consistent patterns. Pattern consistency is an equivalence relation. We can form equivalence classes of patterns with the same 1D name and then classify the text as belonging to at most one group. We choose a representative for each equivalence class. The class representative is the shift in which the Lyndon word of the first row begins at the first position.

Each row of the 2D array is represented by its 1D arrays of names and *LYpos*. To convert a pattern to one that is consistent with it, its rows are shifted by the same constant, but the *LYpos* of its rows may not be. However, the shift is the same across the rows, relative to the period size of each row. Figure 3 shows an example of consistent patterns and the relative shifts of their rows. Notice that (b) can be obtained from (c) by shifting two columns towards the left. The first occurrence of the Lyndon word of the first row is at position 3 in

(b) and at position 1 in (c). This shift seems to reverse in the third row, since the Lyndon word first occurs at position 1 in (b) and at position 3 in (c). However, the relative shift remains the same, since the shift is cyclic. We summarize this relationship in the following lemma.

Lemma 5. *Two patterns are consistent iff the* LYpos *of all their rows are shifted by* C mod period size of the row*, where C is a constant.*

The proof is omitted due to lack of space and will be included in the journal version.

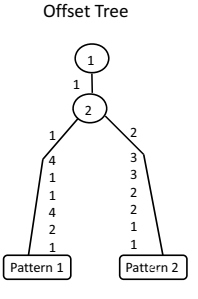
4.1 Offset Tree

We construct an offset tree to align the shifted *LYpos* arrays of patterns with the same 1D name so that the text can be classified, and ultimately verified, in $O(m)$ time. This allows the text scanning stage to complete in time proportional to the text size, independent of the dictionary size. An offset tree is shown in Figure 4.

Components of offset tree:

- *Root:* represents the first row of a pattern.
- *Internal node:* represents a row from 1 to m, strictly larger than its parent.
- *Edge:* labeled by shifted *LYpos* entries. Two edges that leave a node must have different labels.
- *Leaf:* represents a consistency class of dictionary patterns.

We construct an offset tree for each 1D pattern of names. One pattern at a time, we traverse the tree and compare the shifted *LYpos* arrays in sequential order until a mismatch is found or we reach a leaf. If a mismatch occurs at an edge leading to a leaf, a new internal node and a leaf are created, to represent the position of mismatch and the new consistency class, respectively. If a mismatch occurs at an edge leading to an internal node, a new branch is created (and possibly a new internal node) with a new leaf to represent the new consistency class.



Name	Period size	Pattern1 LYpos	Pattern2 LYpos
1	4	1	3
2	4	1	4
2	4	4	1
3	3	1	2
4	2	1	2
1	4	4	4
2	4	2	3
4	2	1	1

Fig. 4. Offset tree for patterns 1 and 2 which have the same 1D name. The *LYpos* entries are not shifted for the first pattern since its first entry is 1, while the *LYpos* entries of the second pattern are shifted by 2 mod period size of row.

Lemma 6. *The consistency class of a string of length m is found in $O(m)$ time.*

The proof is omitted due to lack of space and will be included in the journal version.

Observation 2. *The offset trees for k 1D patterns, each of size m, is of size $O(km)$.*

We modify the *LYpos* array of the text to reflect the first Lyndon word occurrence in each text row after *maxLeft*. Each modified *LYpos* entry is $\geq$ *maxLeft* and can be computed in $O(1)$ time with basic arithmetic.

We shift the text's *LYpos* values so that the Lyndon word of the first row occurs at the first position. We traverse the offset tree to determine which pattern(s), if any, are consistent with the text. If traversal ceases at a leaf, then its pattern(s) can occur in the text, provided the text is sufficiently wide.

At this point, we know which patterns are consistent with the window of m rows beginning in a given text row. The last step is to locate the actual positions at which a pattern begins, within the given text row. We need to reverse the shift of the consistent patterns by looking up the first *LYpos* of each pattern that is consistent with the text block. Then we verify that the periodic substrings of the text are sufficiently wide. That is, we announce position i as a pattern occurrence iff $minRight - i \geq m$. Subsequent pattern occurrences in the same row are at LCM multiples of the pattern.

Complexity of Step 3: There can be $O(m)$ rows in a text block that contain candidates. *maxLeft* and *minRight* are computed in $O(m)$ time for the m rows that a pattern can span. The *LYpos* array is modified and shifted in $O(m)$ time. Then, the offset tree is traversed with $O(m \log \sigma)$ comparisons. Determining the actual occurrences of a pattern requires $O(m)$ time, proportional to the width of a pattern row.

Verification of a candidate row is done in $O(m \log \sigma)$ time. Overall, verification of a text block is done in time proportional to the uncompressed text block size, $O(m^2 \log \sigma)$. The verification process requires $O(m)$ space in addition to the $O(km)$ preprocessing space.

Complexity of Text Scanning Stage: Each block of text is processed separately in $O(m)$ space and in $O(m^2 \log \sigma)$ time. Since the text blocks are $O(m^2)$ in size, there are $O(n^2)/(m^2)$ blocks of text. Overall, $O(n^2 \log \sigma)$ time and $O(m)$ space are required to process text of uncompressed size $n \times n$.

5 Conclusion

We have developed the first strongly-inplace dictionary matching algorithm for 2D LZ78-compressed data. Our algorithm is for h-periodic patterns in which the period of each row is $\leq m/4$. The preprocessing time-complexity of our algorithm is optimal, as it is proportional to the uncompressed dictionary size. The text scanning stage searches for multiple patterns simultaneously, allowing the text

block to be decompressed and processed one row at a time. After information is gathered about the rows of a text block, potential pattern occurrences are identified and then verified in a single pass. Overall, our algorithm requires only $O(km)$ working space.

We would like to extend the algorithm to patterns with an aperiodic row or with a row whose period period $> m/4$. With such a row, many pattern rows with different 1D names can overlap in a text block row. Pattern preprocessing can focus on the first such row of each pattern and form an AC automaton of those rows. However, verification of the candidates requires a small-space 1D dictionary matching algorithm, which seems to be an open problem.

References

[1] Aho, A.V., Corasick, M.J.: Efficient string matching: an aid to bibliographic search. Commun. ACM 18(6), 333–340 (1975)
[2] Amir, A., Farach, M.: Two-dimensional dictionary matching. Inf. Process. Lett. 44(5), 233–239 (1992)
[3] Amir, A., Landau, G.M., Sokol, D.: Inplace 2d matching in compressed images. J. Algorithms 49(2), 240–261 (2003)
[4] Amir, A., Landau, G.M., Sokol, D.: Inplace run-length 2d compressed search. Theor. Comput. Sci. 290(3), 1361–1383 (2003)
[5] Baker, T.J.: A technique for extending rapid exact-match string matching to arrays of more than one dimension. SIAM J. Comp. (7), 533–541 (1978)
[6] Berman, P., Karpinski, M., Larmore, L.L., Plandowski, W., Rytter, W.: On the complexity of pattern matching for highly compressed two-dimensional texts. J. Comput. Syst. Sci. 65(2), 332–350 (2002)
[7] Bird, R.S.: Two dimensional pattern matching. Information Processing Letters 6(5), 168–170 (1977)
[8] Crochemore, M., Gasieniec, L., Hariharan, R., Muthukrishnan, S., Rytter, W.: A constant time optimal parallel algorithm for two-dimensional pattern matching. SIAM J. Comput. 27(3), 668–681 (1998)
[9] Idury, R.M., Schäffer, A.A.: Multiple matching of rectangular patterns. In: STOC 1993: Proceedings of the twenty-fifth annual ACM symposium on Theory of computing, pp. 81–90. ACM, New York (1993)
[10] Kida, T., Takeda, M., Shinohara, A., Miyazaki, M., Arikawa, S.: Multiple pattern matching in lzw compressed text. In: DCC 1998: Proceedings of the Conference on Data Compression, Washington, DC, USA, p. 103. IEEE Computer Society, Los Alamitos (1998)
[11] Knuth, D.E., Morris, J.H., Pratt, V.R.: Fast pattern matching in strings. SIAM J. Comput. 6(2), 323–350 (1977)
[12] Lothaire, M.: Applied Combinatorics on Words (Encyclopedia of Mathematics and its Applications). Cambridge University Press, New York (2005)
[13] Main, M.G., Lorentz, R.J.: An O(n log n) algorithm for finding all repetitions in a string. ALGORITHMS: Journal of Algorithms 5 (1984)
[14] Ziv, J., Lempel, A.: Compression of individual sequences via variable-rate coding. IEEE Transactions on Information Theory IT-24, 530–536 (1978)

Bidirectional Search in a String with Wavelet Trees

Thomas Schnattinger, Enno Ohlebusch, and Simon Gog

Institute of Theoretical Computer Science, University of Ulm, D-89069 Ulm
{Thomas.Schnattinger,Enno.Ohlebusch,Simon.Gog}@uni-ulm.de

Abstract. Searching for genes encoding microRNAs (miRNAs) is an important task in genome analysis. Because the secondary structure of miRNA (but not the sequence) is highly conserved, the genes encoding it can be determined by finding regions in a genomic DNA sequence that match the structure. It is known that algorithms using a bidirectional search on the DNA sequence for this task outperform algorithms based on unidirectional search. The data structures supporting a bidirectional search (affix trees and affix arrays), however, are rather complex and suffer from their large space consumption. Here, we present a new data structure called *bidirectional wavelet index* that supports bidirectional search with much less space. With this data structure, it is possible to search for RNA secondary structural patterns in large genomes, for example the human genome.

1 Introduction

It is now known that microRNAs (miRNAs) regulate the expression of many protein-coding genes and that the proper functioning of certain miRNAs is important for preventing cancer and other diseases. microRNAs are RNA molecules that are encoded by genes from whose DNA they are transcribed, but they are not translated into protein. Instead each primary transcript is processed into a secondary structure (consisting of approximately 70 nucleotides) called a pre-miRNA and finally into a functional miRNA. This so-called mature miRNA is 21-24 nucleotides long, so a gene encoding a miRNA is much longer than the processed mature miRNA molecule itself. Mature miRNA molecules are either fully or partially complementary to one or more messenger RNA (mRNA) molecules, and their main function is to down-regulate gene expression. The first miRNA was described by Lee et al. [1], but the term miRNA was only introduced in 2001 when the abundance of these tiny regulatory RNAs was discovered; see [2] for an overview. miRNAs are highly conserved during evolution, not on the sequence level, but as secondary structures. Thus, the task of finding the genes coding for a certain miRNA in a genome is to find all regions in the genomic DNA sequence that match its structural pattern. Because the structural pattern often consists of a hairpin loop and a stem (which may also have bulges), the most efficient algorithms first search for candidate regions matching the loop and then try to extend both ends by searching for complementary base pairs A–U, G–C, or G–U

A. Amir and L. Parida (Eds.): CPM 2010, LNCS 6129, pp. 40–50, 2010.

that form the stem. Because T (thymine) is replaced with U (uracil) in the transcription from DNA to RNA, one must search for the pairs A–T, G–C, or G–T in the DNA sequence. For example, if the loop is the sequence GGAC, then it is extended by one of the four nucleotides to the left or to the right, say by G to the right, and all regions in the DNA sequence matching GGACG are searched for (by forward search). Out of these candidate regions only those survive that can be extended by C or T to the left because only C and T (U, respectively) form a base pair with G, and the stem is formed by complementary base pairs. In other words, in the next step one searches for all regions in the DNA sequence matching either CGGACG or TGGACG (by backward search). Such a search strategy can be pursued only if bidirectional search is possible. Mauri and Pavesi [3] used affix trees for this purpose, while Strothmann [4] employed affix arrays.

Research on data structures supporting bidirectional search in a string started in 1995 with Stoye's diploma thesis on affix trees (the English translation appeared in [5]), and Maaß [6] showed that affix trees can be constructed on-line in linear time. Basically, the affix tree of a string S comprises both the suffix tree of S (supporting forward search) and the suffix tree of the reverse string S^{rev} (supporting backward search). It requires approximately $45n$ bytes, where n is the length of S. Strothmann [4] showed that affix arrays have the same functionality as affix trees, but they require only $18n$–$20n$ bytes (depending on the implementation). An affix array combines the suffix arrays of S and S^{rev}, but it is a complex data structure because the interplay between the two suffix arrays is rather difficult to implement. In this paper, we present a new data structure called *bidirectional wavelet index* that consists of the wavelet tree of the Burrows-Wheeler transformed string of S (supporting backward search) and the wavelet tree of the Burrows-Wheeler transformed string of S^{rev} (supporting forward search). In contrast to affix arrays, however, the interplay between the two is easy to implement. Our experiments show that the bidirectional wavelet index decreases the space requirement by a factor of 21 (compared to affix arrays), making it possible to search bidirectionally in very large strings.

2 Preliminaries

Let Σ be an ordered alphabet whose smallest element is the so-called sentinel character \$. If Σ consists of σ characters and is fixed, then we may view Σ as an array of size σ such that the characters appear in ascending order in the array $\Sigma[1..\sigma]$, i.e., $\Sigma[1] = \$ < \Sigma[2] < \ldots < \Sigma[\sigma]$. In the following, S is a string of length n over Σ having the sentinel character at the end (and nowhere else). For $1 \leq i \leq n$, $S[i]$ denotes the *character at position* i in S. For $i \leq j$, $S[i..j]$ denotes the *substring* of S starting with the character at position i and ending with the character at position j. Furthermore, S_i denotes the ith suffix $S[i..n]$ of S. The *suffix array* SA of the string S is an array of integers in the range 1 to n specifying the lexicographic ordering of the n suffixes of the string S, that is, it satisfies $S_{\mathsf{SA}[1]} < S_{\mathsf{SA}[2]} < \ldots < S_{\mathsf{SA}[n]}$; see Fig. 1 for an example. In the following, SA^{-1} denotes the inverse of the permutation SA. The suffix array was introduced by

i	SA	BWT	$S_{\mathsf{SA}[i]}$
1	19	n	$
2	3	l	_anele_lepanelen$
3	9	e	_lepanelen$
4	4	_	anele_lepanelen$
5	13	p	anelen$
6	8	l	e_lepanelen$
7	1	$	el_anele_lepanelen$
8	6	n	ele_lepanelen$
9	15	n	elen$
10	17	l	en$
11	11	l	epanelen$
12	2	e	l_anele_lepanelen$
13	7	e	le_lepanelen$
14	16	e	len$
15	10	_	lepanelen$
16	18	e	n$
17	5	a	nele_lepanelen$
18	14	a	nelen$
19	12	e	panelen$

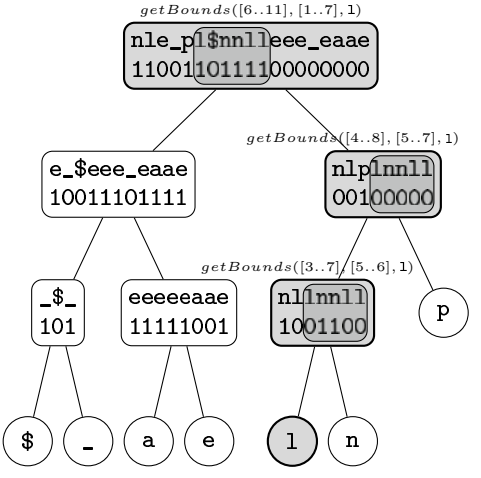


Fig. 1. Left: Suffix array and Burrows-Wheeler-transformed string BWT of string $S =$ `el_anele_lepanelen$`. Right: Conceptual illustration of the wavelet tree of the string $\mathsf{BWT} =$ `nle_pl$nnlleee_eaae`. Only the bit vectors are stored; the corresponding strings are shown for clarity. The shaded regions and the function *getBounds* will be explained later.

Manber and Myers [7]. In 2003, it was shown independently and contemporaneously by three research groups that a direct linear time construction of the suffix array is possible. To date, over 20 different suffix array construction algorithms are known; see [8] for details. Forward search on a suffix array can be done in $O(\log n)$ time per character by binary search; see [7].

Given the suffix array SA of a string S, the Burrows and Wheeler transformation $\mathsf{BWT}[1..n]$ of S is defined by $\mathsf{BWT}[i] = S[\mathsf{SA}[i] - 1]$ for all i with $\mathsf{SA}[i] \neq 1$ and $\mathsf{BWT}[i] = \$$ otherwise; see Fig. 1. In virtually all cases, the Burrows-Wheeler transformed string compresses much better than the original string; see [9]. The permutation LF, defined by $LF(i) = \mathsf{SA}^{-1}[\mathsf{SA}[i] - 1]$ for all i with $\mathsf{SA}[i] \neq 1$ and $LF(i) = 1$ otherwise, is called LF-mapping. Its inverse permutation is usually called ψ-function. Both LF and ψ can be represented more compactly than the suffix array. A compressed full-text index based on a compressed form of the LF-mapping is commonly referred to as FM-index [10]. If it is based on a compressed ψ-function it is usually called *compressed suffix array* [11]. The LF-mapping can be implemented by $LF(i) = C[c] + Occ(c, i)$ where $c = \mathsf{BWT}[i]$, $C[c]$ is the overall number (of occurrences) of characters in S which are strictly smaller than c, and $Occ(c, i)$ is the number of occurrences of the character c in $\mathsf{BWT}[1..i]$. Details about the Burrows and Wheeler transform and related topics can for instance be found in [12].

Algorithm 1. Given $c \in \Sigma$ and an ω-interval $[i..j]$, $backwardSearch(c, [i..j])$ returns the $c\omega$-interval if it exists, and $\perp$ otherwise.

```
backwardSearch(c, [i..j])
  i ← C[c] + Occ(c, i − 1) + 1
  j ← C[c] + Occ(c, j)
  if i ≤ j then return [i..j]
  else return ⊥
```

Ferragina and Manzini [10] showed that it is possible to search a pattern character-by-character backwards in the suffix array SA of string S, without storing SA. Backward search can be implemented such that each step takes only constant time, albeit a more space-efficient implementation takes $\mathcal{O}(\log \sigma)$ time; see below. In the following, the ω-interval in SA of a substring ω of S is the interval $[i..j]$ such that ω is a prefix of $S_{\mathsf{SA}[k]}$ for all $i \le k \le j$, but ω is not a prefix of any other suffix of S. For example, the `le`-interval in the suffix array of Fig. 1 is the interval [13..15]. Searching backwards in the string $S =$ `el_anele_lepanelen$` for the pattern `le` works as follows. By definition, backward search for the last character of the pattern starts with the ε-interval $[1..n]$, where ε denotes the empty string. Algorithm 1 shows the pseudo-code of one backward search step. In our example, $backwardSearch(\mathtt{e}, [1..19])$ returns the `e`-interval [6..11] because $C[\mathtt{e}] + Occ(\mathtt{e}, 1-1) + 1 = 5 + 0 + 1 = 6$ and $C[\mathtt{e}] + Occ(\mathtt{e}, 19) = 5 + 6 = 11$. In the next step, $backwardSearch(\mathtt{l}, [6..11])$ delivers the `le`-interval [13..15] because $C[\mathtt{l}] + Occ(\mathtt{l}, 6-1) + 1 = 11 + 1 + 1 = 13$ and $C[\mathtt{l}] + Occ(\mathtt{l}, 11) = 11 + 4 = 15$.

With the space-efficient *wavelet tree* introduced by Grossi et al. [13], each step of the backward search in string S takes $O(\log \sigma)$ time, as we shall see next. We say that an interval $[l..r]$ is an *alphabet interval*, if it is a subinterval of $[1..\sigma]$, where $\sigma = |\Sigma|$. For an alphabet interval $[l..r]$, the string $\mathsf{BWT}^{[l..r]}$ is obtained from the Burrows-Wheeler transformed string BWT of S by deleting all characters in BWT that do not belong to the subalphabet $\Sigma[l..r]$ of $\Sigma[1..\sigma]$. As an example, consider the string $\mathsf{BWT} =$ `nle_pl$nnlleee_eaae` and the alphabet interval [1..4]. The string $\mathsf{BWT}^{[1..4]}$ is obtained from `nle_pl$nnlleee_eaae` by deleting the characters `l`, `n`, and `p`. Thus, $\mathsf{BWT}^{[1..4]} =$ `e_$eee_eaae`.

The wavelet tree of the string BWT over the alphabet $\Sigma[1..\sigma]$ is a balanced binary search tree defined as follows. Each node v of the tree corresponds to a string $\mathsf{BWT}^{[l..r]}$, where $[l..r]$ is an alphabet interval. The root of the tree corresponds to the string $\mathsf{BWT} = \mathsf{BWT}^{[1..\sigma]}$. If $l = r$, then v has no children. Otherwise, v has two children: its left child corresponds to the string $\mathsf{BWT}^{[l..m]}$ and its right child corresponds to the string $\mathsf{BWT}^{[m+1..r]}$, where $m = \lfloor \frac{l+r}{2} \rfloor$. In this case, v stores a bit vector $B^{[l..r]}$ of size $r - l + 1$ whose i-th entry is 0 if the i-th character in $\mathsf{BWT}^{[l..r]}$ belongs to the subalphabet $\Sigma[l..m]$ and 1 if it belongs to the subalphabet $\Sigma[m+1..r]$. To put it differently, an entry in the bit vector is 0 if the corresponding character belongs to the left subtree and 1 if it

Algorithm 2. For a character c, an index i, and an alphabet interval $[l..r]$, the function $Occ'(c, i, [l..r])$ returns the number of occurrences of c in the string $\mathsf{BWT}^{[l..r]}[1..i]$, unless $l = r$ (in this case, it returns i).

```
Occ'(c, i, [l..r])
  if l = r then return i
  else
    m = ⌊(l+r)/2⌋
    if c ≤ Σ[m] then
      return Occ'(c, rank_0(B^[l..r], i), [l..m])
    else
      return Occ'(c, rank_1(B^[l..r], i), [m+1..r])
```

belongs to the right subtree; see Fig. 1. Moreover, each bit vector B in the tree is preprocessed such that the queries $rank_0(B, i)$ and $rank_1(B, i)$ can be answered in constant time [14], where $rank_b(B, i)$ is the number of occurrences of bit b in $B[1..i]$. Obviously, the wavelet tree has height $O(\log \sigma)$. Because in an actual implementation it suffices to store only the bit vectors, the wavelet tree requires only $n \log \sigma$ bits of space plus $o(n \log \sigma)$ bits for the data structures that support rank-queries in constant time.

The query $Occ(c, i)$ can be answered by a top-down traversal of the wavelet tree in $O(\log \sigma)$ time. As an example, we compute $Occ(\mathtt{e}, 16)$ on the wavelet tree of the string $\mathsf{BWT} = \mathtt{nle\_pl\$nnlleee\_eaae}$ from Fig. 1. Because `e` belongs to the first half $\Sigma[1..4]$ of the ordered alphabet Σ, the occurrences of `e` correspond to zeros in the bit vector at the root, and they go to the left child, say node v_1, of the root. Now the number of `e`'s in $\mathsf{BWT}^{[1..7]} = \mathtt{nle\_pl\$nnlleee\_eaae}$ up to position 16 equals the number of `e`'s in the string $\mathsf{BWT}^{[1..4]} = \mathtt{e\_\$eee\_eaae}$ up to position $rank_0(B^{[1..7]}, 16)$. So we compute $rank_0(B^{[1..7]}, 16) = 8$. Because `e` belongs to the second quarter $\Sigma[3..4]$ of Σ, the occurrences of `e` correspond to ones in the bit vector at node v_1, and they go to the right child, say node v_2, of v_1. The number of `e`'s in $\mathsf{BWT}^{[1..4]} = \mathtt{e\_\$eee\_eaae}$ up to position 8 is equal to the number of `e`'s in $\mathsf{BWT}^{[3..4]} = \mathtt{eeeeeaae}$ up to position $rank_1(B^{[1..4]}, 8) = 5$. In the third step, we must go to the right child of v_2, and the number of `e`'s in $\mathsf{BWT}^{[3..4]} = \mathtt{eeeeeaae}$ up to position 5 equals the number of `e`'s in $\mathsf{BWT}^{[4..4]} = \mathtt{eeeeee}$ up to position $rank_1(B^{[3..4]}, 5) = 5$. Since $\mathsf{BWT}^{[4..4]}$ consists solely of `e`'s (by the way, that is the reason why it does not appear in the wavelet tree) the number of `e`'s in $\mathsf{BWT}^{[4..4]}$ up to position 5 is 5. Pseudo-code for the computation of $Occ(c, i) = Occ'(c, i, [1..\sigma])$ can be found in Algorithm 2.

3 Bidirectional Search

The *bidirectional wavelet index* of a string S consists of

- the *backward index*, supporting backward search based on the wavelet tree of the Burrows-Wheeler transformed string BWT of S, and

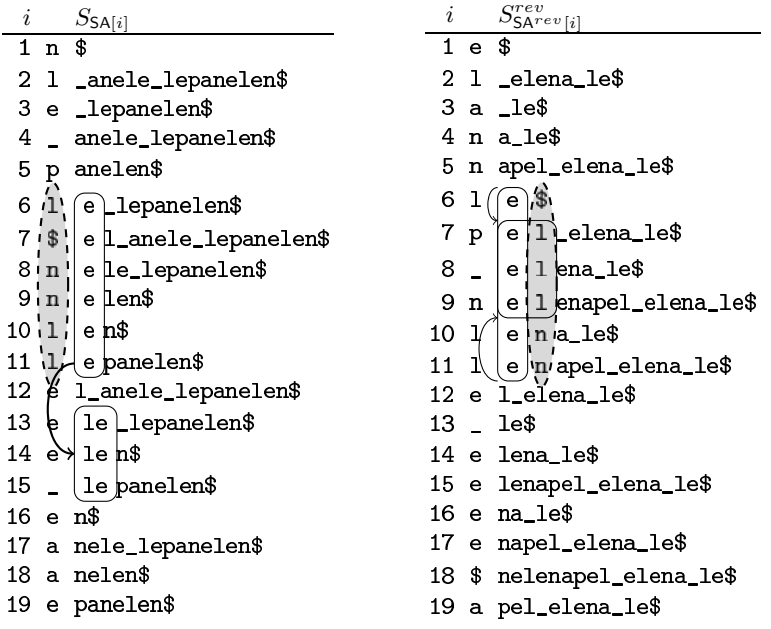


Fig. 2. Bidirectional wavelet index of $S =$ `el_anele_lepanelen$`

– the *forward index*, supporting backward search on the reverse string S^{rev} of S (hence forward search on S) based on the wavelet tree of the Burrows-Wheeler transformed string BWT^{rev} of S^{rev}.

The difficult part is to synchronize the search on both indexes. To see this, suppose we know the ω-interval $[i..j]$ in the backward index as well as the ω^{rev}-interval $[i^{rev}..j^{rev}]$ in the forward index, where ω is some substring of S. Given $[i..j]$ and a character c, $backwardSearch(c, [i..j])$ returns the $c\omega$-interval in the backward index (cf. Algorithm 1), but it is unclear how the corresponding interval, the interval of the string $(c\omega)^{rev} = \omega^{rev}c$, can be found in the forward index. Vice versa, given $[i^{rev}..j^{rev}]$ and a character c, backward search returns the $c\omega^{rev}$-interval in the forward index, but it is unclear how the corresponding ωc-interval can be found in the backward index. Because both cases are symmetric, we will only deal with the first case. So given the ω^{rev}-interval, we have to find the $\omega^{rev}c$-interval in the forward index. As an example, consider the bidirectional wavelet index of the string $S =$ `el_anele_lepanelen$` in Fig. 2, and the substring $\omega =$ `e` $= \omega^{rev}$. The `e`-interval in both indexes is [6..11]. The `le`-interval in the backward index is determined by $backwardSearch(\texttt{l}, [6..11]) = [13..15]$ and the task is to identify the `el`-interval in the forward index.

All we know is that the suffixes of S^{rev} are lexicographically ordered in the forward index. In other words, the $\omega^{rev}c$-interval $[p..q]$ is a subinterval of $[i^{rev}..j^{rev}]$ such that (note that $|\omega^{rev}| = |\omega|$)

- $S^{rev}[\mathsf{SA}^{rev}[k] + |\omega|] < c$ for all k with $i^{rev} \leq k < p$,
- $S^{rev}[\mathsf{SA}^{rev}[k] + |\omega|] = c$ for all k with $p \leq k \leq q$,
- $S^{rev}[\mathsf{SA}^{rev}[k] + |\omega|] > c$ for all k with $q < k \leq j^{rev}$.

In the example of Fig. 2,

- $S^{rev}[\mathsf{SA}^{rev}[k] + 1] = \texttt{\$} < \texttt{l}$ for $k = 6$,
- $S^{rev}[\mathsf{SA}^{rev}[k] + 1] = \texttt{l}$ for all k with $7 \leq k \leq 9$,
- $S^{rev}[\mathsf{SA}^{rev}[k] + 1] = \texttt{n} > \texttt{l}$ for all k with $9 < k \leq 11$.

Unfortunately, we do not know these characters, but if we would know the number *smaller* of all occurrences of characters at these positions that precede c in the alphabet and the number *greater* of all occurrences of characters at these positions that follow c in the alphabet, then we could identify the unknown $\omega^{rev}c$-interval $[p..q]$ by $p = i^{rev} + smaller$ and $q = j^{rev} - greater$. In our example, the knowledge of $smaller = 1$ and $greater = 2$ would yield the $\texttt{el}$-interval $[6 + 1..11 - 2] = [7..9]$. The *key observation* is that the multiset of characters

$$\{S^{rev}[\mathsf{SA}^{rev}[k] + |\omega|] : i^{rev} \leq k \leq j^{rev}\}$$

coincides with the multiset $\{\mathsf{BWT}[k] : i \leq k \leq j\}$. In the example of Fig. 2,

$$\{S^{rev}[\mathsf{SA}^{rev}[k]+1] : 6 \leq k \leq 11\} = \{\ \texttt{\$}, \texttt{l}, \texttt{l}, \texttt{l}, \texttt{n}, \texttt{n}\} = \{\mathsf{BWT}[k] : 6 \leq k \leq 11\}$$

In other words, it suffices to determine the numbers *smaller* and *greater* of all occurrences of characters in the string $\mathsf{BWT}[i..j]$ that precede and follow character c in the alphabet Σ. And this task can be accomplished by a top-down traversal of the wavelet tree of BWT. The procedure is similar to the implementation of $Occ(c, i)$ as explained above. As an example, we compute the values of *smaller* and *greater* for the interval [6..11] and the character $\texttt{l}$. This example is illustrated in Fig. 1. Because $\texttt{l}$ belongs to the second half $\Sigma[5..7]$ of the ordered alphabet Σ, the occurrences of $\texttt{l}$ correspond to ones in the bit vector at the root, and they go to the right child, say node v_1, of the root. In order to compute the number of occurrences of characters in the interval [6..11] that belong to $\Sigma[1..4]$ and hence are in the left child of the root, we compute

$$(a_0, b_0) = (rank_0(B^{[1..7]}, 6 - 1), rank_0(B^{[1..7]}, 11)) = (2, 3)$$

and the number we are searching for is $b_0 - a_0 = 3 - 2 = 1$. Then we descend to the right child v_1 and have to compute the boundaries of the search interval in the bit vector $B^{[5..7]}$ that corresponds to the search interval [6..11] in the bit vector $B^{[1..7]}$. These boundaries are $a_1 + 1$ and b_1, where

$$(a_1, b_1) = (rank_1(B^{[1..7]}, 6 - 1), rank_1(B^{[1..7]}, 11)) = (3, 8)$$

Proceeding recursively, we find that $\texttt{l}$ belongs to the third quarter $\Sigma[5..6]$ of Σ, so the occurrences of $\texttt{l}$ correspond to zeros in the bit vector at v_1, and they go to the left child, say node v_2, of v_1. Again, we compute

Algorithm 3. Given a BWT-interval $[i..j]$, an alphabet-interval $[l..r]$, and $c \in \Sigma$, $getBounds([i..j],[l..r],c)$ returns the pair $(smaller, greater)$, where $smaller$ ($greater$) is the number of all occurrences of characters from the subalphabet $\Sigma[l..r]$ in $\mathsf{BWT}[i..j]$ that are smaller (greater) than c.

```
getBounds([i..j],[l..r],c)
  if l = r then return (0,0)
  else
    (a0,b0) ← (rank0(B^[l..r],i−1), rank0(B^[l..r],j))
    (a1,b1) ← (i−1−a0, j−b0)
    /* (a1,b1) = (rank1(B^[l..r],i−1), rank1(B^[l..r],j)) */
    m = ⌊(l+r)/2⌋
    if c ≤ Σ[m] then
      (smaller,greater) ← getBounds([a0+1..b0],[l..m],c)
      return (smaller, greater+b1−a1)
    else
      (smaller,greater) ← getBounds([a1+1..b1],[m+1..r],c)
      return (smaller+b0−a0, greater)
```

$$(a'_0, b'_0) = (rank_0(B^{[5..7]}, 4-1), rank_0(B^{[5..7]}, 8)) = (2, 7)$$
$$(a'_1, b'_1) = (rank_1(B^{[5..7]}, 4-1), rank_1(B^{[5..7]}, 8)) = (1, 1)$$

The number of occurrences of characters in the string $\mathsf{BWT}^{[3..8]}$ that belong to $\Sigma[7] = \mathtt{p}$ is $b'_1 - a'_1 = 1 - 1 = 0$ and the new search interval in the bit vector $B^{[5..6]}$ is $[a'_0 + 1..b'_0] = [3..7]$. In the third step, we compute

$$(a''_0, b''_0) = (rank_0(B^{[5..6]}, 3-1), rank_0(B^{[5..6]}, 7)) = (1, 4)$$
$$(a''_1, b''_1) = (rank_1(B^{[5..6]}, 3-1), rank_1(B^{[5..6]}, 7)) = (1, 3)$$

and find that there are $b''_1 - a''_1 = 2$ occurrences of the character `n` and $b''_0 - a''_0 = 3$ occurrences of the character `l`. In summary, during the top-down traversal, we found that in the string $\mathsf{BWT}[6..11]$ there is one character smaller than `l` (so $smaller = 1$), there are two characters greater than `l` (so $greater = 2$), and three characters coincide with `l`. Pseudo-code for the computation of $(smaller, greater) = getBounds([i..j], [1..\sigma], c)$ can be found in Algorithm 3.

4 Experimental Results

An implementation of the bidirectional wavelet index is available under the GNU General Public License at `http://www.uni-ulm.de/in/theo/research/seqana`. To assess the performance of our new data structure, we used it to search for RNA secondary structures in large DNA sequences. We adopted the depth-first search method described in [4]; for space reasons, it is not repeated here. The following RNA secondary structures are also taken from [4]:

Table 1. Comparison of the running times (in seconds) of the searches for the six RNA structural patterns in the human DNA sequence (about one billion nucleotides). The numbers in parentheses below the pattern names are the numbers of matches found. Index ① is our new bidirectional wavelet index, Index ② consists of the suffix array SA of S (supporting binary search in the forward direction), and the wavelet tree of the Burrows-Wheeler transformed string of S (supporting backward search). Index ③ is similar to Index ②, but SA is replaced with a compressed suffix array.

Index	MB	hairpin1 (2343)	hairpin2 (286)	hairpin4 (3098)	hloop(5) (14870)	acloop(5) (294)	acloop(10) (224)
①	799	11.053	0.079	0.792	28.373	0.958	0.420
②	4408	8.855	0.041	0.365	22.208	0.781	0.336
③	1053	137.371	0.651	5.642	345.860	12.174	6.381

1. `hairpin1 = (stem:=N{20,50}) (loop:=NNN) ^stem`
2. `hairpin2 = (stem:=N{10,50}) (loop:=GGAC) ^stem`
3. `hairpin4 = (stem:=N{10,15}) (loop:=GGAC[1]) ^stem`
4. `hloop(length) = (stem:=N{15,20}) (loop:=N{length}) ^stem`
5. `acloop(length) = (stem:=N{15,20}) (loop:=(A|C){length}) ^stem`

The symbol `N` is used as a wildcard matching any nucleotide. The first pattern describes a hairpin structure with an apical loop consisting of three nucleotides. On the left and right hand sides of the loop are two reverse complementary sequences, each consisting of 20 - 50 nucleotides. The second pattern describes a similar structure, where the loop must be the sequence `GGAC`. The `[1]` in the third pattern means that one nucleotide can be inserted at any position, i.e., the loop is one of the sequences `GGAC`, `NGGAC`, `GNGAC`, `GGNAC`, `GGANC` or `GGACN`. In the last two patterns `length` denotes the length of the loop sequence. For example, in the pattern `acloop(5)` the loop consists of five nucleotides, each of which must either be `A` or `C`. In the experiments reported in Table 1, we searched for six patterns in the first five chromosomes of the human genome.[1] The concatenation of the DNA sequences of these five chromosomes is called "human DNA sequence" in the following; it constitutes about one third of the whole genome (one billion nucleotides). All experiments were conducted on a PC with a *Dual-Core AMD Opteron 8218* processor (2,6 GHz) and 8 GB main memory. Unfortunately, the implementations of affix trees/arrays [3,4] are currently not available.[2] For this reason, one cannot compare the running times. (We conjecture, however, that our method outperforms the affix array method.) Nevertheless, we can say something about the space consumption. According to Strothmann [4], an affix array requires 18 bytes per nucleotide, so approximately 16.8 GB for the human DNA sequence. The bidirectional wavelet index (index ①) takes only 799 MB; see Table 1. Hence it decreases the space requirement by a factor of 21.

[1] `http://hgdownload.cse.ucsc.edu/goldenPath/hg19/bigZips/chromFa.tar.gz`
[2] However, a reimplementation of the affix array method is under way.

Due to the lack of affix tree/array implementations, we compared our method with the following two approaches to support bidirectional search. First, we combined the two well-known data structures supporting forward search (the suffix array SA of string S) and backward search (the wavelet tree of the Burrows-Wheeler transformed string BWT of S), and obtained an index (index ②) which also supports bidirectional search. Because both data structures deliver intervals of the suffix array, the two searches can directly be combined without synchronization. Interestingly enough, in the technical literature this natural approach has not been considered yet, i.e., it is new as well. Table 1 shows that index ② takes 4.4 GB for the human DNA sequence. Second, to reduce the space consumption even more, we replaced the suffix array in index ② by a compressed suffix array (CSA) which also supports binary search in the forward direction, yielding index ③. This reduces the memory consumption by another factor of 4, but slows down the running time by a factor of 15.5 (compared with index ②); see Table 1. This is because the CSA must frequently recover SA-values from its sampled SA-values (in our implementation every twelveth value is stored; more samples would decrease the running time, but increase the memory requirements). By contrast, the time-space trade-off of our bidirectional wavelet index ① is much better: it reduces the space consumption by a factor of 5.5, but it is only 1.2 - 2.2 time slower than index ②. This can be attributed to the fact that SA-values are solely needed to output the positions of the matching regions in the string S (in our implementation a hundredth of all SA-values is stored).

References

1. Lee, R., Feinbaum, R., Ambros, V.: The C. elegans heterochronic gene lin-4 encodes small RNAs with antisense complementarity to lin-14. Cell 75(5), 843–854 (1993)
2. Kim, N., Nam, J.W.: Genomics of microRNA. TRENDS in Genetics 22(3), 165–173 (2006)
3. Mauri, G., Pavesi, G.: Pattern discovery in RNA secondary structure using affix trees. In: Baeza-Yates, R., Chávez, E., Crochemore, M. (eds.) CPM 2003. LNCS, vol. 2676, pp. 278–294. Springer, Heidelberg (2003)
4. Strothmann, D.: The affix array data structure and its applications to RNA secondary structure analysis. Theoretical Computer Science 389, 278–294 (2007)
5. Stoye, J.: Affix trees. Technical report 2000-04, University of Bielefeld (2000)
6. Maaß, M.: Linear bidirectional on-line construction of affix trees. Algorithmica 37, 43–74 (2003)
7. Manber, U., Myers, E.: Suffix arrays: A new method for on-line string searches. SIAM Journal on Computing 22(5), 935–948 (1993)
8. Puglisi, S., Smyth, W., Turpin, A.: A taxonomy of suffix array construction algorithms. ACM Computing Surveys 39(2), 1–31 (2007)
9. Burrows, M., Wheeler, D.: A block-sorting lossless data compression algorithm. Research Report 124, Digital Systems Research Center (1994)
10. Ferragina, P., Manzini, G.: Opportunistic data structures with applications. In: Proc. IEEE Symposium on Foundations of Computer Science, pp. 390–398 (2000)
11. Grossi, R., Vitter, J.: Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In: Proc. ACM Symposium on the Theory of Computing, pp. 397–406. ACM Press, New York (2000)

12. Manzini, G.: The Burrows-Wheeler Transform: Theory and practice. In: Kutyłowski, M., Wierzbicki, T., Pacholski, L. (eds.) MFCS 1999. LNCS, vol. 1672, pp. 34–47. Springer, Heidelberg (1999)
13. Grossi, R., Gupta, A., Vitter, J.: High-order entropy-compressed text indexes. In: Proc. 14th ACM-SIAM Symposium on Discrete Algorithms, pp. 841–850 (2003)
14. Jacobson, G.: Space-efficient static trees and graphs. In: Proc. 30th Annual Symposium on Foundations of Computer Science, pp. 549–554. IEEE, Los Alamitos (1989)

A Minimal Periods Algorithm with Applications

Zhi Xu

The University of Western Ontario, Department of Computer Science,
Middlesex College, London, Ontario, Canada N6A 5B7
zhi_xu@csd.uwo.ca

Abstract. Kosaraju in "Computation of squares in a string" briefly described a linear-time algorithm for computing the minimal squares starting at each position in a word. Using the same construction of suffix trees, we generalize his result and describe in detail how to compute the minimal α power, with a period of length longer than s, starting at each position in a word w for arbitrary exponent $\alpha > 1$ and integer $s \geq 0$. The algorithm runs in $O(\alpha|\,w\,|)$-time for $s = 0$ and in $O(|\,w\,|^2)$-time otherwise. We provide a complete proof of the correctness and computational complexity of the algorithm. The algorithm can be used to detect certain types of pseudo-patterns in words, which was our original goal in studying this generalization.

1 Introduction

A word of the form ww is a *square*, which is the simplest type of repetition. Study on repetitions in words occurred as early as Thue's work [23] in the early 1900's. There are many works in the literature on finding repetitions (*periodicities*), an important topic in combinatorics on words. In 1983, Slisenko [21] described a linear-time algorithm for finding all syntactically distinct maximal repetitions in a word. Crochemore [5], Main and Lorentz [17] described linear-time algorithms for testing whether a word contains any square and thus for testing whether a word contains any repetition. Since a word of length n may have $\Omega(n^2)$-many square factors (such as word 0^n), only primitively-rooted or maximal repetitions are ordinarily considered. Crochemore [4] described an $O(n \log n)$-time algorithm for finding all maximal primitively-rooted integer repetitions, where maximal means that some kth power cannot be extended in either direction to obtain the $(k+1)$th power. The $O(n \log n)$-time is optimal since a word of length n may have $\Omega(n \log n)$-many primitively-rooted repetitions (such as Fibonacci words). Apostolico and Preparata [1] described an $O(n \log n)$-time algorithm for finding all right-maximal repetitions. Main and Lorentz [16] described an $O(n \log n)$-time algorithm for finding all maximal repetitions. Gusfield and Stoye [22,10] described several algorithms for finding repetitions. Both the number of distinct squares [8,12] and the number of maximal repetitions (*runs*) [14] in a word are in $O(n)$. This fact suggests the existence of linear-time algorithms on distinct (or maximal) repetitions. Main [18] described a linear-time algorithm for finding all left-most occurrences of distinct maximal repetitions. Kolpakov and

A. Amir and L. Parida (Eds.): CPM 2010, LNCS 6129, pp. 51–62, 2010.

Kucherov [14] described a linear-time algorithm for finding all occurrences of maximal repetitions. See the paper [6] for the most recent survey of the topic.

Rather than considering repetitions from a global point of view, there are works considering repetitions from a local point of view. In a five-pages extended abstract, Kosaraju [15] briefly described a linear-time algorithm for finding the minimal square when starting at each position in a word. In the same vein, Duval, Kolpakov, Kucherov, Lecroq, and Lefebvre [7] described a linear-time algorithm for finding the local periods (of squares) centered at each position in a word. Since there may be $\Omega(\log n)$ primarily-rooted maximal repetitions starting at the same position (for example, consider the left-most position in a Fibonacci word), the local approach cannot achieve the same efficiency by directly applying linear-time algorithms on finding maximal repetitions.

In this paper, we generalize Kosaraju's algorithm [15] for computing minimal squares. Instead of squares, we discuss arbitrary (fractional) powers. Based on a proper modification of Kosaraju's algorithm, we use the same techniques of Weiner's algorithm for suffix-tree construction and lowest-common-ancestor query algorithms, and describe in detail an algorithm: the algorithm takes an arbitrary rational number $\alpha > 1$ and integer $s \geq 0$, starts at each position in a word w, and finds the minimal α power with a period of length longer than s. This algorithm runs in $O(\alpha|\, w\,|)$-time for $s = 0$ and in $O(|\, w\,|^2)$-time for arbitrary s. In this paper, we provide a complete proof of the correctness and computational complexity of the modified algorithm. In concluding, we show how this algorithm can be used to detect certain types of pseudo-patterns in words, which was our original goal in studying this algorithm.

2 Preliminary

We assume the alphabet Σ is fixed throughout this paper. Let $w = a_1 a_2 \cdots a_n$ be a word. The *length* $|\, w\,|$ of w is n. A *factor* $w[p\,..\,q]$ of w is the word $a_p a_{p+1} \cdots a_q$ if $1 \leq p \leq q \leq n$; otherwise $w[p\,..\,q]$ is the *empty word* ϵ. In particular, $w[1\,..\,q]$ and $w[p\,..\,n]$ are called a *prefix* and a *suffix*, respectively. The *reverse* of w is the word $w^R = a_n \cdots a_2 a_1$. Word w is called an α *power* for rational number $\alpha > 1$ if $w = x^k y$ for some words x, y and integer k such that $x \neq \epsilon$, y is a prefix of x, and $\alpha = k + \frac{|y|}{|x|}$, where α is called the *exponent* and x is called the *period*; we also write $w = x^\alpha$ in this case. The 2nd power and the 3rd power are called the *square* and the *cube*, respectively.

The *prefix period* (resp., *strict prefix period*) of a word w with respect to exponent α and integer s is the shortest word x of length $|\, x\,| > s$, such that x^β is a prefix of w for some exponent $\beta \geq \alpha$ (resp., $\beta = \alpha$). We denote the length of the prefix period by $pp_s^\alpha(w)$, if there is one, or otherwise $pp_s^\alpha(w) = +\infty$. For example, if word $w = \texttt{0100101001}$, then $pp_0^2(w) = 3$, $pp_0^3(w) = +\infty$, $pp_3^2(w) = 5$, $pp_0^{3/2}(w) = pp_0^{5/4}(w) = 2$, and the length of the strict prefix period of w with respect to exponent $5/4$ and integer 0 is 8. By definition, $pp_{s_1}^{p_1}(w) \leq pp_{s_2}^{p_2}(w)$ for $s_1 \leq s_2, p_1 \leq p_2$. Furthermore, the following lemma holds naturally.

Lemma 1. *Let $\alpha > 1$ be a rational number, $s \geq 0$ be an integer, and word u be a prefix of word v. (1) If $pp_s^\alpha(u) \neq +\infty$, then $pp_s^\alpha(v) = pp_s^\alpha(u)$. (2) If $pp_s^\alpha(v) \neq +\infty$ and $|\,u\,| \geq \alpha \cdot pp_s^\alpha(v)$, then $pp_s^\alpha(u) = pp_s^\alpha(v)$; otherwise, $pp_s^\alpha(u) = +\infty$. The lemma also holds when the length of the prefix period pp_s^α is replaced by the length of the strict prefix period.*

The *right minimal period array* of a word w is defined with respect to the exponent α and the integer s as $rmp_s^\alpha(w)[i] = pp_s^\alpha(w[i\,..\,n])$, and the *left minimal period array* is defined as $lmp_s^\alpha(w)[i] = pp_s^\alpha(w[1\,..\,i]^R)$, for $1 \leq i \leq n$. For example, if word $w = \texttt{0100101001}$, then $rmp_0^2(w) = [\ 3, +\infty, 1, 2, 2, +\infty, +\infty, 1, +\infty, +\infty\ ]$ and $rmp_1^{3/2}(w) = [\ 2, 3, 5, 2, 2, 2, +\infty, +\infty, +\infty, +\infty\ ]$ (in the non-strict sense).

A *suffix tree* $\mathcal{T}_w$ for a word $w = w[1\,..\,n]$ is a rooted tree with each edge labeled with a non-empty word that satisfies the following conditions.

1. All internal nodes excluding the root have at least two children;
2. labels on edges from the same node begin with different letters;
3. there are exactly n leaves, denoted by $leaf_i$, and $\tau(leaf_i) = w[i\,..\,n]\$$,

where function τ maps each node v to the word obtained by concatenating the labels along the path from the root to v, and \$ is a special letter not in the alphabet of w. By definition, a suffix tree for a word is unique up to renaming nodes and reordering among children. For more details on suffix tree, see the book [9, Chap. 5–9].

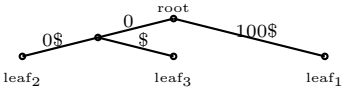


Fig. 1. The suffix tree for 100

We denote by $p(v)$, or more specifically by $p_{\mathcal{T}_w}(v)$, the *father* of node v in the tree $\mathcal{T}_w$. The concepts *ancestor* and *descendent* are defined as usual. If node x is a common ancestor of nodes y and z in $\mathcal{T}_w$, by the definition of a suffix tree, $\tau(x)$ is a common prefix of $\tau(y)$ and $\tau(z)$. We define function δ as $\delta(v) = |\,\tau(v)\,|$.

We denote by $\mathrm{lca}(u, v)$ the *lowest common ancestor* of nodes u and v in a tree such that any other common ancestor of u and v is an ancestor of $\mathrm{lca}(u, v)$. After a linear-time preprocessing on the tree, the lowest common ancestor of any pair of nodes can be found in constant-time [11,20].

Lemma 2. *Let $leaf_i, leaf_j$, $i > j$, be leaves in $\mathcal{T}_w$. The word on the edge from $p(leaf_i)$ to $leaf_i$ is not longer than the word on the edge from $p(leaf_j)$ to $leaf_j$.*

A suffix tree for a word can be constructed in linear-time [25,19,24]. Both Kosaraju's algorithm [15] for computing $rmp_0^2(w)$ and our generalization for computing $rmp_s^\alpha(w)$ for arbitrary $\alpha > 1$ and $s \geq 0$ are based on Weiner's algorithm [25]. Consequently, we briefly describe it here (see Fig. 2). Weiner's

algorithm constructs suffix tree $\mathcal{T}_w$ by adding $leaf_n, \ldots, leaf_2, leaf_1$ into a suffix tree incrementally. After each extension by $leaf_i$, the new tree is precisely the suffix tree $\mathcal{T}_{w[i\,..\,n]}$. By using indicator vectors and inter-node links, the total time to locate positions y at Lines 7–8 is in $O(n)$. We omit the details of the method for locating y because it is not quite relevant.

```
   Input: a word w = w[1..n].
   Output: the suffix tree T_w.
 1 begin function make_suffix_tree(w)
 2 |   construct T_n = T_{w[n..n]} ;
 3 |   for i from n − 1 to 1 do  T_i ⟵ extend(T_{i+1}, w[i..n]) ; // T_i = T_{w[i..n]}
 4 |   return T_1 ;
 5 end
 6 begin function extend(tree, word[i..n]) // we assume tree = T_{word[i+1..n]}
 7 |   find the proper position y in tree to insert the new node leaf_i ;
 8 |   if needed, split an edge x → z to x → y, y → z by adding a new node y ;
 9 |   create and label the edge y → leaf_i by word[i + |τ(y)| .. n]$ ;
10 end
```

Fig. 2. Framework of Weiner's algorithm for constructing suffix tree

3 The Algorithm for Computing $rmp_s^\alpha(w)$ and $lmp_s^\alpha(w)$

First, we show how to compute both non-strict and strict prefix periods from the suffix tree $\mathcal{T}_w$ in $O\left(\frac{|\,w\,|}{\min\{s, pp_0^\alpha(w)\}}\right)$-time. Although in the worst case the time can be in $\Theta(|\,w\,|)$, when both s and $pp_0^\alpha(w)$ are in $\Omega(|\,w\,|)$, the time does not depend on $|\,w\,|$, which is one of the essential reasons that the time of computing $rmp_0^\alpha(w)$ and $lmp_0^\alpha(w)$ is linear in $|\,w\,|$.

Lemma 3. *Let $\alpha > 1$ be a rational number, $s \geq 0$ be an integer, and $\mathcal{T}_w$ be the suffix tree for a word w. Then $pp_s^\alpha(w)$ can be computed in $O\left(\frac{|\,w\,|}{\min\{s, pp_0^\alpha(w)\}}\right)$-time, even for the strict prefix period case.*

Proof. There is an $O\left(\frac{|\,w\,|}{\min\{s, pp_0^\alpha(w)\}}\right)$-time algorithm (see Fig. 3) to compute $pp_s^\alpha(w)$. First, along the path from $leaf_1$ to the root, we find the highest ancestor h of $leaf_1$ such that $\delta(h) \geq (\alpha - 1)(s + 1)$. Second, we find the lowest common ancestor of $leaf_1$ and every $leaf_i$, $i > s + 1$, that is a descendent of h and check whether the inequality

$$\delta(\mathrm{lca}(leaf_1, leaf_i)) \geq (\alpha - 1)(i - 1) \tag{1}$$

holds. If no such $leaf_i$ satisfies (1), then $pp_s^\alpha(w) = +\infty$; otherwise, $pp_s^\alpha(w) = i - 1$, where i is the smallest i such that $leaf_i$ satisfies (1).

To prove correctness, we observe that $w = x^\beta y$ for some non-empty word x and $\beta \geq \alpha$ if, and only if, the common prefix of $w[1\,..\,n]$ and $w[|\,x\,| + 1\,..\,n]$ is of length

Input: a suffix tree $tree = \mathcal{T}_{w[1\,..\,n]}$ and two integers $s \geq 0$, $\alpha > 1$.
Output: the length of the prefix period $pp_s^\alpha(w)$.

```
1  begin function compute_pp(tree, s, α)
2      if α(s+1) > n then return +∞ ; else h ⟵ leaf_1 ;
3      while δ(p(h)) ≥ (α−1)(s+1) do h ⟵ p(h) ;
4      pp ⟵ +∞ ;
5      preprocessing the tree rooted at h for constant-time lca operation ;
6      foreach leaf_i being a descendent of h other than leaf_1 do
7          if δ(lca(leaf_1, leaf_i)) ≥ (α−1)(i−1) and i−1 > s then
8              if pp > i−1 then pp ⟵ i−1 ; // w[1..i−1] is a period
9      return pp ;
10 end
```

Fig. 3. Algorithm for computing $pp_s^\alpha(w)$, using the suffix tree $\mathcal{T}_w$

at least $\lceil (\alpha-1)|\,x\,|\rceil$, which means $leaf_{|\,x\,|+1}$ satisfies (1). Furthermore, such x satisfies $|\,x\,| > s$ only if $leaf_{|\,x\,|+1}$ satisfies $\delta(\mathrm{lca}(leaf_1, leaf_{|\,x\,|+1})) \geq (\alpha-1)(s+1)$, which means $leaf_{|\,x\,|+1}$ is a descendent of h. The minimal length of such a period, if any, is returned and correctness is ensured.

Let us turn to the computational complexity. Let T_h be the sub-tree rooted at h and let l be the number of leaves in T_h. By the definition of a suffix tree, each internal node has at least two children, and thus the number of internal nodes in T_h is less than l. Therefore, the time cost of the algorithm is linear in l. Now we prove $l \leq 1 + \frac{n}{\min\{s+1, pp_0^\alpha(w)\}}$ by contradiction. Suppose $l > 1 + \frac{n}{\min\{s+1, pp_0^\alpha(w)\}}$ and $leaf_{i_1}, leaf_{i_2}, \ldots, leaf_{i_l}$ are leaves of T_h. Then, there are l-many factors of length $t = (\alpha-1)(s+1)$ such that $w[i_1\,..\,i_1+t-1] = w[i_2\,..\,i_2+t-1] = \cdots = w[i_l\,..\,i_l+t-1]$. Since $1 \leq i_j \leq n$ for $1 \leq j \leq l$, the pigeon hole principle guarantees two indices, say i_1 and i_2, such that $0 \leq i_2 - i_1 \leq \frac{n}{l-1} < \min\{s+1, pp_0^\alpha(w)\}$. Then the common prefix of $w[i_1\,..\,n]$ and $w[i_2\,..\,n]$ is of length at least $t = (\alpha-1)(s+1) > (\alpha-1)(i_2-i_1)$, which means there is a prefix of $w[i_1\,..\,i_1+t-1] = w[i_2\,..\,i_2+t-1] = w[1\,..\,t-1]$ that is an α power with period of length $i_2 - i_1$, which contradicts $i_2 - i_1 < pp_0^\alpha(w)$. Therefore, the number of leaves in T_h is $l \leq \frac{n}{\min\{s+1, pp_0^\alpha(w)\}} + 1$ and the algorithm runs in $O\left(\frac{n}{\min\{s, pp_0^\alpha(w)\}}\right)$-time.

For the strict prefix period, we add an extra condition "$i-1 \bmod den = 0$" to the **if**-statement in Line 7 to check whether the length of a candidate period is a multiple of den for $\alpha = num/den, \gcd(num, den) = 1$. This condition ensures that the period for the exponent α is strict. □

For a word $w = w[1\,..\,n]$, the left minimal period array and the right minimal period array satisfy $lmp_s^\alpha(w)[i] = rmp_s^\alpha(w^R)[n+1-i]$ for $1 \leq i \leq n$. In what follows, we solely discuss the algorithm for computing $rmp_s^\alpha(w)$.

A *suffix tree with prefix periods* $\mathcal{T}_w^{\pi_s^\alpha}$ for a word w is a suffix tree $\mathcal{T}_w$ integrated with a labeling function $\pi_s^\alpha(v) = pp_s^\alpha(\tau(v))$. When s and α are clear from the

context, we simply write $\mathcal{T}_w^\pi$. The suffix tree with prefix periods satisfies the following property.

Lemma 4. *Let $\alpha > 1$ be a rational number, $s \geq 0$ be an integer, and w be a word. For any node v in the tree $\mathcal{T}_w^\pi$ such that $\pi_s^\alpha(p(v)) = +\infty$, either $\pi_s^\alpha(v)$ is $+\infty$ or $\pi_s^\alpha(v)$ satisfies $\delta(p(v))/\alpha < \pi_s^\alpha(v) \leq p(v)/(\alpha - 1)$.*

Proof. Let v be a node in $\mathcal{T}_w^\pi$ with $\pi_s^\alpha(p(v)) = +\infty$. Suppose $\pi_s^\alpha(v) \neq +\infty$. Since $\tau(p(v))$ is a prefix of $\tau(v)$ and $\pi_s^\alpha(p(v)) = +\infty$, the inequality $\delta(p(v)) < \alpha\pi_s^\alpha(v)$ follows by Lemma 1. The common prefix of $\tau(v)[1\,..\,\delta(v)]$ and $\tau(v)[\pi_s^\alpha(v) + 1\,..\,\delta(v)]$ is of length at least $(\alpha - 1)\pi_s^\alpha(v)$. Since $p(v)$ is the lowest ancestor of v in $\mathcal{T}_w^\pi$, the inequality $(\alpha - 1)\pi_s^\alpha(v) \leq \delta(p(v))$ holds. This completes the proof. □

Now we will show how to construct the $\mathcal{T}_w^{\pi_s^\alpha}$ for a word w with arbitrary $s \geq 0$ and $\alpha > 1$. Then $rmp_s^\alpha(w)$ can be obtained directly from $\mathcal{T}_w^\pi$ by $rmp_s^\alpha(w) = [\pi_s^\alpha(leaf_1), \ldots, \pi_s^\alpha(leaf_n)]$. This result generalizes Kosaraju's result [15] for the case $s = 0$, $\alpha = 2$.

The algorithm is outlined in Fig. 4. The main idea is to use Weiner's algorithm to construct the underlying suffix tree $\mathcal{T}_{w[i\,..\,n]}$ for $i = n, \ldots, 1$ (Lines 2,5) and a series of auxiliary trees (Lines 3,8,11,15,17) to help compute the π (Line 18). By Weiner's algorithm (see Fig. 2), at each step, either one or two nodes are created in the underlying suffix tree and we assign the π values on those new nodes (Lines 7,10,18). The father y of $leaf_i$ is a new node when there is a split on the edge from x to z. Since $\pi_s^\alpha(z)$ is already computed, we update $\pi_s^\alpha(y)$ directly. $leaf_i$ is the second new node. When $\pi_s^\alpha(p(leaf_i)) \neq +\infty$, we update $\pi_s^\alpha(leaf_i)$ directly. Otherwise, we compute $\pi_s^\alpha(leaf_i)$ by constructing auxiliary suffix trees. The naïve method constructs $\mathcal{T}_{w[i\,..\,n]}$ and then computes $\pi_s^\alpha(leaf_i) = pp_s^\alpha(w[i\,..\,n])$, both of which run in $O(|\,w[i\,..\,n]\,|)$-time. We instead construct a series of trees $A = \mathcal{T}_{w[i\,..\,j]}$ for some j in such a way that $pp_s^\alpha(w[i\,..\,n]) = pp_s^\alpha(w[i\,..\,j])$. Additionally, the total time of constructing the trees A is in $O(n)$; the time of computing $\pi_s^\alpha(leaf_i) = pp_s^\alpha(w[i\,..\,j])$ in each A is in $O(\alpha)$ for $s = 0$ and in $O(n)$ for arbitrary s.

Theorem 1. *Let $\alpha > 1$ be a rational number and $s \geq 0$ be an integer. Function* `compute_rmp` *in Fig. 4 correctly computes $rmp_s^\alpha(w)$ for w.*

Proof. The correctness of the algorithm relies on the claim $T_i = \mathcal{T}^\pi_{w[i\,..\,n]}$. By Weiner's algorithm, the underlying suffix tree of T_i is indeed $\mathcal{T}_{w[i\,..\,n]}$. So it remains to show the assignment of $\pi_s^\alpha(v)$ on each node v is correct.

At the beginning, $\mathcal{T}_{w[n\,..\,n]}$ contains two nodes and we have $\pi_s^\alpha(root) = pp_s^\alpha(\epsilon) = +\infty$, $\pi_s^\alpha(leaf_n) = pp_s^\alpha(w[n\,..\,n]) = +\infty$. Thus, the assignments on Line 2 are correct. Node y is the father of z (when splitting happens) and the father of $leaf_i$. Thus, by Lemma 1, the assignments on Lines 7,10 are correct. The only remaining case is the assignment of $\pi_s^\alpha(leaf_i)$ when $\pi_s^\alpha(y) = +\infty$. Since $y = p(leaf_i)$, by Lemma 4, $pp_s^\alpha(\tau(leaf_i)) > \delta(y)/\alpha$, and thus the arguments for calling `compute_pp` on Line 18 is valid. The only thing that remains is to prove that $pp_s^\alpha(w[i\,..\,n]) = pp_s^\alpha(w[i\,..\,j])$.

Input: a word $w = w[1\,..\,n]$ and two integers $s \geq 0$, $\alpha > 1$.
Output: the right minimal period array $rmp_s^{\alpha}(w)$.

```
1  begin function compute_rmp(w, s, α)
2      construct T_n by constructing T_{w[n..n]} with π(root), π(leaf_n) ⟵ +∞ ;
3      A ⟵ empty, j ⟵ n, and d ⟵ 0 ;
4      for i from n − 1 to 1 do
5          T_i ⟵extend(T_{i+1}, w[i..n]) ; // T_i = T_{w[i..n]}
6          if splitting then // y, z are obtained from extend()
7              if δ(y) ≥ απ(z) then π(y) ⟵ π(z) ; else π(y) ⟵ +∞ ;
8          if j − i + 1 > 2αd/(α − 1) or δ(y) < d/2 then A ⟵ empty ;
9          if π(y) ≠ +∞ then
10             π(leaf_i) ⟵ π(y) ;
11             if A ≠ empty then A ⟵extend(A, w[i..j]) ;
12         else
13             if A = empty then
14                 d ⟵ δ(y) and j ⟵ i + (α + 1)d/(α − 1) − 1 ;
15                 A ⟵make_suffix_tree(w[i..j]) ;
16             else
17                 A ⟵extend(A, w[i..j]) ;
18             π(leaf_i) ⟵compute_pp(A, max{s, δ(y)/α}, α) ;
19         rmp[i] ⟵ π(leaf_i) ; // T_i = T^π_{w[i..n]} is made
20     rmp[n] ⟵ +∞ and return rmp ;
21 end
```

Fig. 4. Algorithm for computing $rmp_s^{\alpha}(w)$

First, we claim that $\delta(p_{T_i}(leaf_i)) \leq \delta(p_{T_{i+1}}(leaf_{i+1})) + 1$, where the subscript of p specifies in which tree the father is discussed. If $p_{T_i}(leaf_{i+1}) p_{T_{i+1}}(leaf_{i+1})$, then there is splitting on the edge from $p_{T_{i+1}}(leaf_{i+1})$ to $leaf_{i+1}$, and thus $leaf_i, leaf_{i+1}$ have the same father in T_i. So $\tau(leaf_i)$ begins with a repetition of a single letter. Thus, we have $\delta(p_{T_i}(leaf_i)) = \delta(p_{T_i}(leaf_{i+1})) = \delta(p_{T_{i+1}}(leaf_{i+1})) + 1$. If $p_{T_i}(leaf_{i+1}) = p_{T_{i+1}}(leaf_{i+1})$, then since $\delta(leaf_i) = \delta(leaf_{i+1}) + 1$, by Lemma 2, we have $\delta(p_{T_i}(leaf_i)) \leq \delta(p_{T_i}(leaf_{i+1})) + 1 = \delta(p_{T_{i+1}}(leaf_{i+1})) + 1$.

We claim $\delta(y) \leq j - i + 1 - \frac{2}{\alpha-1}d$ holds immediately before Line 18, where $y = p(leaf_i)$. Consider the suffix tree A. If A is newly created, then $\delta(y) = d$, $i = j + 1 - \frac{\alpha+1}{\alpha-1}d$. Thus, $\delta(y) = j - i + 1 - \frac{2}{\alpha-1}d$. If A is extended from a previous suffix tree, then the index i decreases by 1, and the depth $\delta(y)$ increases at most by 1. So $\delta(y) \leq j - i + 1 - \frac{2}{\alpha-1}d$ still holds.

Now we prove $pp_s^{\alpha}(w[i\,..\,n]) = pp_s^{\alpha}(w[i\,..\,j])$. If $pp_s^{\alpha}(w[i\,..\,n]) = +\infty$, by Lemma 1, $pp_s^{\alpha}(w[i\,..\,j]) = +\infty = pp_s^{\alpha}(w[i\,..\,n])$. Assume $pp_s^{\alpha}(w[i\,..\,n]) \neq +\infty$. By Lemma 4, $pp_s^{\alpha}(w[i\,..\,n]) = pp_s^{\alpha}(\tau(leaf_i)) \leq \frac{\delta(y)}{\alpha-1}$. In addition, $j - i + 1 \leq \frac{2\alpha}{\alpha-1}d$ always holds immediately before Line 18 whenever $A \neq empty$. Therefore, $\alpha \cdot pp_s^{\alpha}(w[i\,..\,n]) \leq \frac{\alpha}{\alpha-1}\left(j - i + 1 - \frac{2}{\alpha-1}d\right) \leq |\, w[i\,..\,j]\,|$, and thus, by Lemma 1, it follows $pp_s^{\alpha}(w[i\,..\,j]) = pp_s^{\alpha}(w[i\,..\,n])$. This completes the proof of the correctness of the algorithm. □

Theorem 2. *The algorithm in Fig. 4 computes $rmp^{\alpha}_{s}(w)$ in $O(\alpha|\,w\,|)$-time for $s = 0$ and in $O(|\,w\,|^2)$-time for arbitrary s.*

Proof. Let $n = |\,w\,|$. Constructing the underlying suffix tree $\mathcal{T}_w$ is in $O(n)$-time. Every remaining statement except those on Lines 11,15,17,18 can each be done on constant-time in a unit-cost model, where we assume the operations on integers with $O(\log n)$-bits can be done in constant-time.

Now we consider the computation of Line 18. We already showed in the proof of Theorem 1 that $pp^{\alpha}_{s}(w[i\,..\,j]) = pp^{\alpha}_{s}(w[i\,..\,n])$. By Lemma 4, $pp^{\alpha}_{s}(w[i\,..\,n]) > \frac{\delta(y)}{\alpha}$. In addition, $j - i + 1 \leq \frac{2\alpha}{\alpha-1}d$ and $\delta(y) \geq \frac{1}{2}d$ always hold when $A \neq \mathit{empty}$. By Lemma 3, since $A = \mathcal{T}_{w[i\,..\,j]}$, the running time of each calling to `compute_pp` in Fig. 3 is linear in

$$\frac{|\,w[i\,..\,j]\,|}{\min\{\max\{s, \delta(y)/\alpha\}, pp^{\alpha}_{0}(w[i\,..\,j])\}} \leq \frac{2\alpha d/(\alpha-1)}{\min\{d/2\alpha, pp^{\alpha}_{0}(w[i\,..\,j])\}},$$

which is in $O(\alpha)$ for $s = 0$ and is in $O(n)$ otherwise.

Now we consider the computation of Lines 11,15,17. Those statements construct a series of suffix trees $A = \mathcal{T}_{w[i\,..\,j]}$ by calling `make_suffix_tree` and `extend` in Fig. 2. Each suffix tree is initialized at Line 15, extended at Lines 11,17, and destroyed at Line 8. Suppose there are, in total, l such suffix trees, and suppose, for $1 \leq m \leq l$, they are initialized by $A = \mathcal{T}_{w[i_m\,..\,j_m]}$ with $d_m = \delta(p_{T_{i_m}}(leaf_{i_m}))$ and destroyed when $A = \mathcal{T}_{w[i'_m\,..\,j_m]}$ such that either $j_m - (i'_m - 1) + 1 > \frac{2\alpha}{\alpha-1}d_m$ or $\delta(p_{T_{i'_m-1}}(leaf_{i'_m-1})) < \frac{1}{2}d_m$. In addition, when $A \neq \mathit{empty}$, the inequality $j_m - i + 1 \leq \frac{2\alpha}{\alpha-1}d_m$ always holds for $i_m \leq i \leq i'_m$. Since a suffix tree is constructed in linear-time in the tree size, the total running time on Lines 11,15,17 is linear in

$$\sum_{m=1}^{l} |\,w[i'_m\,..\,j_m]\,| = \sum_{m=1}^{l}(j_m - i'_m + 1) \leq \sum_{m=1}^{l}\frac{2\alpha}{\alpha-1}d_m.$$

First, we consider those cases $j_m - (i'_m - 1) + 1 > \frac{2\alpha}{\alpha-1}d_m$. Then $j_m - i'_m + 1 = \frac{2\alpha}{\alpha-1}d_m$, $j_m = i_m + \frac{\alpha+1}{\alpha-1}d_m - 1$ hold, and thus $i_m - i'_m = \left(j_m - \frac{\alpha+1}{\alpha-1}d_m + 1\right) - \left(j_m + 1 - \frac{2\alpha}{\alpha-1}d_m\right) = d_m$. Hence, $\sum_{case\ 1}\frac{2\alpha}{\alpha-1}d_m = \frac{2\alpha}{\alpha-1}\sum_{case\ 1}(i_m - i'_m) \leq \frac{2\alpha}{\alpha-1}((n-1)-1) = O(n)$. Second, we consider those cases $\delta(p_{T_{i'_m-1}}(leaf_{i'_m-1})) < \frac{1}{2}d_m$. It follows $\delta(p_{T_{i'_m-1}}(leaf_{i'_m-1})) - \delta(p_{T_{i_m}}(leaf_{i_m})) < -\frac{1}{2}d_m$. In the proof of Theorem 1, we already showed $\delta(p_{T_i}(leaf_i)) - \delta(p_{T_{i+1}}(leaf_{i+1})) \leq 1$. Hence, we have $\sum_{case\ 2}\frac{2\alpha}{\alpha-1}d_m < \frac{2\alpha}{\alpha-1}\sum_{case\ 2}2\left(\delta(p_{T_{i_m}}(leaf_{i_m})) - \delta(p_{T_{i'_m-1}}(leaf_{i'_m-1}))\right) \leq \frac{4\alpha}{\alpha-1}(n-1) = O(n)$. The only remaining case is that the suffix tree A is not destroyed even after the construction of T_1. This situation can be avoided by virtually adding a letter $\pounds$ not in the alphabet of w at the beginning of w. Thus, the total running time on Lines 11,15,17 is in $O(n)$.

Therefore, the total running time of the algorithm is in $O(\alpha n)$ for $s = 0$ and in $O(n^2)$ for arbitrary s. □

The discussion in this section is also valid for the strict prefix period. The strict version of the algorithm in Fig. 3 slightly differs from the non-strict version as described in the proof of Lemma 3, and the algorithm in Fig. 4 is the same.

4 Applications — Detecting Special Pseudo-Powers

In this section, we will show how the algorithm for computing $rmp_s^\alpha(w)$ and $lmp_s^\alpha(w)$ can be applied to test whether a word w contains any factor of a particular type of repetition: the pseudo-powers.

Let Σ be the alphabet. A function $\theta : \Sigma^* \to \Sigma^*$ is called an *involution* if $\theta(\theta(w)) = w$ for all $w \in \Sigma^*$ and called an *antimorphism* if $\theta(uv) = \theta(v)\theta(u)$ for all $u, v \in \Sigma^*$. We call θ an *antimorphic involution* if θ is both an involution and an antimorphism. For example, the classic Watson-Crick complementarity in biology is an antimorphic involution over four letters $\{\mathtt{A}, \mathtt{T}, \mathtt{C}, \mathtt{G}\}$ such that $\mathtt{A} \mapsto \mathtt{T}$, $\mathtt{T} \mapsto \mathtt{A}$, $\mathtt{C} \mapsto \mathtt{G}$, $\mathtt{G} \mapsto \mathtt{C}$. For integer k and antimorphism θ, we call word w a *pseudo kth power* (with respect to θ) if w can be written as $w = x_1 x_2 \cdots x_k$ such that either $x_i = x_j$ or $x_i = \theta(x_j)$ for $1 \le i, j \le k$. In particular, we call a pseudo 2nd (3rd) power a *pseudo square* (*cube*). For example, over the four letters $\{\mathtt{A}, \mathtt{T}, \mathtt{C}, \mathtt{G}\}$, the word $\mathtt{ACGCGT} = \mathtt{ACG}\theta(\mathtt{ACG})$ is a pseudo square and $\mathtt{ACGTAC} = \mathtt{AC}\theta(\mathtt{AC})\mathtt{AC}$ is a pseudo cube with respect to the Watson-Crick complementarity. Pseudo kth powers are of particular interest in bio-computing [3]. A variation on the pseudo kth power has also appeared in tiling problems [2].

Chiniforooshan, Kari, and Xu [3] discussed the problem of testing whether a word w contains any pseudo kth power as a factor. There is a linear-time algorithm and a quadratic-time algorithm for testing pseudo squares and pseudo cubes, respectively. For testing arbitrary pseudo kth powers, the known algorithm is in $O(|\,w\,|^2 \log |\,w\,|)$-time.

We will show these particular types of pseudo kth powers, $\theta(x)x^{k-1}$, $x^{k-1}\theta(x)$, and $(x\theta(x))^{\frac{k}{2}}$ (where $(x\theta(x))^{\frac{k}{2}} = (x\theta(x))^{\lfloor \frac{k}{2} \rfloor}x$ for odd k) can be tested faster. First, we need the following concept. The *centralized maximal pseudo-palindrome array* cmp_w^θ of word w with respect to an antimorphic involution θ is defined by $cmp_w^\theta[i] = \max\{m : 0 \le m \le \min\{i, |\,w\,| - i\}, \theta(w[i-m+1\,..\,i]) = w[i+1\,..\,i+m]\}$ for $0 \le i \le |\,w\,|$. For example, $cmp_{0100101001}^\theta = [\ 0, 0, 0, 3, 0, 0, 0, 0, 2, 0, 0\ ]$.

Lemma 5. *Let θ be an antimorphic involution. The array cmp_w^θ can be computed in $O(|\,w\,|)$-time.*

Proof. Constructing suffix tree $\mathcal{T}_{w\pounds\theta(w)}$, where letter $\pounds$ is not in w, cmp_w^θ can be computed via $\mathcal{T}_{w\pounds\theta(w)}$ by $cmp_w^\theta[i] = \delta(\mathrm{lca}(leaf_{i+1}, leaf_{2n-i+2}))$ for $1 \le i \le n-1$ and $cmp_w^\theta[0] = cmp_w^\theta[n] = 0$. □

Theorem 3. *Let $k \ge 2$ and $s \ge 0$ be integers, and θ be an antimorphic involution. Whether a word w contains any factor of the form $x^{k-1}\theta(x)$ (resp., $\theta(x)x^{k-1}$) with $|\,x\,| > s$ can be tested in $O(|\,w\,|^2)$-time and in $O(k|\,w\,|)$-time for $s = 0$.*

Proof. Computing $lmp_s^{k-1}(w)$ (resp., $rmp_s^{k-1}(w)$) and cmp_w^θ, there is a factor $x^{k-1}\theta(x)$ (resp., $\theta(x)x^{k-1}$) with $|\,x\,| > s$ if and only if $lmp_s^{k-1}(w)[i] \leq cmp_w^\theta[i]$ (resp., $rmp_s^{k-1}(w)[i] \leq cmp_w^\theta[i-1]$) for some $1 \leq i \leq n$. □

Theorem 4. *Let $k \geq 2$ and $s \geq 0$ be integers and θ be an antimorphic involution. Whether a word w contains any factor of the form $(x\theta(x))^{\frac{k}{2}}$ with $|\,x\,| > s$ can be tested in $O(|\,w\,|^2/k)$-time.*

Proof. Computing cmp_w^θ and enumerating all possible indices and periods, there is a factor $(x\theta(x))^{\frac{k}{2}}$ with $|\,x\,| > s$ if, and only if, there are $k-1$ consecutive terms greater than s in cmp_w^θ with indices being an arithmetic progression with difference greater than s. □

5 Conclusion

We generalized Kosaraju's $O(|\,w\,|)$-time algorithm of computing minimal squares starting at each position in a word w, which by our definition is presented by $rmp_0^2(w)$. We showed a modified algorithm that can compute, for an arbitrary rational number $\alpha > 1$ and integer $s \geq 0$, the minimal α powers to the right and to the left, with (either non-strict or strict) period larger than s, starting at each position in a word, which are presented as the right minimal period array $rmp_s^\alpha(w)$ and the left minimal period array $lmp_s^\alpha(w)$, respectively.

The algorithm is based on the frame of Weiner's suffix-tree construction. Although there are other linear-time algorithms for suffix-tree construction, such as McCreight's algorithm and Ukkonen's algorithm, none of the two can be altered to compute minimal period arrays with the same efficiency, due to the special requirement that the suffixes of the given word are added into the tree in the order of shortest to longest and $\pi_s^\alpha(v)$ is only updated when node v is created.

The naïve approach to compute $rmp_s^\alpha(w)$ is to compare factors, for each position and for each possible choice of period, to test whether that period satisfies the definition of the prefix period. This procedure leads to an algorithm using $O(1)$ extra space and running in $O(|\,w\,|^3/\alpha)$-time. By building a failure table as used in the Knuth-Morris-Pratt pattern matching algorithm [13], there is an algorithm using $O(n)$-space and running in $O(n^2)$-time for the case of non-strict prefix period with $s = 0$. The algorithm in the paper uses $O(n)$-space, runs in $O(\alpha|\,w\,|)$-time for $s = 0$, and runs in $O(|\,w\,|^2)$-time for arbitrary s. Here we assume the alphabet is fixed. An online interactive demonstration of all three algorithms can be found at the author's web-page [26].

We showed the algorithm for computing minimal period arrays can be used to test whether a word w contains any factor of the form $x^k\theta(x)$ (resp., $\theta(x)x^k$) with $|\,x\,| > s$, which runs in $O(k|\,w\,|)$-time for $s = 0$ and runs in $O(|\,w\,|^2)$-time for arbitrary s. We also discussed an $O(|\,w\,|^2/k)$-time algorithm for testing whether a word w contains any factor of the form $(x\theta(x))^{\frac{k}{2}}$ with $|\,x\,| > s$. All of the words $xx\cdots x\theta(x)$, $\theta(x)x\cdots xx$, $x\theta(x)x\theta(x)\cdots$ are pseudo-powers. There are possibilities that some particular types of pseudo-powers other than those

discussed here may be detected faster than the known $O(|\,w\,|^2 \log |\,w\,|)$-time algorithm.

Acknowledgements

The author would like to thank Prof. Lila Kari for discussion on pseudo-powers, Prof. Lucian Ilie for discussion on the computing of cmp_w^θ, and the anonymous referees for their valuable comments.

References

1. Apostolico, A., Preparata, F.P.: Optimal off-line detection of repetitions in a string. Theoret. Comput. Sci. 22, 297–315 (1983)
2. Beauquier, D., Nivat, M.: On translating one polyomino to tile the plane. Discrete Comput. Geom. 6(1), 575–592 (1991)
3. Chiniforooshan, E., Kari, L., Xu, Z.: Pseudo-power avoidance. CoRR abs/0911.2233 (2009), `http://arxiv.org/abs/0911.2233`
4. Crochemore, M.: Optimal algorithm for computing the repetitions in a word. Info. Proc. Lett. 12(5), 244–250 (1981)
5. Crochemore, M.: Recherche linéaire d'un carré dans un mot. Comptes Rendus Acad. Sci. Paris Sér. I 296, 781–784 (1983)
6. Crochemore, M., Ilie, L., Rytter, W.: Repetitions in strings: Algorithms and combinatorics. Theoret. Comput. Sci. 410(50), 5227–5235 (2009)
7. Duval, J., Kolpakov, R., Kucherov, G., Lecroq, T., Lefebvre, A.: Linear-time computation of local periods. Theoret. Comput. Sci. 326, 229–240 (2004)
8. Fraenkel, A.S., Simpson, J.: How many squares can a string contain? J. Combin. Theory Ser. A 82(1), 112–120 (1998)
9. Gusfield, D.: Algorithms on strings, trees, and sequences: computer science and computational biology. Cambridge University Press, Cambridge (1997)
10. Gusfield, D., Stoye, J.: Linear time algorithms for finding and representing all the tandem repeats in a string. J. Comput. Syst. Sci. 69(4), 525–546 (2004)
11. Harel, D., Tarjan, R.E.: Fast algorithms for finding nearest common ancestors. SIAM J. Comput. 13(2), 338–355 (1984)
12. Ilie, L.: A note on the number of squares in a word. Theoret. Comput. Sci. 380(3), 373–376 (2007)
13. Knuth, D., Morris, J., Pratt, V.: Fast pattern matching in strings. SIAM J. Comput. 6(2), 323–350 (1977)
14. Kolpakov, R., Kucherov, G.: Finding maximal repetitions in a word in linear time. In: Proc. 40th Ann. Symp. Found. Comput. Sci (FOCS 1999), pp. 596–604. IEEE Computer Society Press, Los Alamitos (1999)
15. Kosaraju, S.R.: Computation of squares in a string. In: Crochemore, M., Gusfield, D. (eds.) Proc. 5th Combinat. Patt. Matching, pp. 146–150. Springer, Heidelberg (1994)
16. Main, M., Lorentz, R.: An $O(n \log n)$ algorithm for finding all repetitions in a string. J. Algorithms 5(3), 422–432 (1984)
17. Main, M., Lorentz, R.: Linear time recognition of square free strings. In: Apostolico, A., Galil, Z. (eds.) Combinat. Algor. on Words, pp. 272–278. Springer, Heidelberg (1985)

18. Main, M.G.: Detecting leftmost maximal periodicities. Discrete Appl. Math. 25(1–2), 145–153 (1989)
19. McCreight, E.M.: A space-economical suffix tree construction algorithm. J. Assoc. Comput. Mach. 23(2), 262–272 (1976)
20. Schieber, B., Vishkin, U.: On finding lowest common ancestors: Simplification and parallelization. SIAM J. Comput. 17(6), 1253–1262 (1988)
21. Slisenko, A.O.: Detection of periodicities and string-matching in real time. J. Math. Sci (N. Y.) 22(3), 1316–1387 (1983)
22. Stoye, J., Gusfield, D.: Simple and flexible detection of contiguous repeats using a suffix tree preliminary version. In: Farach-Colton, M. (ed.) Proc. 9th Combinat. Patt. Matching, pp. 140–152. Springer, Heidelberg (1998)
23. Thue, A.: Über unendliche Zeichenreihen. Norske Vid. Selsk. Skr. I. Mat.-Nat. Kl (7), 1–22 (1906)
24. Ukkonen, E.: Constructing suffix trees on-line in linear time. In: Leeuwen, J.V. (ed.) Proc. Infor. Proces. 92, IFIP Trans. A-12., Vol. 1. pp. 484–492. Elsevier, Amsterdam (1992)
25. Weiner, P.: Linear pattern matching algorithms. In: Proc. 14th IEEE Ann. Symp. on Switching and Automata Theory (SWAT), pp. 1–11 (1973)
26. Xu, Z.: `http://www.csd.uwo.ca/~zhi_xu/demons/cpm2010xu.html` (2010)

The Property Suffix Tree with Dynamic Properties

Tsvi Kopelowitz

Department of Computer Science, Bar-Ilan University, Ramat-Gan 52900, Israel

Abstract. Recently there has been much interest in the Property Indexing Problem ([1],[7],[8]), where one is interested to preprocess a text T of size n over alphabet Σ (which we assume is of constant size), and a set of intervals π over the text positions, such that give a query pattern P of size m we can report all of the occurrences of P in T which are completely contained within some interval from π. This type of matching is extremely helpful in scenarios in molecular biology where it has long been a practice to consider special areas in the genome by their structure.

The work done so far has focused on the static version of this problem where the intervals are given a-priori and never changed. This paper is the first to focus on several dynamic settings of π including an incremental version where new intervals are inserted into π, decremental version where intervals are deleted from π, fully dynamic version where intervals may be inserted or deleted to or from π, or batched insertions where a set of intervals is inserted into π. In particular, the batched version provides us with a new (optimal) algorithm for the static case.

1 Introduction

In many pattern matching applications the text has some properties attached to various locations in it. A *property* for a string is the set of intervals corresponding to the parts of the string satisfying the conceptual property we are looking for. Property Matching, involves a string matching between the pattern and the text, and the requirement that the text part is contained within one of the intervals.

Some examples come from molecular biology, where it has long been a practice to consider special areas of the genome by their structure. Examples are repetitive genomic structures [10] such as *tandem repeats*, *LINEs* (Long Interspersed Nuclear Sequences) and *SINEs* (Short Interspersed Nuclear Sequences) [9]. Many problems in biology can be expressed as property matching problems, for example, finding all occurrences of a given pattern in a genome, provided it appears in a SINE, or LINE.

Clearly, there is no great challenge in sequential pattern matching with properties since the intersection of the properties and matching can be done in linear time. However, the problem becomes more complex when it is required to *index* a text with properties. The classical pattern matching problem [2],[13] is that of finding all occurrences of *pattern* $P = p_1p_2 \cdots p_m$ in *text* $T = t_1t_2 \cdots t_n$, where

A. Amir and L. Parida (Eds.): CPM 2010, LNCS 6129, pp. 63–75, 2010.

T and P are strings over alphabet Σ. In the *indexing problem* we are given a large text that we want to preprocess in a manner that allows fast solution of the following queries: "Given a (relatively short) pattern P find all occurrences of P in T in time proportional to $|P|$ and the number of occurrences".

The indexing problem and its many variants have been central in pattern matching (e.g. [18],[6],[5],[15],[3]). However, when it comes to indexing a text with properties, we are now presented with a dilemma. If we use the conventional indexing techniques and then do the intersection with the properties, our worst case time may be very large in case the pattern appears many times, and there may not be any final matches in case all the indexed matches do not satisfy the property.

Thus, a few recent results papers have tackled this problem by building the *Property Suffix Tree* (or PST for short), which is essentially a suffix tree, where each suffix is truncated so we only consider the smallest prefix of it which satisfies our property. In [1], where the PST was defined, it was shown how one can construct the PST in $O(|\pi| + n \log\log n)$ time using weighted ancestor queries ([4],[12]). Later on, in [7] the authors there showed how one can construct the PST in $O(|\pi| + n)$ time using range minimum queries. However, there was an unfortunate error in their result, which was corrected by [8].

In this paper, we consider the dynamic version of the problem, where intervals are inserted or removed to or from the property. We present algorithms for a few versions of this problem, namely incremental updates, decremental updates, fully dynamic updates, and batched updates. While one could use the previous results, and use the dynamic versions of the inner data structures in there (like dynamic range minimum queries), the overhead of such a change is too expensive. Thus, we present a new approach for confronting the PST, resulting in a new optimal construction algorithm for the static case as well. The approach is based on clever use of suffix links.

The paper is organized as follows. In section 2 we give some definitions and preliminaries. In section 3 we confront the incremental version of the dynamic problem, which includes the introduction of the basic procedure implementing our new approach for this problem, using suffix links. In section 4 we show how to solve the decremental version. In sections 5 and 6, we show how to solve the fully dynamic and batched versions, respectively, and the batched version leads us to a new optimal static construction.

2 Preliminaries and Definitions

For a string $T = t_1 \cdots t_n$, we denote by $T_{i\cdots j}$ the substring $t_i \cdots t_j$. The suffix $T_{i\cdots n}$ is denoted by T^i, and the suffix tree of T is denoted by $ST(T)$. The leaf corresponding to T^i in $ST(T)$ is denoted by $leaf(T^i)$. The label of an edge e in $ST(T)$ is denoted by $label(e)$. The concatenation of all of the labels of edges from the root of the suffix tree to a node u is denoted by $label(u)$. For a node u in the suffix tree of a string T, we denote by ST_u the subtree of the suffix tree rooted by u.

We are now ready to define a property for a string.

Definition 1. *A property π of a string $T = t_1...t_n$ is a set of intervals $\pi = \{(s_1, f_1), ..., (s_t, f_t)\}$ where for each $1 \leq i \leq t$ it holds that: (1) $s_i, f_i \in \{1, ..., n\}$, and (2) $s_i \leq f_i$. The size of property π, denoted by $|\pi|$, is the number of intervals in the property (or in other words - t).*

The following definition is a slight simplification of a similar definition in [1].

Definition 2. *Given a text $T = t_1...t_n$ with property π and pattern $P = p_1...p_m$, we say that P matches $T_{i...j}$ under property π if $P = T_{i...j}$, and there exists $(s_k, f_k) \in \pi$ such that $s_k \leq i$ and $j \leq f_k$.*

If $P = T_{i...j}$ we want to be able to quickly check if there exists an interval $(s, f) \in \pi$ for which $s \leq i \leq j \leq f$. To simplify such a test we introduce the notion of *extents* and the *maximal extent*.

Definition 3. *Given a text $T = t_1...t_n$ with property π for every text location $1 \leq i \leq n$ and interval $(s, f) \in \pi$ such that $s \leq i \leq f$ we say that f is an extent of i. The maximal extent of i is the extent of i of largest value, or in other words, the finish of the interval containing i which is the most distant from i. We denote the maximal extent of i by $end(i)$. If for some location i there is no interval in π containing it, we define $end(i) = NIL$.*

The following lemma shows us the connection between the maximal extents defined by a property π, and being able to quickly test if a pattern matches a location in the text under π.

Lemma 1. *Given a text $T = t_1...t_n$ with property π and pattern $P = p_1...p_m$, P matches $T_{i...j}$ under property π if and only if $P = T_{i...j}$ and $j \leq end(i)$.*

Proof. We need to show that $j \leq end(i)$ if and only if there exists $(s_k, f_k) \in \pi$ such that $s_k \leq i$ and $j \leq f_k$. The first direction is true, as by definition of maximal extents, if $j \leq end(i)$ we have that there exists an interval $(s, f) \in \pi$ such that $s \leq i \leq j \leq f = end(i)$. The reverse direction is true as well, as if there exists an interval $(s_k, f_k) \in \pi$ such that $s_k \leq i$ and $j \leq f_k$, we have that $i \in [s_k, j] \subseteq [s_k, f_k]$ and so f_k is an extent of i implying that $j \leq f_k \leq end(i)$. □

Being that the size of π can be $O(n^2)$, it would be helpful if we could reduce π to another property π' where we are guaranteed that the size of π' is at most n. In fact, it will be even more helpful to reduce our property to one of minimal size. This can be accomplished with the following.

Definition 4. *Two properties π and π' for text $T = t_1...t_n$ are said to be congruent if for any pattern P, P matches a substring $T_{i,j}$ under π if and only if it matches the same substring under π'.*

The congruent relation between properties defines equivalence classes for all possible properties of a text of size n. Every two properties in a given equivalence class will produce the same output when querying a text T with pattern P under those properties. This naturally leads us to the following definition:

Definition 5. *A property π for a string of length n is said to be minimal if for any property π' congruent to π we have that $|\pi| \leq |\pi'|$. Also, the process of converting π' into π is called minimizing π'.*

Lemma 2. *A property π for a string of length n can be minimized using the following process. For any two intervals $(s, f), (s', f') \in \pi$ we remove (s', f') if one of the following conditions holds: (a)$s < s' \leq f' < f$, (b)$s < s' \leq f' = f$, or (c) $s' = s \leq f \leq f'$. Furthermore, the minimized form of π is of size $O(n)$.*

Proof. Due to space limitations, this easy proof is omitted. □

Definition 6. *A property π for a string of length n is said to be in standard form if (a) π is minimal, and (b) $s_1 < s_2 < \cdots < s_{|\pi|}$.*

For some of the dynamic settings (incremental, and batch inserts) we will want to maintain π in its minimized form. For other settings, we will need to maintain π completely (meaning not minimized), as deletions of intervals can strongly affect the resulting properties' equivalence classes due to deletions from two different properties in the same equivalence class.

2.1 The Property Suffix Tree

The PST in essence shortens the root to leaf path of each suffix (leaf) in the suffix tree, to its maximal extent. A simple (naive) method for constructing the PST would be for every leaf in the suffix tree, traverse the path from the root to it, and stop at the character corresponding to the maximal extent of the suffix. If this location is an edge, we break that edge into two, and insert a new node for the suffix. If this location is a node, we just add the suffix to a list of suffixes in each node. When removing the leaf corresponding to the suffix in the original suffix tree, and the edge connecting it to its parent, we might need to remove the parent from the tree as well. In fact, we won't want to remove nodes from the tree, but rather create a *shortcut* edge skipping the internal node that needs to be removed. So, we can envision the PST to be the complete suffix tree with the addition of some shortened suffix nodes, and shortcut edges, so that when we traverse the PST (through the suffix tree) with a pattern query, we take a shortcut edge whenever possible.

When answering a query, we traverse down the PST with our pattern, and once we find the edge or node corresponding to the pattern, we traverse that node's or edge's subtree to output all of the (shortened) suffixes in it. It should be noted that although not all of the nodes in the subtree have at least two children (as some inner nodes correspond to shortened suffixes), we can still perform this traversal in time linear in the size of the output as the non branching nodes are those which correspond to part of the output.

The construction takes quadratic time, and is therefore inefficient. However, we show in section 6.1 how the PST can be constructed in linear time.

3 The Incremental Version

In this section it is shown how to solve the following problem:

Problem 1. Given a PST for text $T = t_1...t_n$ and a property π of T we wish to maintain the PST under the following updates efficiently:

- Insert(s, f) - Insert a new interval (s, f) into π.

We assume that π is maintained in standard form. So π consists of at most n intervals, each with a different starting index.

There are several types of updates that can happen to the PST due to an Insert(s, f) update. These different types of updates are provided by the different interactions between the newly inserted interval (s, f) and the intervals already in π prior to the insertion. For simplicity we assume that there is only one interval $(s', f') \in \pi$, such that $[s, f] \cap [s', f'] \neq \emptyset$. This interval can be easily located in $O(f - s)$ time by maintaining each interval in its starting and end location (due to the minimized form, there is at most one starting and one ending interval at each location). Then we can scan the length of the interval, locating (s', f') (if it exists). In a situation where there are no starting or finishing locations in the interval, we check $end(s)$ so that if it is not NIL, it must be that $end(s) > f$ and so the new interval is completely contained within an already existing interval in π (a situation which we briefly discuss next).

The possible interactions are as follows

1. The first type of interaction is when $s' \leq s \leq f \leq f'$. In such a case the new interval does not affect the maximum extent of any of the text locations due to this interaction and so the PST and π remain unchanged as we want to maintain π in standard form.
2. The second type of interaction is when $s \leq s' \leq f' \leq f$. In such a case the new interval completely contains (s', f') so for any text location i where $s \leq i \leq f$, the new interval will change $end(i)$ to be f, and the PST must be updated to support this. Furthermore, the insertion of the new interval will force (s', f') out of π to be replaced by (s, f).
3. The third type of interaction is when $s \leq s' \leq f \leq f'$. In such a case we only need to update the maximal extent for text location i where $s \leq i < s'$ as for any location $s' \leq i \leq f'$ we have that f' provides a longer property extent. The PST must be updated to support this, and the new interval is added to π.
4. The fourth type of interaction is when $s' \leq s \leq f' \leq f$. In such a case we only need to update the maximal extent for text location i where $s \leq i \leq f'$ as for any text location $s \leq i \leq f'$ we have that f provides a longer property extent (as opposed to the one provided by (s', f')), and for the remaining $f' < i \leq f$ we have that $end(i) = f$ as it is the only extent available. The PST must be updated to support this, and the new interval is added to π.

For the last three (out of four) types of interactions, we need to update the maximum extent for some text locations, and update the PST accordingly.

The locations for which we will have changes made in the PST can be found in $O(f-s)$ time by scanning the interval in the text, and marking the appropriate locations (according to the interaction). If we are given the old location of a shortened suffix in the PST prior to the insertion, together with its new location in the PST after the insertion, the additional work will take constant time. Thus, we are left with the job of locating the new position of each location in the interval which imposes a change in the PST, as the old locations can be easily maintained per location. This is explained next.

3.1 The PST Update Traversal

We begin by noting that in any interaction that imposes changes to the suffix tree, the location s (the start of the interval) will always cause a change. We traverse the PST with the substring $T_{s,f}$, till we reach a node u for which $label(u)$ is a prefix of $T_{s,f}$ and is of maximum length. We can denote $label(u) = T_{s,x}$ for some $s \leq x \leq f$. If $x = f$ then the shortened suffix of s needs to be inserted at u in the PST (as f is the maximal extent of s). Otherwise, let w be the child of u in the PST for which the first character on the edge (u, w) is t_{x+1}. The path in the PST corresponding to $T_{s,f}$ ends on this edge, and so, the shortened suffix of s needs to be inserted at a new node v breaking the edge (u, w) into two. We note briefly that the time to traverse any edge in the traversal can be done in constant time, as we know that $T_{s,f}$ is in the text. However, when inserting the new edge v, we want to insert it into the original suffix tree as well (as edge (u, w) might be a shortcut). To do this, we traverse down the *suffix tree* from u till we reach a node $(\hat{u})$ for which $label(\hat{u})$ is a prefix of $T_{s,f}$ and is of maximum length. We can denote $label(\hat{u}) = T_{s,\hat{x}}$ for some $s \leq \hat{x} \leq f$. If $\hat{x} = f$ then the shortened suffix of s needs to be inserted at $\hat{u}$ in the suffix tree. Otherwise, let $\hat{w}$ be the child of $\hat{u}$ in the suffix tree for which the first character on the edge $(\hat{u}, \hat{w})$ is $t_{\hat{x}+1}$. We then insert node v into edge $(\hat{u}, \hat{w})$, updating shortcuts as needed.

We now wish to find the new location in the PST for $s+1$. We could rescan the PST with $T_{s+1,f}$, however that would take too long. Instead, we use the suffix link of u to find a node u' for which $label(u) = t_s \dot{l}abel(u')$. Thus, $label(u') = T_{s+1,x}$. From u' we continue to traverse down the suffix tree with the substring $T_{x+1,f}$, till we reach a node u_1 for which $label(u_1)$ is a prefix of $T_{s+1,f}$ and is of maximum length. We can denote $label(u_1) = T_{s,x_1}$ for some $s+1 \leq x_1 \leq f$. If $x_1 = f$ then the shortened suffix of $s+1$ needs to be inserted at the node corresponding to u_1 in the PST. Otherwise, let w_1 be the child of u_1 for which the first character on the edge (u_1, w_1) is t_{x_1+1}. The path in the PST corresponding to $T_{s+1,f}$ ends on this edge, and so, the shortened suffix of $s+1$ needs to be inserted at a new node v_1 breaking the edge (u_1, w_1) into two. As before, when inserting the new node v_1, we want to insert it into the original suffix tree as well. To do this, we take the suffix link from $\hat{u}$, and continue to traverse down the *suffix tree* from the node at the other side of the suffix link $\hat{u}_1$ for which $label(\hat{u}_1)$ is a prefix of $T_{s+1,f}$ and is of maximum length. We can denote $label(\hat{u}_1) = T_{s+1,\hat{x}_1}$ for some $s+1 \leq \hat{x}_1 \leq f$. The rest of the work for

this case is the same as in the first phase, however we might also need to update suffix links for the newly inserted node.

We continue this process, where at the $i'th$ iteration we use the suffix link of u_{i-1} to find a node u'_{i-1} for which $label(u_{i-1}) = t_s\dot{l}abel(u'_{i-1})$. Thus, $label(u'_{i-1}) = T_{s+i,x}$. From u'_{i-1} we continue to traverse down the suffix tree with the substring $T_{x_i,f}$, till we reach a node u_i for which $label(u_i)$ is a prefix of $T_{s+i,f}$ and is of maximum length. We can denote $label(u_i) = T_{s,x_i}$ for some $s + i \leq x_1 \leq f$. If $x_i = f$ then the shortened suffix of $s + i$ needs to be inserted at the node corresponding to u_i in the PST. Otherwise, let w_i be the child of u_i for which the first character on the edge (u_i, w_i) is t_{x_i+1}. The path in the PST corresponding to $T_{s+i,f}$ ends on this edge, and so, the shortened suffix of $s + i$ needs to be inserted at a new node breaking the edge (u_i, w_i) into two. As before, when inserting the new node v_i, we want to insert it into the original suffix tree as well. To do this, we take the suffix link from $\hat{u}_i$, and continue to traverse down the *suffix tree* from the node at the other side of the suffix link $\hat{u}_i$ for which $label(\hat{u}_i)$ is a prefix of $T_{s+i,f}$ and is of maximum length. We can denote $label(\hat{u}_i) = T_{s+i,\hat{x}_1}$ for some $s+i \leq \hat{x}_i \leq f$. The rest of the work for this case is the same as in the first phase, however we might also need to update suffix links for the newly inserted node.

The process ends after $f - s$ iterations. In each iteration we updated the location of at most one shortened suffix. The running time is as follows. All of the updates done to a suffix once its new location is found take constant time. The total traversal in order to find all of the new locations in the PST take a total of $(f - s)$ time, as in each iteration i we will traverse from x_{i-1} to x_i, and from $\hat{x}_{i-1}$ to $\hat{x}_i$.

3.2 Multiple Interactions

If our new interval (s, f) interacts with more than one interval already in π we need to be able to determine which different interactions occur, and decide accordingly which locations require a change. In order to do this, we note that the two types of interactions that might cause the maximum extent for a given location *not* to change are the first and third. It is enough to detect one interval $(s', f') \in \pi$ whose interaction with (s, f) is of the first type. This can be done by checking the end location of s. So assume this is not the case, and focus on dealing with many interactions of the third type. we can locate all such interactions in $O(f - s)$ time by scanning the interval, and checking end locations for each location encountered. for a location i if $end(i) > f$ then we know that for any location j such that $\leq j \leq f$, $end(j)$ will not change. So, once we reach the first $s \leq i \leq f$ we know that the only locations that need to change are in the range $[s, i-1]$. Once these locations have been determined, we perform the PST update traversal in order to complete the process.

Theorem 7. *Given a PST for text $T = t_1...t_n$ and a property π of T it is possible to maintain the PST under Insert(s, f) operations in $O(f - s)$ time.*

4 The Decremental Version

In this section it is shown how to solve the following problem:

Problem 2. Given a PST for text $T = t_1...t_n$ and a property π of T we wish to maintain the PST under the following updates efficiently:

- Delete(s, f) - Delete the interval (s, f) from π.

As intervals are being deleted from π, maintaining π in standard form is dangerous, as once an interval is deleted, we might require another interval which was not in the standard form. Therefore we can no longer assume that the size of π is linear in the size of the text. We begin with a preprocessing phase in which for every text location i we build two lists. The first list, denoted by φ_i, is the list of all intervals in π for which the starting time is i, sorted by decreasing finishing time. The second list, denoted by γ_i is the list of all intervals in π for which the finishing time is i, sorted by increasing starting time. We can easily construct both lists in $O(n + |\pi|)$ time. In addition, we use a hash function such that given an interval (s, f) we can locate in constant time if $(s, f) \in \pi$, and if so it will return two pointers to nodes, one to the node in the list φ_s and one to the node in the list γ_f which refer to that interval in those lists.

Like in the incremental version, there are several types of updates that can happen to the PST due to a Delete(s, f) update. These different types of updates are provided by the different interactions between the deleted interval (s, f) and the other intervals currently in π. For simplicity we assume that there is only one interval $(s', f') \in \pi \setminus \{(s, f)\}$, such that $[s, f] \cap [s', f'] \neq \phi$. This interval can be easily found in $O(f - s)$ time as mentioned before in the incremental version, however we need to use the first nodes in the lists φ_i and γ_i for each $s \leq i \leq f$.

1. The first type of interaction is when $s' \leq s \leq f \leq f'$. In such a case the deleted interval does not affect the maximum extent of any of the text locations.
2. The second type of interaction is when $s \leq s' \leq f' \leq f$. In such a case the deleted interval completely contains (s', f') and so for any text location i where $s' \leq i \leq f'$, the deletion will change $end(i)$ to be f', while for the other locations in $[s, f]$ we need to set their end location to NIL.
3. The third type of interaction is when $s \leq s' \leq f \leq f'$. In such a case we only need to update the end location for text location i where $s \leq i < s'$ to be NIL.
4. The fourth type of interaction is when $s' \leq s \leq f' \leq f$. In such a case we only need to update the end location for text location i where $s \leq i \leq f$ as for any location $s \leq i \leq f'$ we have that f' provided the longer property extent (as opposed to the one provided by (s, f)), and for the remaining $f' < i \leq f$ we have that $end(i) = NIL$ as there is no extent available.

For the last three (out of four) types of interactions, we need to update the maximum extent for some text locations, and update the PST accordingly.

The locations for which we will have changes made in the PST can be found in $O(f-s)$ time by scanning the interval in the text, and marking the appropriate locations (according to the interaction). If we are given the old location of a shortened suffix in the PST prior to the insertion, together with its new location in the PST after the insertion or an indication that no such extent exists, the additional work will take constant time. Thus, we are left with the job of locating the new position of each location in the interval which imposes a change in the PST, as the old locations can be easily maintained per location.

In order to do this, we want to use the PST update traversal. However in order for the traversal to be correct, we must prove the following lemma.

Lemma 3. *Any change to a maximal extent made due to a Delete(s, f) update will result in a NIL, or an extend k such that $s \leq k \leq f$.*

Proof. Assume by contradiction that there exists a text location i such that due to a Delete(s, f) update, $end(i)$ changes to be k such that $k < s$ or $k > f$. if $k < s$ then we also must have $i < s$, and as such the interval which provides i with its maximal extent does not interact with (s, f), contradicting our assumption that $end(i)$ was changed. If $k > f$, then there exists an interval $(s', k) \in \pi$ such that $s' \leq i \leq f < k$. However, this interval existed prior to the deletion of (s, f) and so the maximal extent of i should not have changed due to the deletion. □

Now aided by the lemma, we note that the PST traversal will traverse through all of the new locations that the shortened suffixes need to be updated at due to the deletions. Also, we must remove (s, f) from φ_s and $gamma_f$, however as these values are hashed, this takes constant time. The running time is the same as that of the insert operation - $O(f-s)$.

4.1 Multiple Interactions

If our deleted interval (s, f) interacts with more than one interval in $\pi \setminus \{(s, f)\}$ we need to be able to determine which different interactions occur, and decide accordingly which locations require a change. This is done in a similar method to that of the insertions, and is thus omitted.

Theorem 8. *Given a PST for text $T = t_1...t_n$ and a property π of T it is possible to maintain the PST under Delete(s, f) operations in $O(f-s)$ time.*

5 The Fully Dynamic Version

Problem 3. Given a PST for text $T = t_1...t_n$ and a property π of T we wish to maintain the PST under the following updates efficiently:

- Insert(s, f) - Insert a new interval (s, f) into π.
- Delete(s, f) - Delete the interval (s, f) from π.

As intervals are being deleted from π, maintaining π in standard form is still dangerous, as once an interval is deleted, we might require another interval which was not in the standard form. We begin with a preprocessing phase in which for every text location i we build φ_i and γ_i, both maintained as dynamic priority queues. In addition, we use a hash function such that given an interval (s, f) we can locate in constant time if $(s, f) \in \pi$, and if so it will return two pointers to nodes, one to the node in φ_s and one to the node in γ_f which refer to that interval in those lists.

The procedures for processing an insertion or deletion are the same as that of the incremental and decremental ones, with the following changes. All of the interactions that are made with φ_i and γ_i for some location i are done through the appropriate priority queue. Thus each update performs two lookups (one for a minimum value, and one for a maximum value), each insertion performs two priority queue insertions, and each deletion performs two priority queue deletions. We can use the data structure by van Emde Boas [17] so that each priority queue operation requires $O(\log \log n)$ time. Thus we have the following.

Theorem 9. *Given a PST for text $T = t_1...t_n$ and a property π of T it is possible to maintain the PST under Delete(s, f) and Insert(s, f) operations in $O(f - s + \log \log n)$ time per operation.*

6 The Batched Insert Version and the Static Case

In this section we solve the following problem.

Problem 4. Given a PST for text $T = t_1...t_n$ and a property π of T we wish to maintain the PST under the following updates efficiently:

- Insert(I) - Insert the set of intervals $I = \{(s_1, f_1), (s_2, f_2), ..., (s_\ell, f_\ell)\}$ into π.

We could perform an insert operation per interval in I, and that would take $O(\sum_{(s,f)\in I} f - s)$ time. However, we do better by presenting an algorithm that runs in $O(\textit{cover-size}(I))$ where

$$\textit{cover-size}(I) = |\{1 \leq i \leq n : \exists (s, f) \in I s.t. i \in [s, f]\}|.$$

To do this, we begin by processing the intervals in I as follows. We start with an array A of size n. We do not need to spend the time to initialize A - instead we can use standard techniques which know for a given location in A if it has been set or not. Next we create an undirected graph $G = (V, E)$, where for each interval $(s_k, f_k) \in I$ we have a corresponding vertex $v_k \in V$, and we initiate E to be empty. Now for each $1 \leq i \leq \ell$ we scan $A[s], A[s + 1], ..., A[f]$ and do the following. For each location in A, if it was never initialized, we insert i. Otherwise, it has some value j in it, and so we insert (v_i, v_j) into E, and stop the scan for this interval. The total running time for all of the scans will be $O(\textit{cover-size}(I)+|I|) = O(\textit{cover-size}(I))$. Next, we find the connected components in G in $O(|I|)$ time. For each connected component C, we define I_C to be the subset of intervals in I which corresponds to the vertices in C.

Lemma 4. *Let C and C' be two different connected components of G. Then cover-size(I_C) + cover-size$(I_{C'})$ = cover-size$(I_C \cup I_{C'})$.*

Proof. If *cover-size*(I_C) + *cover-size*$(I_{C'})$ > *cover-size*$(I_C \cup I_{C'})$ (as the opposite is clearly not true) then this implies that there exists an interval with a vertex in C, and an interval with a vertex in C', such that the intersection of those two intervals is not empty. However, from the way we chose the edges in G, this implies that there is an edge between those two vertices, contradicting the assumption that C and C' are different connected components. □

For each connected component C we can convert I_C to be in standard form in time $O($*cover-size*$(I_C))$. So we will assume I_C is in standard form. We also assume with out loss of generality, that for every interval in I_C, its interactions with π prior to the batch insertion cause some changes (as this can be checked in $O($*cover-size*$(I_C))$ as well). Denote $I_C = \{(s_{i_1}, f_{i_1}), (s_{i_2}, f_{i_2}), ..., (s_{i_{|C|}}, f_{i_{|C|}})\}$, where for every $i_1 \leq j \leq i_{|C|} - 1$ we have $i_j < i_{j+1}$. For each $i_1 \leq j \leq i_{|C|}$ we do Insert(s_j, f_j) with the following changes. When we run the PST update traversal, we only run till we reach the suffix at s_{j+1}. Once this point is met, we know that the maximal extent from this point onwards is at least f_{j+1} which is larger than f_j. Being that at that point we are considering the shortened suffix T_{s_{j+1}, f_j} we can continue traversing down the tree in order to obtain the location of $T_{s_{j+1}, f_{j+1}}$, and then start the work needed for interval (s_{j+1}, f_{j+1}). Thus, the total time spent on interval j is $s_{j+1} - s_j$, and the total time spent on all intervals is

$$f_{|C|} - s_{|C|} + \sum_{j=i_1}^{j=i_{|C|-1}} s_{j+1} - s_j = O(\textit{cover-size}(I_C)).$$

For each connected component C we can convert I_C to be in standard form in time $O($*cover-size*$(I_C))$. So we will assume I_C is in standard form. We also assume that for every interval in I_C, its interactions with π prior to the batch insertion cause some changes (as this can be checked in $O($*cover-size*$(I_C))$ as well). Denote $I_C = \{(s_{i_1}, f_{i_1}), (s_{i_2}, f_{i_2}), ..., (s_{i_{|C|}}, f_{i_{|C|}})\}$, where for every $i_1 \leq j \leq i_{|C|} - 1$ we have $i_j < i_{j+1}$. For each $i_1 \leq j \leq i_{|C|}$ we do Insert(s_{i_j}, f_{i_j}) with the following changes. When we run the PST update traversal, we only run till we reach the suffix at $s_{i_{j+1}}$. Once this point is met, we know that the maximal extent from this point onwards is at least $f_{i_{j+1}}$ which is larger than f_{i_j}. Being that at that point we are considering the shortened suffix $T_{s_{i_{j+1}}, f_{i_j}}$ we can continue traversing down the tree in order to obtain the location of $T_{s_{i_{j+1}}, f_{i_{j+1}}}$, and then start the work needed for interval $(s_{i_{j+1}}, f_{i_{j+1}})$. Thus, the total time spent on interval i_j is $s_{i_{j+1}} - s_{i_j}$, and the total time spent on all intervals is

$$f_{|C|} - s_{|C|} + \sum_{j=i_1}^{j=i_{|C|-1}} s_{i_{j+1}} - s_{i_j} = O(\textit{cover-size}(I_C)).$$

Problem 5. Given a PST for text $T = t_1...t_n$ and a property π of T can maintain the PST under the following updates:

– Insert(I) - Insert the set of intervals $I = \{(s_1, f_1), (s_2, f_2), ..., (s_\ell, f_\ell)\}$ into π,

where each update takes $O(\textit{cover-size}(I)$ time.

6.1 The (New) Static Version

The static version is a special case of batched update, where we have only one update whose intervals are all the intervals in π. This reproves the following.

Theorem 10. *Given a property π over $T = t_1...t_n$ it is possible to construct the PST $O(|\pi| + m)$ time.*

References

1. Amir, A., Chencinski, E., Iliopoulos, C.S., Kopelowitz, T., Zhang, H.: Property matching and weighted matching. Theor. Comput. Sci. 395, 298–310 (2008)
2. Boyer, R.S., Moore, J.S.: A fast string searching algorithm. Comm. ACM 20, 762–772 (1977)
3. Cole, R., Gottlieb, L., Lewenstein, M.: Dictionary matching and indexing with errors and don't cares. In: Proc. 36th annual ACM Symposium on the Theory of Computing (STOC), pp. 91–100. ACM Press, New York (2004)
4. Farach, M., Muthukrishnan, S.: Perfect Hashing for Strings: Formalization and Algorithms. In: Proc. 7th Combinatorial Pattern Matching Conference, pp. 130–140 (1996)
5. Ferragina, P., Grossi, R.: Fast incremental text editing. In: Proc. 7th ACM-SIAM Symposium on Discrete Algorithms, pp. 531–540 (1995)
6. Gu, M., Farach, M., Beigel, R.: An efficient algorithm for dynamic text indexing. In: Proc. 5th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 697–704 (1994)
7. Iliopoulos, C.S., Rahman, M.S.: Faster index for property matching. Inf. Process. Lett. 105(6), 218–223 (2008)
8. Juan, M.T., Liu, J.J., Wang, Y.L.: Errata for "Faster index for property matching". Inf. Process. Lett. 109(18), 1027–1029 (2009)
9. Jurka, J.: Origin and Evolution of Alu Repetitive Elements. In: The Impact of Short Interspersed Elements (SINEs) on the Host Genome, pp. 25–41 (1995)
10. Jurka, J.: Human Repetitive Elements. In: Molecular Biology and Biotechnology, pp. 438–441 (1995)
11. 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)
12. Kopelwoitz, T., Lewenstein, M.: Dynamic Weighted Ancestors. In: Proc. 18th Annual ACM-SIAM Symposium on Discrete Algorithms(SODA), pp. 565–574 (2003)
13. Knuth, D.E., Morris, J.H., Pratt, V.R.: Fast pattern matching in strings. SIAM J. Comp. 6, 323–350 (1977)
14. McCreight, E.M.: A space-economical suffix tree construction algorithm. J. of the ACM 23, 262–272 (1976)

15. Sahinalp, S.C., Vishkin, U.: Efficient approximate and dynamic matching of patterns using a labeling paradigm. In: Proc. 37th FOCS, pp. 320–328 (1996)
16. Ukkonen, E.: On-line construction of suffix trees. Algorithmica 14, 249–260 (1995)
17. van Emde Boas, P.: Preserving Order in a Forest in Less Than Logarithmic Time and Linear Space. Inf. Process. Letters 6(3), 80–82 (1977)
18. Weiner, P.: Linear pattern matching algorithm. In: Proc. 14th IEEE Symposium on Switching and Automata Theory, pp. 1–11 (1973)

Approximate All-Pairs Suffix/Prefix Overlaps

Niko Välimäki[1,⋆], Susana Ladra[2,⋆⋆], and Veli Mäkinen[1,⋆⋆⋆]

[1] Department of Computer Science, University of Helsinki, Finland
{nvalimak,vmakinen}@cs.helsinki.fi
[2] Department of Computer Science, University of A Coruña, Spain
sladra@udc.es

Abstract. Finding approximate overlaps is the first phase of many sequence assembly methods. Given a set of r strings of total length n and an error-rate ϵ, the goal is to find, for all-pairs of strings, their suffix/prefix matches (overlaps) that are within edit distance $k = \lceil \epsilon \ell \rceil$, where ℓ is the length of the overlap. We propose new solutions for this problem based on *backward backtracking* (Lam et al. 2008) and *suffix filters* (Kärkkäinen and Na, 2008). Techniques use $nH_k + o(n \log \sigma) + r \log r$ bits of space, where H_k is the k-th order entropy and σ the alphabet size. In practice, methods are easy to parallelize and scale up to millions of DNA reads.

1 Introduction

High-throughput *short read sequencing* is revolutionizing the way molecular biology is researched. For example, the routine task of measuring gene expression by microarrays is now being replaced by a technology called *RNA-seq* [4,27]; the transcriptome is shotgun sequenced so that one is left with a set of short reads (typically e.g. of length 36 basepairs) whose sequence is known but it is not known from which parts of the genome they were transcribed. The process is hence reversed by mapping the short reads back to the genome, assuming that the reference genome sequence is known. Otherwise, one must resort to *sequence assembly* methods [24].

The *short read mapping* problem is essentially identical to an *indexed multiple approximate string matching* problem [21] when using a proper distance/similarity measure capturing the different error types (SNPs, measurement errors, etc.). Recently, many new techniques for short read mapping have come out building on the *Burrows-Wheeler transform (BWT)* [1] and on the *FM-index* [7] concept. The FM-index provides a way to index a sequence within space of compressed sequence exploiting BWT. This index provides so-called *backward search* principle that enables very fast exact string matching on the indexed sequence. Lam et al. [13] extended backward search to simulate backtracking on *suffix tree* [28], i.e., to simulate dynamic programming on all relevant paths of suffix tree; their tool `BWT-SW`

⋆ Funded by the Helsinki Graduate School in Computer Science and Engineering.
⋆⋆ Funded by MICINN grant TIN2009-14560-C03-02.
⋆⋆⋆ Funded by the Academy of Finland under grant 119815.

A. Amir and L. Parida (Eds.): CPM 2010, LNCS 6129, pp. 76–87, 2010.

provides an efficient way to do *local alignment* without the heuristics used in many common bioinformatics tools. The same idea of *backward backtracking* coupled with search space pruning heuristics is exploited in the tools tailored for short read mapping: `bowtie` [14], `bwa` [16], `SOAP2` [5]. In a recent study [17], an experimental comparison confirmed that the search space pruning heuristics used in short read mapping software are competitive with the fastest index-based filters — *suffix filters* [11] by Kärkkäinen and Na — proposed in the string processing literature.

In this paper, we go one step further in the use of backward backtracking in short read sequencing. Namely, we show that the technique can also be used when the reference genome is not known, i.e., as part of *overlap-layout-consensus* sequence assembly pipeline [12]. The overlap-phase of the pipeline is to detect all pairs of sequences (short reads) that have significant approximate overlap. We show how to combine suffix filters and backward backtracking to obtain a practical overlap computation method that scales up to millions of DNA reads.

2 Background

A *string* $S = S_{1,n} = s_1 s_2 \cdots s_n$ is a *sequence* of *symbols* (a.k.a. characters or letters). Each symbol is an element of an *alphabet* $\Sigma = \{1, 2, \ldots, \sigma\}$. A *substring* of S is written $S_{i,j} = s_i s_{i+1} \ldots s_j$. A *prefix* of S is a substring of the form $S_{1,j}$, and a *suffix* is a substring of the form $S_{i,n}$. If $i > j$ then $S_{i,j} = \varepsilon$, the empty string of length $|\varepsilon| = 0$. A *text* string $T = T_{1,n}$ is a string terminated by the special symbol $t_n = \$ \notin \Sigma$, smaller than any other symbol in Σ. The *lexicographical order* "$<$" among strings is defined in the obvious way. *Edit distance* $ed(T, T')$ is defined as the minimum number of insertions, deletions and replacements of symbols to transform string T into T' [15]. *Hamming distance* $h(T, T')$ is the number of mismatching symbols between strings T and T'.

The methods to be studied are derivatives of the *Burrows-Wheeler transform (BWT)* [1]. The transform produces a permutation of T, denoted by T^{bwt}, as follows: (i) Build the *suffix array* [19] $\mathsf{SA}[1, n]$ of T, that is an array of pointers to all the suffixes of T in the lexicographic order; (ii) The transformed text is $T^{bwt} = L$, where $L[i] = T[\mathsf{SA}[i] - 1]$, taking $T[0] = T[n]$. The BWT is reversible, that is, given $T^{bwt} = L$ we can obtain T as follows [1]: (a) Compute the array $C[1, \sigma]$ storing in $C[c]$ the number of occurrences of characters $\{\$, 1, \ldots, c-1\}$ in the text T; (b) Define the *LF mapping* as follows: $LF(i) = C[L[i]] + rank_{L[i]}(L, i)$, where $rank_c(L, i)$ is the number of occurrences of character c in the prefix $L[1, i]$; (c) Reconstruct T backwards as follows: set $s = 1$, for each $n - 1, \ldots, 1$ do $t_i \leftarrow L[s]$ and $s \leftarrow LF[s]$. Finally put the end marker $t_n \leftarrow \$$.

The *FM-index* [7] is a self-index based on the BWT. It is able to locate the interval $\mathsf{SA}[sp, ep]$ that contains the occurrences of any given pattern P without having SA stored. The FM-index uses an array C and function $rank_c(L, i)$ in the so-called *backward search* algorithm, calling the $rank_c(L, i)$ function $O(|P|)$ times. Its pseudocode is given below.

Algorithm. Count($P[1 \dots m]$,$L[1 \dots n]$)
(1) $i \leftarrow m$;
(2) $sp \leftarrow 1$; $ep \leftarrow n$;
(3) **while** $(sp \leq ep)$ **and** $(i \geq 1)$ **do**
(4) $\quad s \leftarrow P[i]$;
(5) $\quad sp \leftarrow C[s] + rank_s(L, sp-1)+1$;
(6) $\quad ep \leftarrow C[s] + rank_s(L, ep)$;
(7) $\quad i \leftarrow i-1$;
(8) **if** $(ep < sp)$ **return** "not found"
else return "found $(ep - sp + 1)$ occurrences".

The correctness of the algorithm is easy to see by induction: At each phase i, the range $[sp, ep]$ gives the maximal interval of SA pointing to suffixes prefixed by $P[i \dots m]$.

To report the occurrence positions $\mathsf{SA}[i]$ for $sp \leq i \leq ep$ a common approach is to sample SA values and then use the LF-mapping to derive the unsampled values from the sampled ones.

Many variants of the FM-index have been derived that differ mainly in the way the $rank_c(L, i)$-queries are solved [22]. For example, on small alphabets, it is possible to achieve $nH_k + o(n \log \sigma)$ bits of space, for moderate k, with constant time support for $rank_c(L, i)$ [8]. Here H_k is the standard *k-th order entropy*, i.e., the minimum number of bits to code a symbol once its k-symbol context is seen. There holds $H_k \leq \log \sigma$.

Let us denote by t_{LF} and t_{SA} the time complexities of LF-mapping (i.e. $rank_c(L, i)$ computation) and $\mathsf{SA}[i]$ computation, respectively.

3 All-Pairs Suffix/Prefix Matching

Given a set $\mathcal{T}$ of r strings $T^1, T^2, \dots, T^r$, of total length n, the *exact* all-pairs suffix/prefix matching problem is to find, for each ordered pair $T^i, T^j \in \mathcal{T}$, all nonzero length suffix/prefix matches (dubbed *overlaps*). The problem can be solved in optimal time by building a generalized suffix tree for the input strings:

Theorem 1 ([9, Sect. 7.10]). *Given a set $\mathcal{T}$ of r strings of total length n, let $r*$ be the number of exact suffix/prefix overlaps longer than a given threshold. All such overlaps can be found in $O(n + r*)$ time and in $\Theta(n \log n)$ bits of space.*

In the sequel, we concentrate on approximate overlaps and more space-efficient data structures. Instead of generalized suffix trees, the following techniques use a FM-index built on the concatenated sequence of strings in $\mathcal{T}$. Since all strings T^i contain the \$-terminator as their last symbol, the resulting BWT T^{bwt} contains all r terminators in some permuted order. This permutation is represented with an array D that maps from positions of \$s in T^{bwt} to strings in $\mathcal{T}$. Thus, the string T^i corresponding to a terminator $T^{bwt}[j] = \$$ is $i = D[rank_{\$}(T^{bwt}, j)]$. The array requires $d \log d$ bits.

Next subsection introduces a basic backtracking algorithm that can find approximate overlaps within a fixed distance k. The second subsection describes a filtering method that is able to find approximate overlaps when the maximum number of errors depends on length of the overlap.

3.1 Backward Backtracking

The backward search can be extended to *backtracking* to allow the search for approximate occurrences of the pattern [13]. To get an idea of this general approach, let us first concentrate on the *k-mismatches* problem: The pattern $P_{1,m}$ approximately matches a substring $X_{1,m}$ of some string $T^i \in \mathcal{T}$, if there are at most k indices i such that $P[i] \neq X[i]$ (i.e. Hamming distance $h(P, X) \leq k$). The following pseudocode finds the k-mismatch occurrences, and is analogous to the schemes used in [14,16]. The first call to the recursive procedure is $\texttt{kmismatches}(P, T^{bwt}, k, m, 1, n)$.

Algorithm. $\texttt{kmismatches}(P, L, k, j, sp, ep)$
(1) **if** $(sp > ep)$ **return** ;
(2) **if** $(j = 0)$
(3) Report $\mathsf{SA}[sp], \ldots, \mathsf{SA}[ep]$; **return** ;
(4) **for each** $s \in \Sigma$ **do**
(5) $sp' \leftarrow C[s] + rank_s(L, sp - 1)+1$;
(6) $ep' \leftarrow C[s] + rank_s(L, ep)$;
(7) **if** $(P[j] \neq s)$ $k' \leftarrow k - 1$; **else** $k' \leftarrow k$;
(8) **if** $(k' \geq 0)$ $\texttt{kmismatches}(P, L, k', j - 1, sp', ep')$;

The difference between the `kmismatches` algorithm and exact searching is that the recursion considers incrementally, from right to left, all different ways the pattern can be altered with at most k substitutions. Simultaneously, the recursion maintains the suffix array interval $\mathsf{SA}[sp \ldots ep]$ where suffixes match the current modified suffix of the pattern.

To find approximate overlaps of T^i having at most k mismatches, we call $\texttt{kmismatches}(T^i, T^{bwt}, k, |T^i|, 1, n)$ and modify the algorithm's output as follows. Notice that, at each step, the range $T^{bwt}[sp \ldots ep]$ contains \$-terminators of all strings prefixed (with at most k mismatches) by the suffix $T^i_{j,m}$ where $m = |T^i|$. Thus, each of the terminators correspond to one valid overlap of length j. Terminators and their respective strings $T^{i'}$ can be enumerated from the array D in constant time per identifier; the identifiers i' to output are in the range $D[rank_\$(T^{bwt}, sp) \ldots rank_\$(T^{bwt}, ep)]$.

The worst case complexity of backward backtracking is $O(|\Sigma|^k m^{k+1} t_{\mathsf{LF}})$. There are several recent proposals to prune the search space [14,16] but none of them can be directly adapted to this suffix/prefix matching problem.

To find all-pairs approximate overlaps, the k-mismatch algorithm is called for each string $T^i \in \mathcal{T}$ separately. Thus, we obtain the following result:

Theorem 2. *Given a set $\mathcal{T}$ of r strings of total length n, and a distance k, let $r*$ be the number of approximate suffix/prefix overlaps longer than a given threshold and within Hamming distance k. All such approximate overlaps can be found in $O(\sigma^k \sum_{T \in \mathcal{T}} |T|^{k+1} t_{\mathsf{LF}} + r*)$ time and in $nH_k + o(n \log \sigma) + r \log r$ bits of space.*

From the above theorem, it is straightforward to achieve a space-efficient and easily parallelizable solution for the exact all-pairs suffix/prefix matching problem (cf. Theorem 1):

Corollary 1. *Given a set $\mathcal{T}$ of r strings of total length n, let $r*$ be the number of exact suffix/prefix overlaps longer than a given threshold. All such overlaps can be found in $O(nt_{\mathsf{LF}} + r*)$ time and in $nH_k + o(n \log \sigma) + r \log r$ bits of space.*[1]

When *k-errors* searching (edit distance in place of Hamming distance) is used instead of k-mismatches, one can apply dynamic programming by building one column of the standard dynamic programming table [26] on each recursive step. Search space can be pruned by detecting the situation when the minimum value in the current column exceeds k. To optimize running time, one can use Myers' bit-parallel algorithm [20] with the bit-parallel witnesses technique [10] that enables the same pruning condition as the standard computation. We omit the details for brevity.

3.2 Suffix Filters

We build on *suffix filters* [11] and show two different ways to modify the original idea to be able to search for approximate overlaps. Let us first describe a simplified version of the original idea using an example of approximate matching of string P with edit distance k.

Suffix filter splits the string to be searched, here P of length m, into $k+1$ pieces. More concretely, let string P be partitioned into pieces $P = \alpha_1 \alpha_2 \cdots \alpha_{k+1}$. Because the FM-index is searched backwards, it is more convenient to talk about *prefix* filters in this context. Now the set of *filters* to be considered is $\mathcal{S} = \{\alpha_1 \alpha_2 \cdots \alpha_{k+1},\ \alpha_1 \alpha_2 \cdots \alpha_k,\ \ldots,\ \alpha_1\}$ as visualized in Fig. 1. To find *candidate* occurrences of P within edit distance k, each filter $S \in \mathcal{S}$ is matched against T as follows. We use backward backtracking (Sect. 3.1) and match pieces of the filter S starting from the last one with distance $k' = 0$. When the backtracking advances from one piece to next one (i.e. the preceding piece), the number of allowed errors k' is increased by one. Figure 1 gives a concrete example on how k' increases. If there is an occurrence of P within distance k, at least one of the filters will output it as an candidate [11]. In the end, all candidate occurrences must be validated since the filters may find matches having edit distance larger than k. However, suffix filters have been shown to be one of the strongest filters producing quite low number of wrong candidates [11].

Approximate suffix/prefix matches of $T^i \in \mathcal{T}$ can be found as follows. Instead of a fixed distance k, we are given two parameters: an *error-rate* $\epsilon \leq 1$ and a minimum overlap threshold $t \geq 1$. Now an overlap of length ℓ is called *valid* if it is within edit distance $\lceil \epsilon \ell \rceil$ and $\ell \geq t$. Again, the string T^i is partitioned into

[1] Notice that a stronger version of the algorithm in [9, Sect. 7.10] (the one using doubly-linked stacks) can be modified to find $r' < r^2$ pairs of strings with *maximum* suffix/prefix overlap longer than a given threshold. We can simulate that algorithm space-efficiently replacing doubly-linked stacks with dynamic compressed bit-vectors [18] so that time complexity becomes $O(n(t_{\mathsf{SA}} + \log n) + r')$ and space complexity becomes $nH_k + o(n \log \sigma) + r \log r + n(1 + o(1))$. We omit the details, as we focus on the approximate overlaps. A stronger variant for approximate overlaps is an open problem.

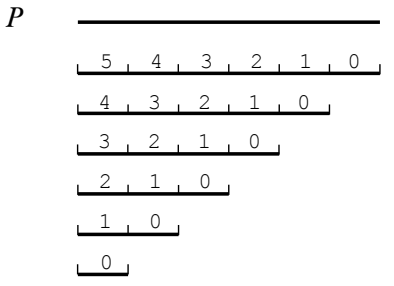


Fig. 1. Prefix filters for a string P that has been partitioned into even length pieces. Numbers correspond to maximum number of errors allowed during backward search.

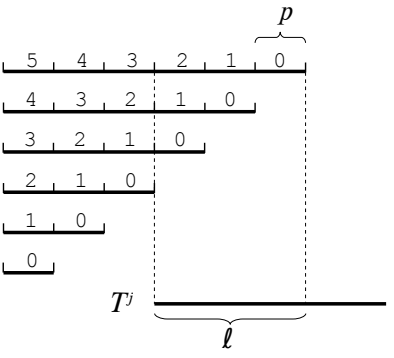


Fig. 2. String T^i has an overlap of length $\ell = 3p$ with T^j. One of the first three filters is bound to find the overlap during backward search.

pieces, denoted α_i, but now the number of pieces is determined by the threshold t and error-rate ϵ. Let $k = \lceil \epsilon t \rceil$ be the maximum number of errors allowed for the shortest overlap possible, and for simplicity, let us assume that all pieces are of even length p (to be defined later). Now the number of pieces is $h = \lceil |T^i|/p \rceil$.

Candidate overlaps are found by searching each prefix filter $S^i = \alpha_1 \alpha_2 \cdots \alpha_i$ for $1 \leq i \leq h$ separately: start the backward search from the end of the last piece α_i and match it exactly. Each time a boundary of two pieces is crossed, the number of allowed errors is increased by one. Now assume that pieces from i to jth piece have been processed, that is, the current range $[sp \ldots ep]$ corresponds to pieces $\alpha_j \alpha_{j+1} \cdots \alpha_i$. Before the backward search crosses the boundary from the piece α_j to α_{j-1}, we check the range $T^{bwt}[sp \ldots ep]$ and output those \$-terminators as candidate overlaps. These candidates are prefixes of strings in $\mathcal{T}$ that *may* be valid approximate overlaps of length $p \cdot (h - j + 1)$. Only overlaps whose lengths are multiples of the piece length p can be obtained.

We give two different strategies to find all approximate overlaps, not just those with length $p, 2p, 3p, \ldots$ But first, let us prove that the final set of candidates produced by the above method contains all valid overlaps of length pj for any $j \geq \lceil t/p \rceil$ (recall that valid overlaps must be longer than t).

Assume that there is a valid overlap of length $\ell = pj$ between T^i and some T^j, as displayed in Fig. 2. Prefix filters of T^i will locate this occurrence if we can guarantee that the suffix $T^i_{m-\ell,m}$ has been partitioned into $\lceil \epsilon \ell \rceil + 1$ pieces, where $\lceil \epsilon \ell \rceil$ gives the maximum edit distance for an overlap of length ℓ. Recall that in our partition the suffix $T^i_{m-\ell,m}$ was split into pieces of length p. We can define p as $\min_{\ell=t}^{|T^i|} \lceil \frac{\ell}{\lceil \epsilon \ell \rceil + 1} \rceil$. This guarantees that we have chosen short enough pieces for our partition, as at least one of the filters $S^h, S^{h-1}, \ldots, S^{h-j+1}$ will output the string T^j as a candidate overlap. Figure 2 illustrates this idea. In the

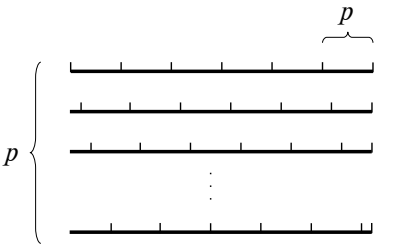


Fig. 3. Strategy I produces p different partitions of T^i

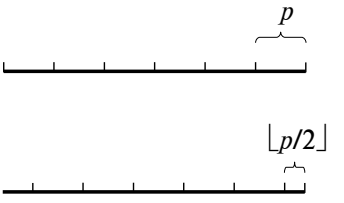


Fig. 4. Strategy II produces two different partitions of T^i

end, all candidate overlaps must be validated since some of the candidates may not represent a valid approximate overlap.

Strategy I produces p different partitions for T^i so that the boundaries (start position of pieces) cover all indices of T^i. For simplicity, assume that $m = |T^i|$ is a multiple of p. The jth partition, $1 \leq j \leq p$, has boundaries $\{j, p+j, 2p+j, \ldots, m\}$. As a result, the very last piece "shrinks" as seen in Fig. 3. Each partition forms its own set of filters, which are then searched as described above. It is straightforward to see that filters of the jth partition find all overlaps of lengths $\ell \in \{p-j+1, 2p-j+1, 3p-j+1, \ldots, m-j+1\}$. Thus, all overlap lengths $\ell \geq t$ are covered by searching through all p partitions. Advantage of this strategy is that during the backward search, we can always match p symbols (with 0-errors) before we check for candidate matches. The "shrinking" last piece α_h can be shorter than p but it never produces candidates since $p \leq t$. Downside is that the number of different filter sets S^i to search for grows to p.

Strategy II produces only two partitions for T^i. Again, assume that $m = |T^i|$ is a multiple of p. Now the two partitions have the boundaries $\{1, p+1, 2p+1, \ldots, m\}$ and $\{\lceil p/2 \rceil, p+\lceil p/2 \rceil, 2p+\lceil p/2 \rceil, \ldots, m\}$, as visualized in Fig. 4. To acquire candidates for all overlap lengths $\ell \geq t$, we modify the backtracking search as follows: instead of outputting candidates only at the boundaries, we start to output candidates after $\lceil p/2 \rceil$ symbols of each piece has been matched. More precisely, assume we are matching symbol at position i' in some α_i. If $i' \leq p - \lceil p/2 \rceil$,we output all \$-terminators from range $T^{bwt}[sp \ldots ep]$ as candidate overlaps. Then the first partition outputs candidates for overlap lengths $\ell \in [\lceil p/2 \rceil, p] \bigcup [p+\lceil p/2 \rceil, 2p] \bigcup \cdots$ and the second partition for lengths $\ell \in [p+1, p+\lceil p/2 \rceil] \bigcup [2p+1, 2p+\lceil p/2 \rceil] \bigcup \cdots$ Since $\lceil p/2 \rceil \leq t$, these filters together cover all overlap lengths $\ell \geq t$. Obvious advantage of this strategy is that only two sets of filters must be searched. However, the number of candidates produced is generally higher than in strategy I. If p is really small, the number of candidates found after $\lceil p/2 \rceil$ symbols grows substantially.

Unfortunately, prefix filters cannot guarantee any worst-case time complexities. We conclude with the following theorem:

Table 1. Experiments with *k-mismatches*. Time is reported as average time (s) per read. Strategy II produces exactly the same overlaps as strategy I.

Method	t	k	ϵ	Time (s)	Max. ℓ	Avg. ℓ	Std.dev. ℓ
Backtracking	20	2	–	0.005	506	33.9	24.0
	20	4	–	0.277	506	27.4	16.4
	20	6	–	≈ 8	*full result not computed*		
Strategy I	20	–	5%	0.365	524	42.1	34.5
	20	–	10%	0.753	1040	46.5	38.1
	40	–	2.5%	0.212	506	74.8	45.6
	40	–	5%	0.213	524	76.7	45.7
	40	–	10%	0.553	1040	78.8	46.4
Strategy II	20	–	5%	0.140	524	42.1	34.5
	20	–	10%	0.990	1040	46.5	38.1
	40	–	2.5%	0.029	506	74.8	45.6
	40	–	5%	0.053	524	76.7	45.7
	40	–	10%	0.341	1040	78.8	46.4

Table 2. Experiments with *k-errors*. Time is reported as average time (s) per read. Strategy II produces exactly the same overlaps as strategy I.

Method	t	k	ϵ	Time (s)	Max. ℓ	Avg. ℓ	Std.dev. ℓ
Backtracking	40	2	–	0.031	535	77.2	49.4
	40	4	–	≈ 6	*full result not computed*		
Strategy I	40	–	2.5%	1.196	561	116.1	80.9
	40	–	5%	1.960	1010	121.4	82.2
	40	–	10%	≈ 6	1040	123.9	80.5
Strategy II	40	–	2.5%	0.072	561	116.1	80.9
	40	–	5%	0.179	1010	121.4	82.2
	40	–	10%	1.730	1040	123.9	80.5

Theorem 3. *Given a set $\mathcal{T}$ of r strings of total length n, a minimum overlap threshold $t \geq 1$ and an error-rate ϵ, all* approximate *overlaps within edit distance $\lceil \epsilon \ell \rceil$, where ℓ is the length of the overlap, can be found using prefix filters and in $nH_k + o(n \log \sigma) + r \log r$ bits of space.*

4 Experiments

We implemented the different techniques described in Sect. 3 on top of succinct data structures from the *libcds* library[2]. The implementation supports both the *k-mismatches* and *k-errors* (i.e. edit distance) models. Edit distance computation is done using bit-parallel dynamic programming [20]. Overlaps can be searched by using either the backtracking algorithm (for fixed k) or suffix filters (for error-rate ϵ). The experiments were run on Intel Xeon E5440 and 32 GB of memory.

[2] `http://code.google.com/p/libcds/`

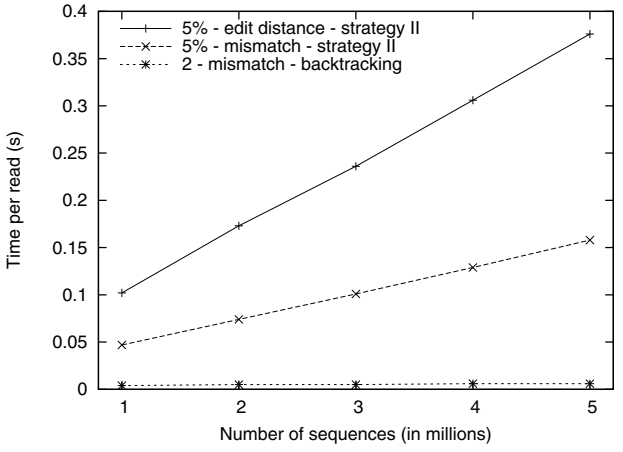


Fig. 5. Average time per read when the number of sequences increases from 1 to 5 million. The average times for $\epsilon = 5\%$ (both edit distance and mismatches) were measured using strategy II with minimum overlap length $t = 40$. All averages were measured by matching 10 000 reads against each set.

The algorithms were tested on sets of DNA sequences produced by a *454 Sequencing System* [2]. All of the DNA was sequenced from one individual *Melitaea cinxia* (a butterfly). Since the 454 system is known to produce sequencing errors in long *homopolymers* (runs of single nucleotide) [6], all homopolymers longer than 5 were truncated. The full set contained 5 million reads of total length 1.7 GB. The average read length was 355.1 with a standard deviation of 144.2. Smaller sets of 4, 3, 2, and 1 million reads were produced by cutting down the full set. Majority of these experiments were run using the smallest set of one million reads to allow extensive coverage of different parameters in feasible time.

Our implementation of the suffix filters uses extra $n \log \sigma + O(d \log \frac{u}{d})$ bits (plain sequences plus a delta-encoded bit-vector in main memory) to be able to check candidate matches more efficiently. In practice, the total size of the index for the sets of 5 and 1 million reads was 2.8 GB and 445 MB, respectively. A minimum overlap length $t \in \{20, 40\}$ was used to limit the output size. Furthermore, results were post-processed to contain only the longest overlaps for each ordered string pair.

Table 1 summarizes our results on k-mismatch overlaps for the set of one million reads. As expected, backtracking slows down exponentially and does not scale up to high values of k. The parameter $k = 4$ corresponds approximately to $0.7\% \leq \epsilon \leq 20\%$. Strategy I is faster than strategy II when the piece length gets small ($\epsilon = 10\%$ and $t = 20$). On all other parameters, however, it is more efficient to check the candidates produced by the two filters in strategy II, than to search through all partitions in strategy I. Notice that strategy II ($\epsilon = 5\%$ and $t = 40$) is only about 10 times slower than $k = 2$ but produces a significantly bigger quantity of long overlaps (cf. Fig. 6). Against $k = 4$, strategy II is on

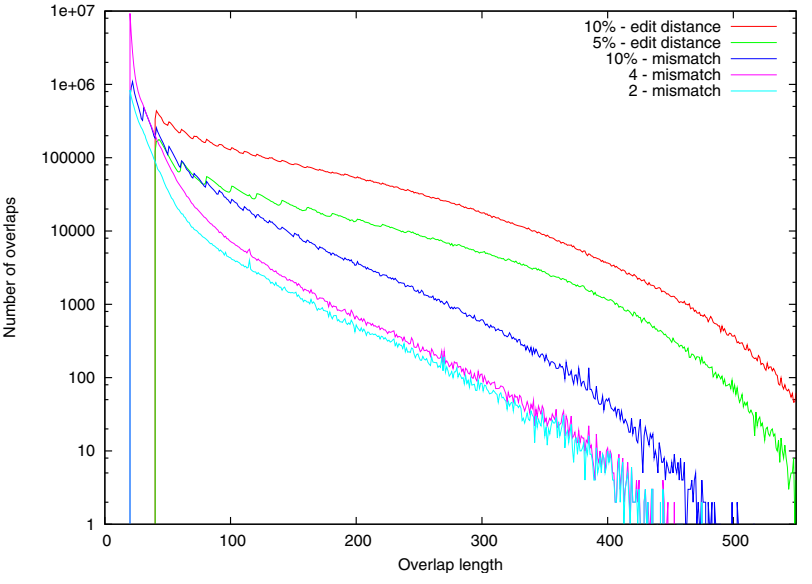


Fig. 6. Graph of overlap lengths for different error-rates ϵ and k-mismatches over a set of one million reads. The mismatch curves $\epsilon = 10\%$ and $k = 4$ cross each other at overlap lengths ℓ where $k = \lceil \epsilon \ell \rceil$. The y-axis is logarithmic.

par regarding time (when $t = 40$) and produces longer overlaps. Table 2 gives numbers for similar tests in the k-errors model.

In our third experiment, we measured the average time as a function of the number of sequences. Figure 5 gives the average times per read for backtracking with 2-mismatch and suffix filters with $\epsilon = 5\%$ and $t = 40$. The suffix filters, for both edit distance and mismatch, slow down by a factor of ≈ 3.5 between the smallest and largest set. The backtracking algorithm slows down only by a factor of ≈ 1.5.

The graph in Fig. 6 displays the frequencies of overlap lengths computed with the different k and ϵ parameters. Notice that increasing k from 2 to 4 mismatches mainly increases the number of short overlaps. Overlaps computed using error-rate give a much gentle distribution of overlaps, since they naturally allow less errors for shorter overlaps. Furthermore, at overlap lengths 100–400, the 10%-mismatch search finds about 5 times more overlaps than methods with fixed k. When searching with 10%-edit distance, there are more than a hundred times more overlaps of length 300 compared to the 2-mismatch search. This suggests that insertions and deletions (especially at homopolymers) are frequent in the dataset.

5 Discussion

Currently, many state-of-the-art sequence assemblers for short read sequences (e.g. [23,29,3]) use de Bruijn graph alike structures that are based on the q-grams

shared by the reads. It will be interesting to see whether starting instead from the overlap graph (resulting from the approximate overlaps studied in this paper), and applying the novel techniques used in the de Bruijn appoaches, yields a competitive assembly result. Such pipeline is currently under implementation [25].

Acknowledgments

We wish to thank Richard Durbin, Jared T. Simpson, Esko Ukkonen and Leena Salmela for insightful discussions, and Jouni Sirén for implementing the bit-parallel techniques.

DNA sequences were provided by The Metapopulation Research Group/The Glanville Fritillary Butterfly Genome and Population Genomics Project: Rainer Lehtonen[3], Petri Auvinen[4], Liisa Holm[5], Mikko Frilander[6], Ilkka Hanski[3], funded by ERC (232826) and the Academy of Finland (133132).

References

1. Burrows, M., Wheeler, D.: A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation (1994)
2. Roche Company. 454 life sciences, `http://www.454.com/`
3. Simpson, J.T., et al.: Abyss: A parallel assembler for short read sequence data. Genome Res. 19, 1117–1123 (2009)
4. Morin, R.D., et al.: Profiling the hela s3 transcriptome using randomly primed cdna and massively parallel short-read sequencing. BioTechniques 45(1), 81–94 (2008)
5. Li, R., et al.: Soap2. Bioinformatics 25(15), 1966–1967 (2009)
6. Wicker, T., et al.: 454 sequencing put to the test using the complex genome of barley. BMC Genomics 7(1), 275 (2006)
7. Ferragina, P., Manzini, G.: Indexing compressed texts. Journal of the ACM 52(4), 552–581 (2005)
8. Ferragina, P., Manzini, G., Mäkinen, V., Navarro, G.: Compressed representations of sequences and full-text indexes. ACM Transactions on Algorithms (TALG) 3(2), article 20 (2007)
9. Gusfield, D.: Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology. Cambridge University Press, Cambridge (1997)
10. Hyyrö, H., Navarro, G.: Bit-parallel witnesses and their applications to approximate string matching. Algorithmica 41(3), 203–231 (2005)
11. Kärkkäinen, J., Na, J.C.: Faster filters for approximate string matching. In: Proc. ALENEX 2007, pp. 84–90. SIAM, Philadelphia (2007)

[3] Metapopulation Research Group, Department of Biological and Environmental Sciences, University of Helsinki.
[4] DNA Sequencing and Genomics Laboratory, Institute of Biotechnology, University of Helsinki.
[5] Institute of Biotechnology and Department of Biological and Environmental Sciences, University of Helsinki.
[6] Institute of Biotechnology and Metapopulation Research Group, University of Helsinki.

12. Kececioglu, J.D., Myers, E.W.: Combinatorial algorithms for dna sequence assembly. Algorithmica 13, 7–51 (1995)
13. Lam, T.W., Sung, W.K., Tam, S.L., Wong, C.K., Yiu, S.M.: Compressed indexing and local alignment of dna. Bioinformatics 24(6), 791–797 (2008)
14. Langmead, B., Trapnell, C., Pop, M., Salzberg, S.L.: Ultrafast and memory-efficient alignment of short dna sequences to the human genome. Genome Biology 10(3), R25 (2009)
15. Levenshtein, V.I.: Binary codes capable of correcting deletions, insertions and reversals. Soviet Physics Doklady 10(8), 707–710 (1966)
16. Li, H., Durbin, R.: Fast and accurate short read alignment with burrows-wheeler transform. Bioinformatics (2009), Advance access
17. Mäkinen, V., Välimäki, N., Laaksonen, A., Katainen, R.: Unifying view of backward backtracking in short read mapping. In: Elomaa, T., Mannila, H., Orponen, P. (eds.) LNCS Festschrifts. Springer, Heidelberg (to appear 2010)
18. Mäkinen, V., Navarro, G.: Dynamic entropy-compressed sequences and full-text indexes. ACM Transactions on Algorithms 4(3) (2008)
19. Manber, U., Myers, G.: Suffix arrays: a new method for on-line string searches. SIAM Journal on Computing 22(5), 935–948 (1993)
20. Myers, G.: A fast bit-vector algorithm for approximate string matching based on dynamic programming. J. ACM 46(3), 395–415 (1999)
21. Navarro, G.: A guided tour to approximate string matching. ACM Comput. Surveys 33(1), 31–88 (2001)
22. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Computing Surveys 39(1), article 2 (2007)
23. Pevzner, P., Tang, H., Waterman, M.: An eulerian path approach to dna fragment assembly. Proc. Natl. Acad. Sci. 98(17), 9748–9753 (2001)
24. Pop, M., Salzberg, S.L.: Bioinformatics challenges of new sequencing technology. Trends Genet. 24, 142–149 (2008)
25. Salmela, L.: Personal communication (2010)
26. Sellers, P.: The theory and computation of evolutionary distances: Pattern recognition. Journal of Algorithms 1(4), 359–373 (1980)
27. Wang, Z., Gerstein, M., Snyder, M.: Rna-seq: a revolutionary tool for transcriptomics. Nature Reviews Genetics 10(1), 57–63 (2009)
28. Weiner, P.: Linear pattern matching algorithm. In: Proc. 14th Annual IEEE Symposium on Switching and Automata Theory, pp. 1–11 (1973)
29. Zerbino, D.R., Birney, E.: Velvet: algorithms for de novo short read assembly using de bruijn graphs. Genome Research 18(5), 821–829 (2008)

Succinct Dictionary Matching with No Slowdown

Djamal Belazzougui

LIAFA, Univ. Paris Diderot - Paris 7, 75205 Paris Cedex 13, France
dbelaz@liafa.jussieu.fr

Abstract. The problem of dictionary matching is a classical problem in string matching: given a set S of d strings of total length n characters over an (not necessarily constant) alphabet of size σ, build a data structure so that we can match in a any text T all occurrences of strings belonging to S. The classical solution for this problem is the Aho-Corasick automaton which finds all *occ* occurrences in a text T in time $O(|T| + occ)$ using a representation that occupies $O(m \log m)$ bits of space where $m \leq n + 1$ is the number of states in the automaton. In this paper we show that the Aho-Corasick automaton can be represented in just $m(\log \sigma + O(1)) + O(d \log(n/d))$ bits of space while still maintaining the ability to answer to queries in $O(|T| + occ)$ time. To the best of our knowledge, the currently fastest succinct data structure for the dictionary matching problem uses $O(n \log \sigma)$ bits of space while answering queries in $O(|T| \log \log n + occ)$ time. In the paper we also show how the space occupancy can be reduced to $m(H_0 + O(1)) + O(d \log(n/d))$ where H_0 is the empirical entropy of the characters appearing in the trie representation of the set S, provided that $\sigma < m^{\varepsilon}$ for any constant $0 < \varepsilon < 1$. The query time remains unchanged.

1 Introduction

A recent trend in text pattern matching algorithms has been to succinctly encode data structures so that they occupy no more space than the data they are built on, without a too significant sacrifice in their query time. The most prominent example being the data structures used for indexing texts for substring matching queries [15, 8, 9].

In this paper we are interested in the succinct encoding of data structures for the dictionary matching problem, which consists in the construction of a data structure on a set S of d strings (a dictionary) of total length n over an alphabet of size σ (wlog we assume that $\sigma \leq n$) so that we can answer to queries of the kind: find in a text T all occurrences of strings belonging to S if any. The dictionary matching problem has numerous applications including computer security (virus detection software, intrusion detection systems), genetics and others. The classical solution to this problem is the Aho-Corasick automaton [1], which uses space $O(m \log m)$ bits (where m is the number of states in the automaton which in the worst case equals $n + 1$) and answers queries in time $O(|T| + occ)$ (where *occ* is number of occurrences) if hashing techniques are

A. Amir and L. Parida (Eds.): CPM 2010, LNCS 6129, pp. 88–100, 2010.

used, or $O(|T| \log \sigma + occ)$ if only binary search is permitted. The main result of our paper is that the Aho-corasick automaton can be represented in just $m(\log \sigma + 3.443 + o(1)) + d(3 \log(n/d) + O(1))$ bits of space while still maintaining the same $O(|T| + occ)$ query time. As a corollary of the main result, we also show a compressed representation suitable for alphabets of size $\sigma < m^{\varepsilon}$ for any constant $0 < \varepsilon < 1$. This compressed representation uses $m(H_0 + 3.443 + o(1)) + O(d \log(n/d))$ bits of space where H_0 is the empirical entropy of the characters appearing in the trie representation of the set S. The query time of the compressed representation is also $O(|T| + occ)$.

The problem of succinct encoding for dictionary matching has already been explored in [2, 10, 16, 11]. The results in [2] and [11] deal with the dynamic case which is not treated in this paper. The best results for the static case we have found in the literature are the two results from [10] and the result from [16]. A comparison of the results from [10, 16] with our main result is summarized in table 1 (the two dynamic results of [2, 11] are not shown as in the static case they are dominated by the static results in [10, 16]). In this table, the two results from [10] are denoted by HLSTV1 and HLSTV2 and the result of [16] is denoted by TWLY. For comparison purpose, we have replaced m with n in the space and time bounds of our data structure.

Table 1. Comparison of dictionary matching succinct indexes

Algorithm	Space usage (in bits)	Query time
HLSTV1	$O(n \log \sigma)$	$O(\|T\| \log \log(n) + occ)$
HLSTV2	$n \log \sigma(1 + o(1)) + O(d \log(n))$	$O(\|T\|(\log^{\varepsilon}(n) + \log(d)) + occ)$
TWLY	$n \log \sigma(2 + o(1)) + O(d \log(n))$	$O(\|T\|(\log(d) + \log \sigma) + occ)$
Ours	$n(\log \sigma + 3.443 + o(1)) + O(d \log(n/d))$	$O(\|T\| + occ)$

Our results assume a word RAM model, in which usual operations including multiplications, divisions and shifts are all supported in constant time. We assume that the computer word is of size $\Omega(\log n)$, where n is the total size of the string dictionary on which we build our data structures. Without loss of generality we assume that n is a power of two. All logarithms are intended as base 2 logarithms. We assume that the strings are drawn from an alphabet of size σ, where σ is not necessarily constant. That is, σ could be as large as n.

The paper is organized as follows: in section 2, we present the main tools that will be used in our construction. In section 3 we present our main result. In section 4, we give a compressed variant of the data structure. Finally some concluding remarks are given in section 5.

2 Basic Components

In this paper, we only need to use three basic data structures from the literature of succinct data structures.

2.1 Compressed Integer Arrays

We will use the following result about compression of integer arrays:

Lemma 1. *Given an array A of n integers such that $\sum_{0 \leq i < n} A[i] = U$. We can produce a compressed representation that uses $n(\lceil \log(U/n) \rceil + 2 + o(1))$ bits of space such that any element of the array A can be reproduced in constant time.*

This result was first described in [9] based on Elias-Fano coding by Elias [5] and Fano [6] combined with succinct bitvectors [3] which support constant time queries.

2.2 Succinctly Encoded Ordinal Trees

In the result of [13] a tree of n nodes of arbitrary degrees where the nodes are ordered in depth first order can be represented in $n(2 + o(1))$ bits of space so that basic navigation on the tree can be done in constant time. In this work we will only need a single primitive: given a node x of preorder i (the preorder of a node is the number attributed to the node in a DFS lexicographic traversal of the tree), return the preorder j of the parent of x.

The following lemma summarizes the result which will be used later in our construction:

Lemma 2. *A tree with n nodes of arbitrary degrees can be represented in $n(2 + o(1))$, so that the preorder of the parent of a node of given preorder can be computed in constant time.*

In this paper we also use the compressed tree representation presented in [12] which permits to use much less space than $2n + o(n)$ bits in the case where tree nodes degrees distribution is skewed (e.g. the tree has much more leaves than internal nodes).

Lemma 3. *A tree with n nodes of arbitrary degrees can be represented in $n(H^* + o(1))$, where H^* is the entropy of the degree distribution of the tree, so that the preorder of the parent of a node of given preorder can be computed in constant time.*

2.3 Succinct Indexable Dictionary

In the paper by Raman, Raman and Rao [14] the following result is proved :

Lemma 4. *a dictionary on a set Γ of m integer keys from a universe of size U can be built in time $O(m)$ and uses $B(m, U) + o(m)$ bits of space, where $B(m, U) = \log \binom{U}{m}$, so that the following two operations can be supported in constant time:*

- *select(i): return the key of rank i in lexicographic order (natural order of integers).*
- *rank(k): return the rank of key k in lexicographic order if $k \in \Gamma$. Otherwise return -1.*

The term $B(m,U) = \log \binom{U}{m}$ is the information theoretic lower bound on the number of bits needed to encode all possible subsets of size n of a universe of size U(we have $\binom{U}{m}$ different subsets and so we need $\log \binom{U}{m}$ to encode them). The term $B(m,U)$ can be upper bounded in the worst case by $m(\log(e)+\log(U/m))$. The space usage of the dictionary can then be simplified as $B(m,U) + o(m) \leq m(\log(e) + \log(U/m) + o(1)) \leq m(\log(U/m) + 1.443 + o(1))$.

3 The Data Structure

Before describing our new representation, we briefly recall the original Aho-Corasick automaton. The variant described here may slightly differ from other ones for the reason that this variant is simpler to adapt to our case. In particular the strings of S are implicitly represented by the automaton and are never represented explicitly.

Given a set of strings S, our Aho-Corasick automaton has $m = |P|$ states where P is the set of all prefixes of strings in S including the empty string and all the strings of S. Each state of the automaton uniquely corresponds to one of the elements of P. We thus have $|P| = m \leq n + 1$ states in the automaton. The states that correspond to strings in S are called terminal states. Our Aho-Corasick representation has three kinds of transitions: $next$,$failure$ and $report$:

- For each state v_p corresponding to a prefix p, we have a transition $next(v_p, c)$ labeled with character c from the state v_p to a state v_{pv} corresponding to a prefix pc for each prefix $pc \in P$. Hence we may have up to σ $next$ transitions per state.
- For each state v_p we have a failure transition which connects v_p to the state v_q corresponding to the longest suffix q of p such that $q \in P$ and $p \neq q$.
- Additionally , for each state v_p, we may have a $report$ transition from the state v_p to the state corresponding to the longest suffix q of p such that $q \in S$ and $p \neq q$ if such q exists (a report transition always points to a terminal state). If for a given state v_p no such string exists, then we do not have a report transition from the state v_p.

Our new data structure is very simple. We essentially use two succinctly encoded dictionaries, two succinctly encoded ordinal trees and one Elias-Fano encoded array. The representation we use is implicit in the sense that the strings of the dictionary are not stored at all. A query will output the occurences as triplets of the form ($occ\_start\_pos, occ\_end\_pos, string\_id$) where $string\_id$ is the identifier of a matched string from S and occ_start_pos (occ_end_pos) is the starting (ending) position of the occurrence in the text.

The central idea is to represent each state corresponding to a prefix $p \in P$, by a unique number $rank_P(p) \in [0, m-1]$ which represents the rank of p in P in suffix-lexicographic order (the suffix-lexicographic order is similar to lexicographic order except that the strings are compared in right-to-left order instead of left-to-right order). Then it is easy to see that the $failure$ transitions form a tree rooted at state 0 (which we call a failure tree) and a DFS traversal of

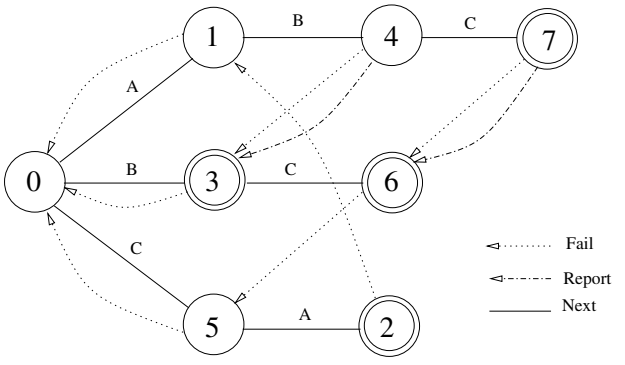


Fig. 1. The Aho-Corasick automaton for the set {"ABC","B","BC","CA"}

the tree will enumerate the states in increasing order. Similarly, the set of *report* transitions represent a forest of trees, which can be transformed into a tree rooted at state 0 (which we call a report tree) by attaching all the roots of the forest as children of state 0. Then similarly a DFS traversal of the report tree will also enumerate the states of the automaton in order. Then computing a *failure* (*report*) transition for a given state amounts to finding the parent of the state in the failure (report) tree. It turns out that the succinct tree representations (lemma 2 and lemma 3) do support parent queries on DFS numbered trees in constant time.

3.1 State Representation

We now describe the state representation and the correspondence between states and strings. The states of our Aho-Corasick automaton representation are defined in the following way:

Definition 1. *Let P be the set of all prefixes of the strings in S, and let $m = |P|$. We define the function state as a function from P into the interval $[0, m-1]$ where $state(p) = rank_P(p)$ is the rank of the string p in P according to the suffix-lexicographic order (we count the number of elements of P which are smaller than p in the suffix lexicographic order).*

The suffix-lexicographic order is defined in the same way as standard lexicographic order except that the characters of the strings are compared in right-to-left order instead of left-to-right order. That is the strings of P are first sorted according to their last character and then ties are broken according to their next-to-last character, etc.... In order to distinguish final states from the other states, we simply note that we have exactly d terminal states corresponding to the d elements of S. As stated in the definition, each of the m states is uniquely identified by a number in range $[0, m-1]$. Therefore in order to distinguish terminal from non-terminal states, we use a succinct indexable dictionary, in which we store the d numbers corresponding to the d terminal states. As those d numbers

all belong to the range $[0, m-1]$, the total space occupation of our dictionary is (at most) $d(\log(m/d) + 1.443 + o(1))$ bits. In the following, we denote this dictionary as the state dictionary.

3.2 Representation of Next Transitions

We now describe how *next* transitions are represented. First, we note that a transition goes always from a state corresponding to a prefix p where $p \in P$ to a state corresponding to a prefix pc for some character c such that $pc \in P$. Therefore in order to encode the transition labeled with character c and which goes from the state corresponding to the string p (if such transition exists), we need to encode two informations: whether there exists a state corresponding to the prefix pc and the number corresponding to that state if it exists. In other words, given $state(p)$ and a character c, we need to know whether there exists a state corresponding to pc in which case, we would wish to get the number $state(pc)$.

The transition from $state(p)$ to $state(pc)$ can be done in a very simple way using a succinct indexable dictionary (lemma 4) which we call the transition dictionary. For that, we notice that $state(p) \in [0, m-1]$. For each non empty string $p_i = p'_i c_i$ where $p_i \in P$, we store in the transition dictionary, the pair $pair(p_i) = (c_i, state(p'_i))$ as the concatenation of the bit representation of c_i followed by the bit representation of $state(p'_i)$. That is we store a total of $m-1$ pairs which correspond to the $m-1$ non empty strings in P. Notice that the pairs are from a universe of size σm. Notice also that the pairs are first ordered according to the characters c_i and then by $state(p'_i)$ (in the C language notation a pair is an integer computed as $pair(p_i) = (c_i << \log m) + state(p'_i)$. Now the following facts are easy to observe:

1. Space occupation of the transition dictionary is $m(\log((\sigma \cdot m)/m) + 1.443 + o(1)) = m(\log \sigma + 1.443 + o(1))$.
2. The rank of the pairs stored in the succinct dictionary reflects the rank of the elements of P in suffix-lexicographic order. This is easy to see as we are sorting pairs corresponding to non empty strings, first by their last characters before sorting them by the rank of their prefix excluding their last character. Therefore we have $rank(pair(p_i)) = state(p_i) + 1$, where $rank$ function is applied on the transition dictionary.
3. A pair $(c_i, state(p'_i))$ exists in the transition dictionary if and only if we have a transition from the state corresponding to p'_i to the state corresponding to $p'_i c_i$ labeled with the c_i.

From the last two observations we can see that a transition from a state $state(p)$ for a character c can be executed in the following way: first compute the pair $(c, state(p))$. Then query the transition dictionary using the function $rank((c, state(p)))$. If that function returns -1, we can deduce that there is no transition from $state(p)$ labeled with character c. Otherwise we will have $state(pc) = rank((c, state(p))) + 1$. In conclusion we have the following lemma:

Lemma 5. *The next transitions of an Aho-corasick automaton whose states are defined according to definition 1 can be represented in (at most)* $m(\log \sigma + 1.443 + o(1))$ *bits of space such that the existence and destination state of a transition can be computed in constant time.*

3.3 Representation of Failure Transitions

We now describe how failure transitions are encoded. Recall that a failure transition connects a state representing a prefix p to the state representing a prefix q where q is the longest suffix of p such that $q \in P$ and $q \neq p$. The set of failure transitions can be represented with a tree called the *failure tree*. Each node in the failure tree represents an element of P and each element of P has a corresponding node in the tree. The failure tree is simply defined in the following way:

- The node representing a string p is a descendant of a node representing the string q if and only if $q \neq p$ and q is suffix of p.
- The children of any node are ordered according to the suffix-lexicographic order of the strings they represent.

Now an important observation on the tree we have just described is that a depth first traversal of the tree will enumerate all the elements of P in suffix-lexicographic order. That is the preorder of the nodes in the tree corresponds to the suffix lexicographic order of the strings of P. It is clear from the above description that finding the failure transition that connects a state $state(p)$ to a state $state(q)$ (where q is the longest element in P such that q is a suffix of p and $q \neq p$) corresponds to finding the parent in the failure tree of the node representing the element q. Using a succinct encoding (lemma 2), the tree can be represented using space $2m + o(m)$ bits such that the parent primitive is supported in constant time. That is the node of the tree corresponding to a state p will have preorder $state(p)$, and the preorder of the parent of that node is $state(q)$. A failure transition is computed in constant time by $state(q) = parent(state(p))$.

Lemma 6. *The failure transitions of the Aho-corasick automaton whose states are defined according to definition 1 can be represented in* $m(2 + o(1))$ *bits of space such that a failure transition can be computed in constant time.*

3.4 Representation of Report Transitions

The encoding of the *report* transitions is similar to that of failure transitions. The only difference with the failure tree is that except for the root, every internal node is required to represent an element of S. We remark that the report transitions form a forest of trees, which can be transformed into a tree by connecting all the roots of the forest (nodes which do not have a report transition) as children of state 0 (which hence becomes the root of the tree). In other words a report tree is the unique tree built on the elements of P which satisfies :

- All the nodes are descendants of the root (representing state 0) which represents the empty string.
- The node representing a string p is a descendant of a node representing a non empty string s if and only if $s \in S$, $s \neq p$ and s is a suffix of p.
- All children of a given node are ordered according to the suffix-lexicographic order of the strings they represent.

We could encode the report tree in the same way as the failure tree (using lemma 2) to occupy $m(2+o(1))$ bits of space. However we can obtain better space usage if we encode the report tree using the compressed tree representation (lemma 3). More specifically, the report tree contains at most d internal nodes as only strings of S can represent internal nodes. This means that the tree contains at least $m - d$ leaves. The entropy of the degree distribution of the report tree is $d(\log(m/d) + O(1))$ bits and the encoding of lemma 3 will use that much space (this can easily be seen by analogy to suffix tree representation in [12] which uses $d(\log((d+t)/d) + O(1))$ bits of space for a suffix tree with d internal nodes and t leaves). Report transitions are supported similarly to failure transitions in constant time using the parent primitive which is also supported in constant time by the compressed tree representation (lemma 3).

Lemma 7. *The report transitions of the Aho-corasick automaton whose states are defined according to definition 1 can be represented in* $d(\log(m/d) + O(1))$ *bits of space such that a report transition can be computed in constant time.*

3.5 Occurence Representation

Our Aho-corasick automaton will match strings from S which are suffixes of prefixes of the text T. This means that the Aho-corasick automaton will output the end positions of occurrences. However the user might need to also have the start position of occurrences. For that we have chosen to report occurrences as triplets $(occ\_start\_pos, occ\_end\_pos, string\_id)$, where $string \in S$ and occ_start_pos (occ_end_pos) is the start (end) position of the occurrence in the text. For that we need to know the length of the matched strings. But this information is not available as we do not store the original strings of the dictionary in any explicit form. We note that our algorithm outputs string identifiers as numbers from interval $[0, d-1]$ where the identifier of each string corresponds to the rank of the string in the suffix lexicographic order of all strings. Hence in order to store the string lengths, we succinctly store an array of d elements using the Elias-Fano encoding. In that array a cell i will store the length of the pattern number i. We call the resulting compressed array as the pattern length store. As the total length of the strings of the dictionary is n, the total space usage of the pattern length store will be $d(\lceil \log(n/d) \rceil + 2)$.

If the user has to associate specific action to be applied when a given string is matched, then he may use a table *action* of size d, where a cell number i stores the value representing the action associated with the pattern number i. The table could be sorted during the building of the state dictionary.

3.6 Putting Things Together

Summarizing the space usage of the data structures which are used for representation of the Aho-Corasick automaton:

1. The state dictionary which indicates the final states occupies at most $d(\log(m/d) + 1.443 + o(1)) \leq d(\log(n/d) + 1.443 + o(1))$ bits of space.
2. The *next* transitions representation occupies $B(m, m\sigma) + o(m) \leq m(\log(\sigma) + 1.443 + o(1))$ bits of space.
3. The $failure$ transitions representation occupies $m(2 + o(1))$ bits of space.
4. The *report* transitions representation occupies $d(\log(m/d) + O(1)) \leq d(\log(n/d) + O(1))$ bits of space.
5. The pattern length store occupies $d(\lceil \log(n/d) \rceil + O(1))$ bits of space.

The following lemma summarizes the space usage of our representation:

Lemma 8. *The Aho-corasick automaton can be represented in* $m(\log \sigma + 3.443 + o(1)) + d(3\log(n/d) + O(1))$ *bits of space.*

Implicit representation of the dictionary strings. We note that the state dictionary and the transition dictionary can be used in combination as an implicit representation of the elements of S.

Lemma 9. *For any integer* $i \in [0, d-1]$*, we can retrieve the string* $x \in S$ *of rank* i *(in suffix-lexicographic order) in time* $O(|x|)$ *by using the transition dictionary and state dictionary.*

The proof of the lemma is left to the full version.

3.7 Queries

Our query procedure essentially simulates the Aho-Corasick automaton operations, taking a constant time for each simulated operation. In particular performing each of the three kinds of transitions takes constant time. Thus our query time is within a constant factor of the query time of the original Aho-Corasick.

Lemma 10. *The query time of the succinct Aho-Corasick automaton on a text* T *is* $O(|T| + occ)$*, where occ is the number of reported occurrences.*

3.8 Construction

We now describe the construction algorithm which takes $O(n)$ time . The algorithm is very similar to the one described in [4]. We first write each string s_i of S in reverse order and append a special character # at the end of each string giving a set R. The character # is considered as smaller than all characters of original alphabet σ. Then, we build a (generalized) suffix-tree on the set R. This can be done in time $O(n)$ using the algorithm in [7] for example. Each leaf in the suffix tree will store a list of suffixes where a suffix s of a string $x \in R$ is represented by the pair $(string\_pointer, suf\_pos)$, where $string\_pointer$ is a pointer to x and suf_pos is the starting position of s in x. Then we can build the following elements:

1. The transition dictionary can be directly built as the suffix tree will give us the (suffix-lexicographic) order of all elements of P by a DFS traversal (top-down lexicographic traversal).
2. The failure tree is built by a simple DFS traversal of the suffix tree.
3. The report tree is built by doing a DFS traversal of the failure tree.
4. The state dictionary can be built by a traversal of the report tree.
5. The pattern length store can be built by a simple traversal of the set S.

Details of the construction are left to the full version.

Lemma 11. *The succinct Aho-corasick automaton representation can be constructed in time $O(n)$.*

The results about succinct Aho-Corasick representation are summarized by the following theorem:

Theorem 1. *The Aho-corasick automaton for a dictionary of d strings of total length n characters over an alphabet of size σ can be represented in $m(\log \sigma + 3.443 + o(1)) + d(3\log(n/d) + O(1))$ bits where $m \leq n+1$ is the number of states in the automaton. A dictionary matching query on a text T using the Aho-corasick representation can be answered in $O(|T| + occ)$ time, where occ is the number of reported strings. The representation can be constructed in $O(n)$ randomized expected time.*

4 Compressed Representation

The space occupancy of theorem 1 can be further reduced to $m(H_0 + 3.443 + o(1)) + d(3\log(n/d) + O(1))$, where H_0 is the entropy of the characters appearing as labels in the *next* transitions of the Aho-Corasick automaton:

Theorem 2. *The Aho-corasick automaton for a set S of d strings of total length n characters over an alphabet of size σ can be represented in $m(H_0 + 3.443 + o(1)) + d(3\log(n/d) + O(1))$ bits where $m \leq n + 1$ is the number of states in the automaton and H_0 is the entropy of the characters appearing in the trie representation of the set S. The theorem holds provided that $\sigma < m^{\varepsilon}$ for any constant $0 < \varepsilon < 1$. A dictionary matching query for a text T can be answered in $O(|T| + occ)$ time.*

Proof. Compared to theorem 1 we only modify the representation of the *next* transition which dominates the total space usage. That is, we reduce the space used to represent the *next* transitions from $m(\log \sigma + 1.443 + o(1))$ to $m(H_0 + 1.443 + o(1))$ and thus reduce the total space usage to $m(H_0 + 3.443 + o(1)) + d(3\log(n/d) + O(1))$ bits of space. We will use σ indexable dictionaries instead of a single one to represent the *next* transitions. Each dictionary corresponds to one of the characters of the alphabet. That is a pair $(c, state)$ will be stored in the dictionary corresponding to character c (we note that dictionary by $I[c]$). Additionally we store a table $T[0..\sigma - 1]$. For each character c we set $T[c]$ to the rank of character c (in suffix-lexicographic order) relatively to the set P (that

is the number of strings in the set P which are smaller than the string $''c''$ in the suffix lexicographic order). Let Y be the set of pairs to be stored in the transition dictionary. The indexable dictionary $I[c]$ will store all values $state_i$ such that $(c, state_i) \in Y$. Thus the number of elements stored in $I[c]$ is equal to the number of *next* transitions labeled with character c.

Now the target state for a transition pair $(c, state)$ is obtained by $T[c] + rank_{I[c]}(state)$, where $rank_{I[c]}(state)$ is the rank operation applied on the dictionary $I[c]$ for the value $state$. Let's now analyze the total space used by the table T and by the indexable dictionaries. The space usage of table T is $\sigma \log m \leq m^{\varepsilon} \log m = o(m)$. An indexable dictionary $I[c]$ will use at most $t_c(\log(m/t_c) + 1.443 + o(1))$ bits , where t_c is the number of transitions labeled with character c. Thus the total space used by all indexable dictionaries is $\sum_{0 \leq c < \sigma} t_c(\log(m/t_c)+1.443+o(1)) = m(H_0+1.443+o(1))$ and the total summed space used by the table T and the indexable dictionaries is $m(H_0+1.443+o(1))$.■

5 Concluding Remarks

Our work gives rise to two open problems: the first one is whether the term $3.443m$ in the space usage of our method which is particularly significant for small alphabets (DNA alphabet for example) can be removed without incurring any slowdown. The second one is whether the query time can be improved to $O(|T| \log \sigma / w + occ)$ (which is the best query time one could hope for).

Acknowledgements

The author is grateful to Mathieu Raffinot for proofreading the paper and for useful comments and suggestions. The author wishes to thank Kunihiko Sadakane and Rajeev Raman for confirming that the construction time of their respective data structures in [12] and [14] is linear.

References

[1] Aho, A.V., Corasick, M.J.: Efficient string matching: An aid to bibliographic search. Commun. ACM 18(6), 333–340 (1975)
[2] Chan, H.-L., Hon, W.-K., Lam, T.W., Sadakane, K.: Dynamic dictionary matching and compressed suffix trees. In: SODA, pp. 13–22 (2005)
[3] Clark, D.R., Munro, J.I.: Efficient suffix trees on secondary storage (extended abstract). In: SODA, pp. 383–391 (1996)
[4] Dori, S., Landau, G.M.: Construction of aho corasick automaton in linear time for integer alphabets. In: Apostolico, A., Crochemore, M., Park, K. (eds.) CPM 2005. LNCS, vol. 3537, pp. 168–177. Springer, Heidelberg (2005)
[5] Elias, P.: Efficient storage and retrieval by content and address of static files. J. ACM 21(2), 246–260 (1974)
[6] Fano, R.M.: On the number of bits required to implement an associative memory, Memorandum 61, Computer Structures Group, Project MAC. MIT, Cambridge (1971)

[7] Farach, M.: Optimal suffix tree construction with large alphabets. In: FOCS, pp. 137–143 (1997)
[8] Ferragina, P., Manzini, G.: Opportunistic data structures with applications. In: FOCS, pp. 390–398 (2000)
[9] Grossi, R., Vitter, J.S.: Compressed suffix arrays and suffix trees with applications to text indexing and string matching (extended abstract). In: STOC, pp. 397–406 (2000)
[10] Hon, W.-K., Lam, T.W., Shah, R., Tam, S.-L., Vitter, J.S.: Compressed index for dictionary matching. In: DCC, pp. 23–32 (2008)
[11] Hon, W.-K., Lam, T.W., Shah, R., Tam, S.-L., Vitter, J.S.: Succinct index for dynamic dictionary matching. In: Dong, Y., Du, D.-Z., Ibarra, O. (eds.) ISAAC 2009. LNCS, vol. 5878. Springer, Heidelberg (2009)
[12] Jansson, J., Sadakane, K., Sung, W.-K.: Ultra-succinct representation of ordered trees. In: SODA, pp. 575–584 (2007)
[13] Munro, J.I., Raman, V.: Succinct representation of balanced parentheses and static trees. SIAM J. Comput. 31(3), 762–776 (2001)
[14] Raman, R., Raman, V., Rao, S.S.: Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In: SODA, pp. 233–242 (2002)
[15] Sadakane, K.: Compressed text databases with efficient query algorithms based on the compressed suffix array. In: Lee, D.T., Teng, S.-H. (eds.) ISAAC 2000. LNCS, vol. 1969, pp. 410–421. Springer, Heidelberg (2000)
[16] Tam, A., Wu, E., Lam, T.W., Yiu, S.-M.: Succinct text indexing with wildcards. In: SPIRE, pp. 39–50 (2009)

A A Full Example

Let's now take as example the set $S = \{''ABC'','' B'','' BC'','' CA''\}$. The example is illustrated in figures 1 and 2. The set P of prefixes of S sorted in suffix-lexicographic order gives the sequence: $''\ ''$, $''A''$, $''CA''$, $''B''$, $''AB''$, $''C''$, $''BC''$, $''ABC''$. The first prefix

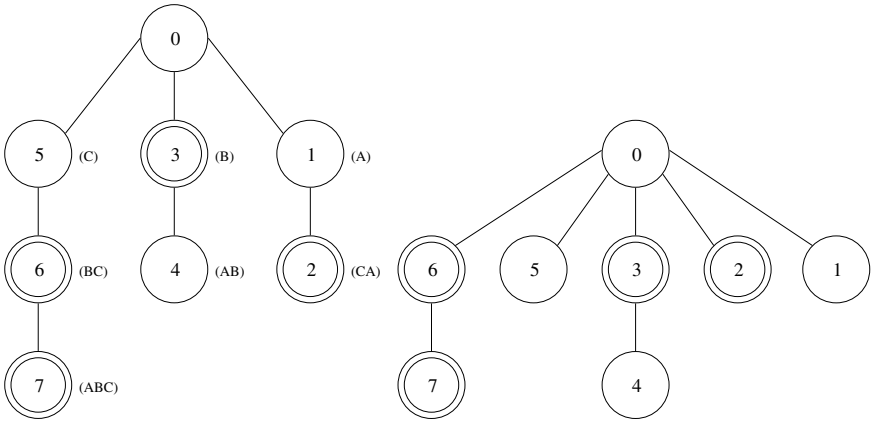


Fig. 2. Failure and report trees for the set {"ABC","B","BC","CA"}

(the empty string) in the sequence corresponds to state 0 and the last one corresponds to state 7. For this example, we store the following elements:

- The transition dictionary stores the following pairs: $(A, 0), (A, 5), (B, 0), (B, 1), (C, 0), (C, 3), (C, 4)$.
- The state dictionary stores the states $2, 3, 6, 7$ which correspond to the final states of the automaton (states corresponding to the strings of S).
- The report and failure trees are depicted in figure 2.
- The pattern length store, stores the sequence $2, 1, 2, 3$ which correspond to the lengths of the strings of S sorted in suffix lexicographic order ($''CA'', '' B'', '' BC'', '' ABC''$).

Pseudo-realtime Pattern Matching: Closing the Gap

Raphaël Clifford and Benjamin Sach

Department of Computer Science, University of Bristol, UK
{clifford,sach}@cs.bris.ac.uk

Abstract. We consider the k-difference and k-mismatch problems in the pseudo-realtime model where the text arrives online and the time complexity measure is per arriving character and unamortised. The well-known k-difference/k-mismatch problems are those of finding all alignments of a pattern of length m with a text of length n where the edit/Hamming distance is at most k. Offline, the literature gives efficient solutions in $O(nk)$ and $O(n\sqrt{k \log k})$ time, respectively. More recently, a pseudo-realtime solution was given for the former in $O(k \log m)$ time and the latter in $O(\sqrt{k \log k} \log m)$ time per arriving text character. Our work improves these complexities to $O(k)$ time for the k-difference problem and $O(\sqrt{k} \log k + \log m)$ for the k-mismatch problem. In the process of developing the main results, we also give a simple solution with optimal time complexity for performing longest common extension queries in the same pseudo-realtime setting which may be of independent interest.

1 Introduction

We revisit the problem of pattern matching in streaming data. Many well known and successful techniques have been developed since the 1970s for pattern matching under a variety of norms. However, almost without exception, it has been assumed that the entirety of the text is available to the algorithm during the length of its operation. In two recent papers [3,4] the pseudo-realtime (PsR) model was introduced. Here it is assumed that we are given a pattern P in advance and the text T to which it is to be matched arrives one character at a time. The overall task is to report matches between the pattern and text as soon as they occur and to bound the worst case time *per text input character*. The terminology extends the idea of realtime computing for situations where achieving constant time per character is not feasible. Crucially, the running time of the resulting algorithms is not amortised.

We focus on the well-known k-difference and k-mismatch problems in the pseudo-realtime model. Previously, deterministic algorithms which run in $O(k \log m)$ and $O(\sqrt{k \log k} \log m)$ time per text character respectively were shown [3,4]. Our contribution is to narrow the gap between the best known result in the offline setting and that which is achievable deterministically in pseudo-realtime. We first consider the k-difference problem and show how to solve it in $O(k)$ time per text character in the PsR setting, thus matching the

A. Amir and L. Parida (Eds.): CPM 2010, LNCS 6129, pp. 101–111, 2010.

best known $O(nk)$ time offline result of [8]. In order to achieve this complexity we require a realtime longest common extension (LCE) data structure which can be updated in constant time as new text characters arrive and which still permits constant time queries. Our solution achieves both aims and is, in the authors' view, remarkably straightforward and of independent interest. We then show how to develop an $O(\sqrt{k}\log k + \log m)$ time per character PsR solution for the k-mismatch problem, improving on the previous PsR result and enabling us to close in on the $O(n\sqrt{k\log k})$ time offline solution of [2]. Our solution requires two main technical innovations. First, a new filtering result given in Theorem 10 limits the number of k-mismatches that can occur in a length k contiguous region of the text. Second, we present a new trick that combines the PsR partitioning scheme presented in [3] with a fast algorithm for small values of k.

Our motivation goes further than the interesting theoretical question of whether offline and online approximate pattern matching algorithms need necessarily have different time complexities. The removal of the multiplicative $\log m$ factor, which is not there in the original offline versions, is also particularly significant as these bounded versions of the Hamming and edit distance problems are relevant to situations where the value of k is small.

Prior to the work on PsR approximate pattern matching, the problem of exact matching in constant time per new text character has been considered in [5] for example. Very recently, randomised algorithms both for the exact matching and k-mismatch problems in the streaming model, where the amount of working memory is smaller than the pattern size, have also been given [10].

2 Pseudo-realtime k-Differences

The k-difference problem is defined in relation to the edit distance. The edit distance between the pattern, P and some text substring, $T[i' \ldots i]$, is the minimum number of *insert*, *delete* and *mismatch* operations required to transform P into $T[i' \ldots i]$. These operations *insert* a single character, *delete* a single character or substitute one character for another (*mismatch*). We consider a formulation of the problem where the goal is to output the locations of all substrings of the text to which the pattern can be transformed in at most k operations. For each such location we also output the edit distance. In the pseudo-realtime model we will require that each such location, $T[i' \ldots i]$ is outputted as $T[i]$ arrives.

In a recent paper, the present authors demonstrated an algorithm for pseudo-realtime k-differences requiring $O(k \log m)$ time per character and $O(m)$ space [4]. As each $T[i]$ arrives, their algorithm outputs the minimal edit distance over all substrings of the form $T[i' \ldots i]$ with distance at most k. Prior to this work, Landau, Myers and Schmidt presented a different k-differences algorithm in a related incremental model [7]. They allow text characters to arrive at either end of the text and output all locations as required in $O(k)$ time per arriving character. However, their work is not immediately applicable in the PsR model as it requires the entire text to be preprocessed in advance for constant time LCE queries.

In order to adapt the previous $O(k)$ time k-differences algorithm to the situation where the text cannot be inspected in advance, we develop a realtime version of the LCE data structure in Section 2. The LCE is traditionally computed via lowest common ancestor (LCA) queries on a generalised suffix tree of the pattern and text. However we completely avoid the complications that would arise from performing dynamic LCA queries on an unamortised version of a suffix tree, giving a conceptually simple solution which may be of independent interest. We are then able to use this result to derive the final PsR k-differences result in Theorem 5.

A Simple Realtime Scheme for Constant Time LCE Queries

We give a dynamic text indexing structure that supports longest common extension (LCE) queries in constant time. The pattern is processed in advance in linear time. An important feature of our approach is that it supports the text arriving online one character at time and all operations, including updating the data structures used, can be performed in constant time without any amortisation assumptions.

Throughout we let i be the index of the most recently received text character, $T[i]$. For a given pattern and text, the longest common extension, $\mathrm{LCE}(i', j)$, is the length of the longest prefix of $P[j, \ldots m]$ and $T[i', i]$ which is common to both. In addition we will also consider pattern/pattern longest common extension queries, denoted by $\mathrm{LCE}_P(j, j')$. The value of $\mathrm{LCE}_P(j, j')$ is the length of the longest common prefix of $P[j \ldots m]$ and $P[j' \ldots m]$. Observe that we can preprocess P in linear time to answer LCE_P queries in constant time before any text has arrived using for example [9].

In the following, we will assume that all symbols in T occur at least once in P. If this is not the case, we add a new character, δ into the alphabet which is different from all pattern and text characters. We then modify P so that $P[m] = \delta$ and modify T as each character arrives online to replace any character not in P with a δ in constant time per character. The LCE returned may be at most one symbol too long or too short. This can be corrected with knowledge of the original $P[m]$ in constant time per query. The details are simple but are omitted for brevity.

Our algorithm splits the text into contiguous substrings which are encoded as triples, (i', j', ℓ), each representing a text substring $T[i', i' + \ell - 1]$ which equals a pattern substring $P[j', j' + \ell - 1]$. We refer to such triple as a p-region and a disjoint ordered sequence of triples which encodes the entire text as a p-representation. The length of a p-representation is the number of triples it contains. Trivially, a representation of at most length n always exists as each $T[i']$ occurs somewhere in P. To motivate the use of these representations, consider that to answer an LCE query between the pattern and text we could use a p-representation instead to identify a sequence of LCE_P queries to perform. We now show in Lemma 1 that if our p-representation is of minimal length, we need never perform more than three LCE_P queries. It will then remain only to show how to build such a minimal representation efficiently in realtime.

Lemma 1. *In a minimal length p-representation of T, for any query $LCE(i', j')$ at most three of the p-regions overlap $T[i', i' + LCE(i', j') - 1]$.*

Proof. For a contradiction, assume that at least four p-regions overlap $T[i', i + LCE(i', j') - 1]$. There must be two contiguous p-regions which correspond to text substrings which are themselves substrings of $T[i', i + LCE(i', j') - 1]$. Let (a, b, ℓ) and $(a+\ell, b+\ell, \ell')$ be two such p-regions. Observe that $T[a, a+\ell+\ell'-1]$ and $P[b, b+\ell+\ell'-1]$ are substrings of $T[i', i' + LCE(i', j') - 1]$ and $P[j', j' + LCE(i', j') - 1]$ respectively. By the definition of the LCE we have that $T[i', i' + LCE(i', j') - 1]$ matches $P[j', j' + LCE(i', j') - 1]$, therefore $T[a, a+\ell+\ell'-1]$ matches $P[b, b+\ell+\ell'-1]$ and $(a, b, \ell+\ell')$ is a p-region. Further we can obtain a shorter p-representation of T by replacing the two original p-regions with this new p-region, a contradiction. □

For any p-representation $\phi(T)$, we can obtain a representation of $T[1, i]$, denoted $\phi_i(T)$ by shortening the region containing $T[i]$ to end at text position i and removing all regions to its right. For our realtime algorithm we desire a p-representation, $\phi(T)$, with the property that $\phi_i(T)$ is minimal for all i. Observe that such a representation is suited to greedy construction. For motivation consider the pattern $P = bab$ and text, $T = aba$. There is a minimal p-representation given by $\phi(T) = (1, 2, 1), (2, 1, 2)$. However, the representation $\phi_2(T) = (1, 2, 1), (2, 1, 1)$ is not of minimal length. On the other hand, it is easily verified that there is another minimal p-representation, $\phi'(T) = (1, 2, 2), (3, 2, 1)$ for which $\phi'_1(T)$ and $\phi'_2(T)$ are both minimal. Lemma 2 shows that for any pattern and text such a representation exists.

Lemma 2. *For any pattern and text, there exists a minimal length p-representation, $\phi(T)$, such that for all i, $\phi_i(T)$ is a p-representation of $T[1, i]$ with minimal length.*

Proof. We begin by letting $r(T)$ be an arbitrary minimal p-representation of T. Consider the largest i such that $r_i(T)$ is not minimal. If no such i exists, then there is no work to be done. Otherwise we will modify $r(T)$ to make $r_{i'}(T)$ minimal for all $i \leq i' \leq n$. Consider the p-region which contains $T[i]$ in $r(T)$. If the region extends to the right of $T[i]$, split it in two so that there is a break immediately after $T[i]$. This split increases $|r(T)|$ by one. However, we can now replace all the regions to the left of this break with the regions in any minimal p-representation of $T[1, i]$. As $r_i(T)$ was not minimal pre-modification this step decreases the length of $|r(T)|$ by at least one. Therefore our modified $r(T)$ is no longer than the original and in fact must be of the same length. Further, observe that $r_i(T)$ is now minimal and $r_{i'}(T)$ is still minimal for all $i < i' \leq n$. We can repeat this process until an $r(T)$ is obtained for which $r_i(T)$ is minimal for all $1 \leq i \leq n$ as required. Let $\phi(T) = r(T)$. □

Our algorithm incrementally constructs a greedy p-representation of the text seen so far, $T[1, i]$, which we denote $g_i(T)$. We show below in Lemma 3 this is a minimal length p-representation of $T[1, i]$. As a preprocessing step, we construct

a suffix tree of the pattern in linear time. When $T[i]$ arrives we compute $g_i(T)$ from $g_{i-1}(T)$ as follows. Consider the rightmost p-region in $g_{i-1}(T)$ which has the form $(i-\ell, b, \ell)$ for some b, ℓ. We determine whether this region can be extended into a p-region $(i-\ell, b, \ell+1)$. To perform this check efficiently we maintain a pointer into the position in the suffix tree representing the rightmost p-region. Observe that the region can be extended iff it is possible to step down from the current position in the suffix tree using the character $T[i]$. If the p-region cannot be extended, we insert a new p-region, $(i, b', 1)$ where b' is some location such that $P[b'] = T[i]$ and update the suffix tree pointer.

Lemma 3. *For all i, the greedy p-representation, $g_i(T)$, which represents $T[1,i]$ is of minimal length.*

Proof. Let $\phi(T)$ be a minimal length p-representation with the property that $\phi_i(T)$ is minimal for all i. Such a $\phi(T)$ exists by Lemma 2. We prove that $|g_i(T)| = |\phi_i(T)|$ by induction on i. Observe that for the base case $i = 1$, we have that $|g_1(T)| = |\phi_1(T)| = 1$. Therefore by the inductive hypothesis, we assume that $|g_{i'}(T)| = |\phi_{i'}(T)|$ for all $1 \leq i' < i$. By the algorithm description, $|g_i(T)|$ either equals $|g_{i-1}(T)|$ or $|g_{i-1}(T)| + 1$. In the former, by the monotonicity of $\phi_i(T)$, we have that $|\phi_i(T)| = |g_i(T)|$. Therefore we assume that $|g_i(T)| = |g_{i-1}(T)| + 1$. Let (a, b, ℓ) and (a', b', ℓ') be the triples corresponding to the rightmost p-regions in $g_{i-1}(T)$ and respectively $\phi_{i-1}(T)$. First suppose that $a < a'$ and observe that $\phi_a(T)$ contains less than $|\phi_{i-1}(T)|$ regions. However, $g_a(T)$ contains $|g_{i-1}(T)| = |\phi_{i-1}(T)|$ regions, which is a contradiction as $|g_a(T)| = |\phi_a(T)|$ by the inductive hypothesis ($a < a' \leq i$). Therefore we have that $a \geq a'$. Further, by the construction of $g_i(T)$, we have that $T[a, i]$ does not match any substring of P and therefore $T[a', i]$ does not match any substring of P. Therefore $|\phi_i(T)| = |\phi_{i-1}(T)| + 1$ as required. □

Theorem 4. *There exists a dynamic data structure which can answer LCE queries between the pattern and $T[1,i]$ in $O(1)$ time. When a new text character, $T[i+1]$ arrives, the structure can be updated in $O(1)$ time. The structure requires $O(i)$ space and $O(m)$ pattern preprocessing time.*

Proof. The algorithm described maintains a p-representation of the text seen so far which is minimal by Lemma 3. When a text character arrives, the checks required can be performed in constant time[1]. By Lemma 1 we require at most three LCE_P queries to perform an LCE query. These LCE_P queries can be identified and then performed in constant time. □

The k-Differences Algorithm

Careful inspection of the k-difference algorithm of Landau, Myers and Schmidt [7] shows that by using the pseudo-realtime LCE processing that we

[1] Strictly speaking traversing a suffix tree also incurs an $O(\log |\Sigma|)$ penalty at the nodes. However we omit this from our results to be consistent with the large body of previous work.

have presented their algorithm can be translated fully to the pseudo-realtime model. The details are left for the full version of the paper but the result is summarised in Theorem 5.

Theorem 5. *The k-differences problem can be solved in the PsR model in $O(k)$ time per character and $O(m)$ space.*

3 Pseudo-realtime k-Mismatches

The k-mismatch problem is defined in relation to the hamming distance which is the number mismatches (single character differences) between two strings. The goal is to find all alignments where the hamming distance is at most k. For each such location we also output the hamming distance. In the pseudo-realtime model we will require that each such location, $T[i-m+1\ldots i]$ is outputted as $T[i]$ arrives.

We now present our pseudo-realtime k-mismatch algorithm which follows the overall structure of the offline solution of Amir et al. [2]. Their structure is in turn based on a general frequent/infrequent splitting trick which is originally due to Abrahamson and Kosaraju [1,6]. Our algorithm is parameterised by two variables f and b which will feature in the time complexity. These will then be set to minimise the time complexity per character in terms of k and m. When minimising, we will ensure that $bf \geq 3k$ which will be required below. We term a character to be *frequent* if it occurs at least $6f$ times in the pattern. We now separate the algorithm into two cases determined by the number of frequent characters in the pattern:

Case 1: There Are Fewer Than $6b$ Frequent Characters in the Pattern

For this case we are able to modify the solution of Amir et al. [2] to make the solution PsR. Their original method counts matches rather than mismatches and considers frequent and infrequent characters separately. They observe that each text position which matches an infrequent character matches at fewer than, in our case $6f$, positions in the pattern. Therefore all matches involving an infrequent character can be found in $O(nf)$ time by directly counting the number of matches at each alignment. This process can be made PsR straightforwardly as the work is performed independently for each new text character that arrives. However we may require $O(\log|\Sigma|)$ time to classify the arriving text character. This process is therefore upper-bounded by $O(f+\log m)$ time per text character if the text is arriving online.

To handle a single frequent character, Amir et al. transform the pattern and text into binary representations. These representations have a 1 at locations where the frequent character occurs and 0 otherwise. They observe that the number of matches at each alignment can then be found using cross-correlations in $O(n\log m)$ time. However, if the text arrives online, we cannot use the standard

FFT-based cross-correlation method. Instead we replace this with the pseudo-realtime cross-correlation method of [3,4]. This method now requires $O(\log^2 m)$ time per arriving text character. As there are fewer than $6b$ frequent characters, the original process requires a total of $O(nb \log m)$ time and our modified pseudo-realtime process requires $O(b \log^2 m)$ time per character. The result is summarised in Lemma 6.

Lemma 6. *Assume that the pattern contains fewer than* $6b$ *frequent characters, each of which occurs at least* $6f$ *times. The* k*-mismatch problem can solved in pseudo-realtime in* $O(f + b \log^2 m)$ *time per character and* $O(m)$ *space.*

Case 2: There Are at Least $6b$ Frequent Characters in the Pattern

As in Amir et al.'s offline algorithm we perform two main stages. First, we *filter* the locations where a potential match could occur and second we *verify* the filtered locations which indeed contain at most k mismatches. It is essential for the translation to the pseudo-realtime setting that unlike the original, our filtering results restrict not just the number but also the distribution of potentially matching locations. Intuitively this is because otherwise we may encounter a long stretch of potentially matching locations each requiring $\Theta(k)$ time to verify. To perform the verification in pseudo-realtime, we will require the use of our realtime LCE results which were presented in Section 2 and also careful scheduling to ensure that the time complexity of the resulting algorithm is unamortised.

We begin by trimming the pattern to remove its rightmost $3k$ characters, as there are at least $6b$ frequent characters in P we have that $m > 3k$. The mismatches between these $3k$ positions and the text will be handled separately. For motivation we consider the advantage of this trimming. Consider some pattern/text alignment where the rightmost character of the trimmed pattern is aligned with the most recently arrived text character, $T[i]$. At this alignment, the rightmost position of the full pattern is aligned with text character $T[i+3k]$. Therefore there are $3k$ text character arrivals between the first point at which we have the seen the text aligned with the trimmed pattern and the point at which we must output the result. We will use this delay to give us sufficient scheduling flexibility to output in pseudo-realtime.

The algorithm begins by preprocessing the pattern to identify a set of $2b$ pattern substrings which can be used to filter the text locations where a k-mismatch could occur. Formally, we say that $R[j \ldots j']$ is an f*-balanced substring* of some string R for symbol $s \in \Sigma$ if the substring $R[j \ldots j' - k]$ contains exactly $2f$ occurrences of s, and $R[j' - k + 1 \ldots j']$ contains at most f occurrences of s.

To find enough f-balanced substrings in the trimmed pattern to perform our filtering we need to show that the trimmed pattern still contains many characters which are almost frequent (but not too frequent). As $bf \geq 3k$ there cannot be more than b symbols which occur at least f times in the rightmost $3k$ positions in the pattern. Further observe that as the trimmed pattern is of length $(m-3k)$ it contains at most b symbols which occur at least $(m-3k)/b$ times. Therefore the trimmed pattern contains at least $4b$ symbols which occur at least $5f$ and less

than $(m-3k)/b$ times and hence by Lemma 7 we have that either the trimmed pattern or its reverse contains an f-balanced substring for each of at least $2b$ distinct symbols in Σ. We concentrate our explanation on the former case. In the latter case, simple modifications are needed and are left for the full version. It is straightforward to find f-balanced substrings in the trimmed pattern for $2b$ distinct symbols in $O(\text{sort}(m))$ time. Here $\text{sort}(m)$ is the time taken to sort the pattern by character which is upper bounded by $O(m \log |\Sigma|)$. We denote the f-balanced substring found for some symbol s by $W_p(s)$. Further for each $W_p(s) := P[j \dots j']$ we construct a linked list, $L_p(s)$ of the $2f$ occurrences of s in $P[j \dots j' - k]$ in $O(m)$ total time and space.

Lemma 7. *Consider an arbitrary string R. Let $s \in \Sigma$ be a symbol which occurs at least $5f$ and less than $|R|/b$ times in R with $bf \geq 3k$. There is an f-balanced substring in either R or the reverse of R.*

Proof. Assume that R has no k-length substring which contains at most f occurrences of s. Therefore we have that all disjoint k length substrings of R contain more than f occurrences of s, so R contains more than $f|R|/k \geq |R|/b$ occurrences of s, a contradiction. Consider the first k-length R substring to contain at most f occurrences of s. As s occurs at least $5f$ times there are at least $2f$ occurrences of s either to the left or to the right of this substring. The result follows from the definition of an f-balanced substring above. □

As the text arrives we will monitor the text substrings which align with the f-balanced substrings found during preprocessing. For each of the $2b$ f-balanced substrings we define the corresponding text window, $W_t(s, i)$, to be the substring of T of length $|W_p(s)|$ which is aligned with $W_p(s)$ when the rightmost position in the trimmed pattern is aligned with $T[i]$ (the most recently arrived character). For each text window, we maintain a list $L_t(s, i)$ of up to $4f + 1$ of the latest (rightmost) occurrences of s in $W_t(s, i)$. As there are $2b$ such windows these lists can be updated when a new character arrives in $O(b)$ total time per character (and use $O(m)$ total space). Lemma 8 gives the first filtering result for these text windows.

Lemma 8. *If at least b of the text windows, $W_t(s, i)$, contain more than $4f$ occurrences of s then $T[i + 3k - m + 1 \dots i]$ has more than k mismatches with the trimmed pattern.*

Proof. We have for a single such s that $W_t(s, i)$ contains more than $4f$ occurrences of s while $W_p(s)$ contains at most $3f$ so we have found more than f mismatches where the text character is s. Across all such symbols we have found more than $bf \geq 3k$ mismatches which gives the desired result. □

As there are $2b$ text windows, by Lemma, 8, we only need to consider locations where at least b of the text windows contain at most $4f$ occurrences of their corresponding symbol. Using the lists described above we can determine whether the current $T[i]$ has this property in constant time. Having found a suitable $T[i]$, we will show how to efficiently find all alignments in the the next k positions

which have at most k mismatches with the trimmed pattern. Recall that the alignment of the rightmost position in the trimmed pattern with $T[i]$ corresponds to the alignment of the full pattern with $T[i+3k]$. Hence we still have $3k$ text arrivals remaining before we need to output our first result. After finding all these positions, the algorithm begins again with the first suitable position after $T[i+k-1]$. Note that computations discussed may overlap but it is easily verified that this only increases the time complexity by a small multiplicative constant.

Let i' be a suitable text position as identified above, which is fixed in the remainder. Consider the at least b f-balanced substrings which correspond to text windows, $W_t(s,i)$ containing at most $4f$ occurrences of s. We have that these f-balanced substrings contain a total of at least $b \cdot 2f \geq 6k$ distinct pattern positions. Pick $2k$ of these pattern positions. For each alignment in the next k, count the matches involving one of those picked pattern positions. Each pattern position is in some list, $L_p(s)$ and by the construction of the f-balanced substrings, during the next k alignments it only matches with text positions in $L_t(s,i')$. As each $|L_t(s,i')| \leq 4f$, we perform at total of at most $8fk$ comparisons in this step. We distribute these comparisons over the next k arriving characters so that $O(f)$ comparisons are made per alignment. Discard any alignment which has less than k matches. By Lemma 9 we have that at most $8f$ alignments remain and that all discarded alignments had more than k mismatches. Note that there are still $2k$ text arrivals before we must make our first output.

Lemma 9. *If at least b of the text windows, $W_t(s,i)$, contain at most $4f$ occurrences of s then there are at most $8f$ alignments where there are at most k-mismatches with the trimmed pattern in the next k text arrivals. The algorithm above correctly identifies these positions.*

Proof. Consider the $2k$ positions picked in the algorithm description. Recall that each picked pattern position is in some L_s and matches at most $|L_t(s,i')| \leq 4f$ times in the next k alignments. Summing over all $2k$ pattern positions this gives a total of at most $8kf$ matches. However, any alignment with less than k matches must have more than k mismatches as there are $2k$ positions. Therefore there are at most $8f$ positions where a k-mismatch could occur. □

Theorem 10 summarises the main filtering result that we have shown which can be seen as a tightening of the central filtering result of Amir et al. [2]. It follows directly from Lemma 8, Lemma 9 and the algorithm description.

Theorem 10. *Assume there are at least $5b$ symbols, each of which occurs at least $5f$ times in the pattern where $bf \geq 3k$. In k consecutive alignments of the pattern with the text, there are at most $8f$ positions where a k-mismatch occurs.*

Having found the at most $8f$ potential matching alignments in the next k alignments, it only remains to verify them. To find mismatches with the trimmed pattern we use the online LCE query algorithm presented in section 2. We can process the text to answer text/pattern LCE queries in constant time per arriving character. Further we can perform the LCE queries in constant time. We can find up to

$k+1$ mismatches at a potential location using $k+1$ constant time LCE queries. We distribute the at most $8f(k+1)$ queries evenly over the next k text arrivals requiring $O(f)$ time per character. Again note that there are still k text arrivals before we must make the first output and $2k$ before the final output (for these k alignments).

We now consider mismatches in the final $3k$ positions in the pattern. We only need to consider the at most $8f$ known potential k-mismatching alignments identified by the filtering as if the trimmed pattern does not k-mismatch then the full pattern certainly does not. Therefore we must make $\Theta(fk)$ comparisons to determine the remaining mismatches. We distribute these comparisons evenly over the next $2k$ text arrivals. The comparisons are performed ordered left to right by corresponding alignment. Careful inspection shows that we will have each result by the time it is needed on the arrival of a new text character. The result is summarised in Lemma 11.

Lemma 11. *Assume that the pattern contains at least $6b$ frequent characters, each of which occurs at least $6b$ times and that $bf \geq 3k$. The k-mismatch problem can solved in pseudo-realtime in $O(f + b\log^2 m)$ time per character and $O(m)$ space.*

We are now able to give the first new result for the k-mismatch problem in pseudo-realtime setting. Theorem 12 combines the algorithms for the two cases detailed above.

Theorem 12. *The k-mismatch problem can be solved in pseudo-realtime in $O(\sqrt{k}\log m + \log^2 m)$ time per character and $O(m)$ space.*

Proof. By combining Lemma 6 and Lemma 11 we obtain a general algorithm for the k-mismatch algorithm in pseudo-realtime which is upper bounded by $O(f + b\log^2 m)$ time per character (and $O(m)$ space). To give the desired result, let $f = 2\lceil\sqrt{k}\log m\rceil$ and $b = 2\lceil\sqrt{k}/\log m\rceil$. Observe that we have that $bf \geq 3k$ as required. By substituting and simplifying we obtain the result as stated. □

An Improved k-Mismatch Algorithm

The result in Theorem 12 depends on both k and m. We now show how to reduce the complexity so that the dependence on $\log m$ is only additive rather than multiplicative. First consider the case that $k^5 \geq m/2$. In this case the algorithm presented above requires $O(\sqrt{k}\log k)$ time per character as desired. Therefore we only consider the case that $k^5 < m/2$:

Following the black box methodology of Clifford et al. [3], we divide the pattern into consecutive substrings of halving length $P_1, P_2 \ldots P_s$. However, we set s so that the final section is of length $k^5/2 \leq |P_s| < k^5$. Note that all sections except section s have $|P_i| > k^5$. Also note that $2 \leq s \leq \lceil\log m\rceil$. Using the techniques of the previous work we can compute matches of $P_1, P_2 \ldots P_{s-1}$ before they are needed. However this time we use the small k algorithm of Amir et al [2] (Cor 6.1) for each section. Their algorithm requires $O(n + nk^4\log k/m)$ time.

Therefore for subpattern P_i with $i < s$, the time per arriving text character is upper bounded by $O\left(1 + k^4 \log k/|P_i|\right) \in O(1)$ as $|P_i| > k^5$. We achieve this complexity by distributing the work over arriving characters as described by the black box methodology. To compute mismatches with the final section, we use the result of Theorem 12. As $|P_s| < k^5$, this requires $O(\sqrt{k} \log k)$ time per character. The results can be summed in $O(\log m)$ time per character giving a total complexity as summarised by Theorem 13.

Theorem 13. *The k-mismatch problem can be solved in pseudo-realtime in* $O(\sqrt{k} \log k + \log m)$ *time per character and* $O(m)$ *space.*

References

1. Abrahamson, K.R.: Generalized string matching. SIAM J. Comput. 16(6), 1039–1051 (1987)
2. Amir, A., Lewenstein, M., Porat, E.: Faster algorithms for string matching with k mismatches. In: SODA 2000, pp. 794–803 (2000)
3. Clifford, R., Efremenko, K., Porat, B., Porat, E.: A black box for online approximate pattern matching. In: Ferragina, P., Landau, G.M. (eds.) CPM 2008. LNCS, vol. 5029, pp. 143–151. Springer, Heidelberg (2008)
4. Clifford, R., Sach, B.: Online approximate matching with non-local distances. In: Kucherov, G., Ukkonen, E. (eds.) CPM 2009. LNCS, vol. 5577, pp. 142–153. Springer, Heidelberg (2009)
5. Galil, Z.: String matching in real time. Journal of the ACM 28(1), 134–149 (1981)
6. Kosaraju, S.R.: Efficient string matching (1987) (manuscript)
7. Landau, G.M., Myers, E.W., Schmidt, J.P.: Incremental string comparison. SIAM J. Comput. 27(2), 557–582 (1998)
8. Landau, G.M., Vishkin, U.: Fast string matching with k differences. J. Comput. Syst. Sci. 37(1), 63–78 (1988)
9. Manber, U., Myers, G.: Suffix arrays: a new method for on-line string searches. In: SODA 1990, pp. 319–327 (1990)
10. Porat, E., Porat, B.: Exact and approximate pattern matching in the streaming model. In: FOCS 2009, pp. 315–323 (2009)

Breakpoint Distance and PQ-Trees

Haitao Jiang[1,2], Cedric Chauve[3], and Binhai Zhu[1]

[1] Department of Computer Science, Montana State University, Bozeman, MT 59717, USA
bhz@cs.montana.edu
[2] School of Computer Science and Technology, Shandong University, China
htjiang@cs.montana.edu
[3] Department of Mathematics, Simon Fraser University, 8888 University Drive, Burnaby, BC V5A 1S6, Canada
cedric.chauve@sfu.ca

Abstract. The PQ-tree is a fundamental data structure that can encode large sets of permutations. It has recently been used in comparative genomics to model ancestral genomes with some uncertainty: given a phylogeny for some species, extant genomes are represented by permutations on the leaves of the tree, and each internal node in the phylogenetic tree represents an extinct ancestral genome, represented by a PQ-tree. An open problem related to this approach is then to quantify the evolution between genomes represented by PQ-trees. In this paper we present results for two problems of PQ-tree comparison motivated by this application. First, we show that the problem of comparing two PQ-trees by computing the minimum breakpoint distance among all pairs of permutations generated respectively by the two considered PQ-trees is NP-complete for unsigned permutations. Next, we consider a generalization of the classical Breakpoint Median problem, where an ancestral genome is represented by a PQ-tree and p permutations are given, with $p \geq 1$, and we want to compute a permutation generated by the PQ-tree that minimizes the sum of the breakpoint distances to the p permutations. We show that this problem is Fixed-Parameter Tractable with respect to the breakpoint distance value. This last result applies both on signed and unsigned permutations, and to uni-chromosomal and multi-chromosomal permutations.

1 Introduction

PQ-tree is a fundamental data structure in computer science. First invented by Booth and Lueker as a tool to verify whether a matrix has the consecutive ones property [4], it has numerous applications: for example, recognizing interval graphs, testing whether a graph is planar, and creating a contig map from DNA segments [4,1,14]. In short, a PQ-tree on the set $\Sigma = \{1, \ldots, n\}$ is a plane rooted tree with three kinds of nodes: P-nodes, Q-nodes and leaves, with n leaves labeled on Σ (no two leaves can have the same label). A fundamental feature of PQ-trees is that a given PQ-tree encode in linear space a possibly exponential number of permutations.

A. Amir and L. Parida (Eds.): CPM 2010, LNCS 6129, pp. 112–124, 2010.

Recently, PQ-trees have been used to represent extinct ancestral genomes from a set of extant genomes represented by permutations on the same set of markers (see [6] and references there). The PQ-tree representing an extinct ancestral genome generates possible marker orders that accounts for some uncertainty regarding the order of some markers along the ancestral chromosomes. Note that some other ways to account for uncertainty or contradictory information have been defined, such as partial orders [18], but not in the context of ancestral genomes.

Once the internal nodes of a phylogenetic tree are each labeled with a PQ-tree representing the corresponding extinct genome, a natural question is to use this information to infer quantitative properties on the evolution that generated the observed extant genomes. For branches linking two internal nodes in the tree, this amounts to quantify the similarity between these two PQ-trees. We consider here the breakpoint distance. Following previous works on comparing structures generating several permutations, we consider the Minimum-Breakpoint-Permutation from PQ-Trees (MBP-PQ): given two PQ-trees T_1 and T_2, find a permutation s_1 generated by T_1 and a permutation s_2 generated by T_2 such that the breakpoint distance between s_1 and s_2 is minimum. We show that, as for partial orders [10,3], this problem is NP-complete. Next, we consider the restricted problem where T_2 generates a single permutation, that we call the One-Sided MBP-PQ, and we show that this problem is Fixed-Parameter Tractable (FPT), with parameter being the optimal breakpoint distance. We show that the same result holds for the more general *median* problem that considers p permutations $\{s_1, \ldots, s_p\}$ and a PQ-tree T and asks for a permutation s generated by T that minimizes the sum of the p breakpoint distances between s and each permutation in $\{s_1, \ldots, s_p\}$, that we call the p-Minimum-Breakpoint-Median from PQ-Tree (p-MBM-PQ). This problem generalizes naturally the classical Breakpoint-Median Problem, by imposing constraints on the possible medians, at least for permutations that represent uni-chromosomal genomes. As far as we know, our FPT algorithm is only the second occurrence of an FPT result for hard median problems, after [11].

2 Preliminaries

Permutations, breakpoints and medians. Genomes with unique gene content are encoded using permutations on an alphabet of genome markers. Let Σ be such an alphabet of n markers. A *uni-chromosomal permutation* is a permutation on Σ. Given a permutation s, an *adjacency* a, b is composed of two markers that form a substring in s, either as ab or ba. A *linear* permutation of n markers contains then $n-1$ adjacencies. From now on, we omit the term linear and consider that by default every permutation is linear. The two extremities of a permutation are called *telomeres*. A *multi-chromosomal* permutation having k chromosomes is a set of k permutations on k disjoint subsets of Σ. It then contains $n-k$ adjacencies and $2k$ telomeres.

Given two permutations s_1 and s_2, over the same set of alphabet Σ, we say ab forms a *common adjacency* if ab or ba is a substring in both s_1 and s_2. Otherwise, if ab appears in s_1 and neither ab nor ba appears in s_2, then we say ab forms a *breakpoint*. A marker a is a common telomere to s_1 and s_2 if it is a telomere in both permutations. We denote by $a(s_1, s_2)$ (resp. $t(s_1, s_2)$) the number of common adjacencies (resp. telomeres) between s_1 and s_2. The breakpoint distance between s_1 and s_2 is defined, as in [17], by the following formula: $d_b(s_1, s_2) = n - a(s_1, s_2) - t(s_1, s_2)/2$. Note that when s_1 and s_2 are uni-chromosomal permutations, it is common to frame them by two new markers, that become telomeres, and the distance formula, that we will use in this case, is $d_b(s_1, s_2) = n - a(s_1, s_2)$, which is the number of breakpoints between s_1 and s_2. In both cases, the breakpoint distance can obviously be computed in linear time.

Given p permutations $\{s_1, \ldots, s_p\}$, the Breakpoint-Median Problem asks for a permutation s that minimizes $\sum_{i=1}^{p} d_b(s_i, s)$.

To handle signed markers in permutations, we use the same idea as in [12]: we double the number of markers and for marker i, we represent it with the two consecutive markers $(2i-1)$ $(2i)$, and for marker $-i$ we represent it with $(2i)$ $(2i-1)$. Common adjacencies and telomeres can then be described as common adjacencies for the corresponding unsigned permutations.

PQ-trees. Formally, a PQ-tree for unsigned permutations is a plane tree with internal nodes that can be either P-nodes or Q-nodes (P-nodes and Q-nodes have at least 2 children). (Note that when a P-node has 2 children, it is really a Q-node.) Reading the leaves of a PQ-tree in a post-order traversal gives a permutation called the *signature* of this PQ-trees. The operations of reordering the children of a P-node in an arbitrary way and reversing the children of a Q-node (and mirroring the corresponding subtrees) are called *allowed operations*. These operations define an equivalence relation between PQ-trees: two PQ-trees are equivalent if and only if we can transform one into the other by a sequence of allowed operations. The set of uni-chromosomal permutations *generated* by a given PQ-tree is the set of the signatures of all the PQ-trees of its equivalence class. See Figure 1 for an illustration of PQ-trees and generated uni-chromosomal permutations. When dealing with multi-chromosomal permutations, we assume the root of the considered PQ-tree T is a P-node. The set of multi-chromosomal

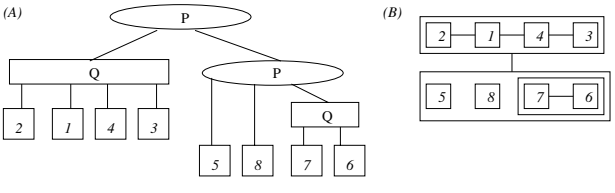


Fig. 1. (A) A PQ-tree T. $\langle 2, 1, 4, 3, 7, 6, 8, 5\rangle$ and $\langle 3, 4, 1, 2, 5, 6, 7, 8\rangle$ are permutations generated by this PQ-tree, but not $\langle 1, 2, 3, 4, 5, 6, 7, 8\rangle$ as 1 has to be adjacent to 4 because they are adjacent siblings in a Q-node. (B) A graph representation G of T.

permutations from a PQ-tree, is defined as follows: a multi-chromosomal permutation s with k chromosomes is generated by a PQ-tree T if and only if there exists a uni-chromosomal permutation s' generated by T such that, discarding $k-1$ adjacencies in s' formed of markers that belong to subtrees rooted at different children of the root of T results in s. We denote the number of permutations generated by a PQ-tree T by $P(T)$, assuming the context makes it clear if they are uni-chromosomal or multi-chromosomal.

PQ-trees for signed permutations have the additional constraint that, for every i, the leaves $2i$ and $2i-1$ are consecutive siblings of a Q-node.

Problem statements. We now formally state the problems we will investigate in this paper. Each has four different versions, depending on whether the considered permutations are uni-chromosomal or multi-chromosomal, and signed or unsigned.

Minimum Breakpoint Permutations from PQ-trees (MBP-PQ):
Input: PQ-trees T_1 and T_2 over the same set of n markers, integer K.
Question: Can T_1 and T_2 generate permutations s_1 and s_2 respectively such that $d_b(s_1, s_2) \leq K$?

The **One-Sided MBP-PQ** is the special case where T_2 generates a single permutation called s_2. It is a special case of a more general problem, that generalizes the classical Breakpoint Median Problem.

p-Minimum Breakpoint Median from PQ-tree (p-MBM-PQ):
Input: PQ-trees T and p permutations $s_1, \ldots, s_p$ over the same set of n markers, integer K.
Question: Can T generate a permutation s such that $\sum_{i=1}^{p} d_b(s, s_i) \leq K$?

FPT algorithms. An FPT (Fixed-Parameter Tractable) algorithm for an optimization problem Π with parameter value p is an algorithm which solves the problem in $O(f(p)n^c)$ time, where f is any function only on p, n is the input size and c is some fixed constant not related to p. For convenience we also say that Π is in FPT. More details on FPT algorithms can be found in [8].

Existing results. If T is a PQ-tree generating all possible permutations, the p-MBM-PQ Problem is equivalent to the classical Breakpoint-Median Problem described above, that is NP-hard, for signed or unsigned, uni-chromosomal or multi-chromosomal permutations [5,16,17]. In the uni-chromosomal case, even in the case where the median is constrained to have only adjacencies that appear in at least one of the genomes s_i, the problem is NP-hard [5]. This implies immediately that the p-MBM-PQ Problem is NP-hard, for $p \geq 3$, in all cases.

The MBP-PQ Problem, which we prove to be NP-complete in next section for unsigned permutations, can be solved by an FPT algorithm whose parameter is $t = P(T_1) \times P(T_2)$, as it is easy to list all permutations generated by T_1 and T_2 in polynomial time and examine each pair of permutations to compute the breakpoint distance. However $P(T)$ can be superexponential for a PQ-tree T

with a P-node of large degree, and it is at least exponential in the number of Q-nodes, as each Q-node can be reversed to generate a new signature.

The same argument applies to the p-MBM-PQ Problem, and, even in the case where T has only Q-nodes (say q Q-nodes), the time complexity of the algorithm is $O(2^q n)$. In datasets where ancestral genomes are well defined and $P(T)$ is small, this approach is the most efficient, especially as it allows to consider more precise distances than the breakpoint distance. However, we consider in Section 5 some real data where $P(T)$ is too large for this approach, which motivates our investigation of an FPT with respect to an alternative parameter. In Section 4, we describe an FPT algorithm parameterized by the value of the searched optimal solution, that is the breakpoint distance of the median permutation to the input permutations.

3 MBP-PQ Is NP-Complete

In this section, we prove that MBP-PQ is NP-complete for uni-chromosomal and multi-chromosomal permutations, on unsigned markers. We first consider uni-chromosomal case. We reduce X3C (Exact Cover by 3-Sets) to MBP-PQ. Recall that the input for X3C is a set of 3-sets $S = \{S_1, S_2, ..., S_m\}$. Each set S_i contains exactly 3 elements from a base set $V = \{v_1, v_2, ..., v_n\}$, where $n = 3q$ for some integer q. The problem is to decide whether there are q 3-sets in S which cover each element in V exactly once.

MBP-PQ is obviously in NP and we now show that X3C can be reduced to MBP-PQ in polynomial time.

We first outline the difficulty in the proof and how to handle them one by one. In terms of generating permutations, P-nodes give the maximum amount of freedom while Q-nodes give the minimum amount of freedom. So we need to somehow balance the use of P-nodes with Q-nodes. (1) In a solution for X3C, each element belongs to exactly one selected 3-set. We enforce this by constructing a sub-tree in T_1 for each element, using both P- and Q-nodes, such that the element will appear exactly once in the final solution. (2) The second difficulty is to make sure that we must construct a subtree in T_2 such that the number of possible adjacencies (non-breaking point) it could generate has a fixed pattern. We construct such a sub-tree, using no P-nodes, for each 3-set. Once these difficulties are resolved, we still need to have a match between the possible adjacencies in T_1 and T_2; moreover, these matches imply a solution for X3C. Next we present the details.

We first construct T_1 as follows. The root of T_1, $r(T_1)$, is a Q-node. Each of the children F_i of the root corresponds to an element v_i in V and is of 4 levels (with some leaves possibly compressed in level-3, see Figure 2 (A)), and these children are further separated by peg markers (which are leaf nodes directly under the root $r(T_1)$). Note that peg markers are only used to separate T_f's. Let v_i appear in $S_{p_1}, S_{p_2}, ..., S_{p_t}$. For each v_i, we construct a subtree F_i as follows. The left child of $r(F_i)$ is a P-node which contains t Q-nodes as children, and the contents of these Q-nodes are: $v_{i,p_1} s'_{i,p_2}, v_{i,p_2} s'_{i,p_3}, ..., v_{i,p_t} s'_{i,p_1}$. The right child of $r(F_i)$ is a P-node

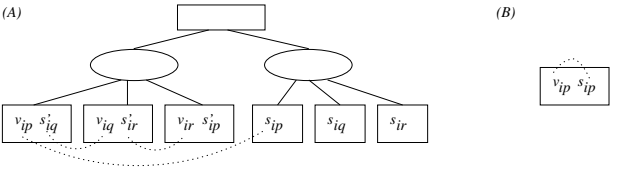


Fig. 2. The subtree F_i. In (A) and (B) the dotted arcs indicate the corresponding adjacencies. (A) shows the construction that v_i appears three times in S. (B) shows the case when v_i appears only once in S.

with t leaves: $s_{i,p_1}, s_{i,p_2}, ..., s_{i,p_t}$. Intuitively, $v_{i,p_w} s_{i,p_w}$ forms an adjacency iff S_{p_w} is selected (to cover v_i) in the final X3C solution. In Figure 2 (A), note that $t = 3$.

When v_i appears in S exactly once (say, in S_p), F_i would be a Q-node with two leaves: $v_{i,p}, s_{i,p}$ (Figure 2 (B)). We would have to use some peg markers to compose new leaf nodes to bound $s'_{i,p}$ so that it will never be adjacent to $v_{i,p}$. We will cover this special case at the end of the whole proof. At this point, we assume that each v_i appears in the 3-sets in S at least twice. We summarize the construction of F_i's with the following lemma.

Lemma 1. *F_i can generate at most one adjacency $v_{i,p_w} s_{i,p_w}$ for some $1 \leq w \leq t$.*

We now construct T_2. The root of T_2 is also a Q-node. Each of the children of $r(T_2)$ is a subtree H_p with a root being a Q-node. H_p corresponds to a 3-set $S_p = \{v_i, v_j, v_k\}$. An illustration of H_p is shown in Figure 3. Notice that H_p has five levels. We have the following lemmas.

Lemma 2. *H_p can generate exactly two sets of adjacencies in the form of $\{v_{i,p}s_{i,p}, v_{j,p}s_{j,p}, v_{k,p}s_{k,p}\}$ or $\{v_{i,p}s'_{i,p}, v_{j,p}s'_{j,p}, v_{k,p}s'_{k,p}\}$.*

Lemma 3. *T_1 and T_2 each can generate at most $3m$ adjacencies in the form of $v_{i,p}s_{i,p}$ or $v_{i,p}s'_{i,p}$.*

Proof. Following Lemma 2, T_2 can generate at most $3m$ adjacencies in the form of $v_{i,p}s_{i,p}$ or $v_{j,p}s'_{j,p}$.

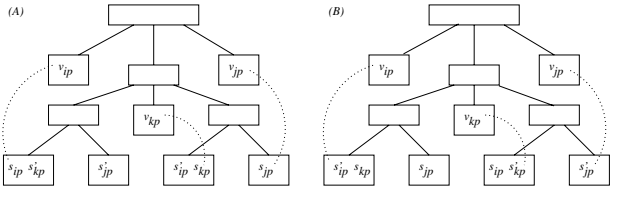


Fig. 3. The subtree H_p corresponding to $S_p = \{v_i, v_j, v_k\}$. (A) and (B) show the two different kinds of adjacencies (marked by dotted arcs).

Following Lemma 1, T_1 can generate exactly n adjacencies in the form of $v_{i,p_w}s_{i,p_w}$ for some $1 \leq w \leq t$. The remaining $3m - n$ adjacencies can obviously be generated in the form of $v_{i,p}s'_{i,p}$. □

Lemma 4. *The input X3C instance has a valid solution if and only if T_1 and T_2 can generate $3m$ adjacencies.*

Proof. The "only if" part is easy to prove. Assume that the instance (S, V) has a solution, let $S_p = \{v_i, v_j, v_k\}$ be in the solution. We permute the P-nodes in F_i and the Q-nodes in H_p such that $v_{i,p}s_{i,p}$ forms an adjacency. Following Lemma 3, we can obtain $3m$ adjacencies in T_1 and T_2.

We now prove the "if" part. Assume that T_1 and T_2 generate exactly $3m$ adjacencies, we first show that there must be n adjacencies in the form of $v_{i,p}s_{i,p}$. If it is not the case, say in T_2 some $v_{i,p}$ is never forming an adjacency with $s_{i,p}$, then the adjacencies in T_1, T_2 will not reach $3m$. Symmetrically, if in T_1 one of the subtrees F_i cannot generate t adjacencies, then there is no way T_1, T_2 can generate $3m$ adjacencies.

Now assume that among the $3m$ adjacencies in T_1, T_2 there are n adjacencies in the form of $v_{i,p}s_{i,p}$, we argue that they exactly present a corresponding solution for X3C. By the way we construct T_1, if $v_{i,p}$ forms an adjacency with $s_{i,p}$ then it implies that S_p is selected as part of the solution for the X3C instance. As we have exactly n adjacencies in the form of $v_{i,p}s_{i,p}$, each of the element appears in the X3C solution exactly once and we have a valid solution for the X3C instance (S, V). □

Theorem 1. *MBP-PQ is NP-complete for uni-chromosomal unsigned permutations.*

Proof. Now it is necessary to cover the special case when v_i appears in S exactly once. In this case we use some peg markers as leaves to bound $s'_{i,p}$ such that it will never be adjacent to $v_{i,p}$. The peg markers will be directly under the roots of T_1 and T_2 so we can order them in increasing and decreasing order respectively so that the peg markers will not form adjacencies in T_1 and T_2. It is easy to see that we will not use more than $O(n)$ peg markers.

Let N be the number of peg markers used in the construction. Following Lemma 4, there are $9m$ markers in T_1 and T_2. Therefore, the input X3C instance has a valid solution if and only if T_1 and T_2 can generate two permutations with $N + 6m - 1$ breakpoints.

It is clear that the whole transformation takes linear time. Hence, MBP-PQ is NP-complete. □

We can extend the proof to the multi-chromosomal case. Given an instance (T_1, T_2) of the uni-chromosomal case, create an instance (T'_1, T'_2) by adding to T_1 (resp. T_2) a P-node root and two children Q-nodes with each 4 leaves $n + 1, n + 2, n + 3, n + 4$ (resp. $n + 2, n + 4, n + 1, n + 3$), in this order in both cases, and $n + 5, n + 6, n + 7, n + 8$ (resp. $n + 6, n + 8, n + 5, n + 7$), again in this order in both cases. There are no common telomeres in T_1 and T_2. Therefore, (T_1, T_2)

has breakpoint distance K if and only if (T'_1, T'_2) has breakpoint distance $K + 8$ because we add 8 markers that do not form any adjacency, neither common telomere.

Corollary 1. *MBP-PQ is NP-complete for multi-chromosomal unsigned permutations.*

It is open whether one can design efficient FPT and/or approximation algorithms for the optimization version of MBP-PQ.

4 An FPT Algorithm for One-Sided MBP-PQ and p-MBM-PQ

In this section, we solve both One-Sided MBP-PQ and p-MBM-PQ with an FPT algorithm, whose parameter is the value of the optimal breakpoint distance. We first describe our algorithm for the uni-chromosomal case, then discuss its generalization to the multi-chromosomal case.

A graphical representation of PQ-trees. We first introduce a graph-like representation of a PQ-tree, that encodes the adjacency constraints between markers, and was used in [6] to represent ancestral genomes in a linear-like way. The graph G associated to a PQ-tree T has vertices for all nodes (internal and leaves) of T except the root, if it is a P-node. We call the vertices that correspond to leaves *markers*. And the vertices corresponding to P-nodes (resp. Q-nodes) are called *super* P-nodes (resp. Q-nodes). Edges of G are defined only between pairs of markers (or, of course, two super-nodes which must be adjacent): two markers x and y define an edge (x, y) if and only if they are consecutive children of a Q-node. Edges of G are called *black edges*. See Figure 1.

We also add an additional structure on G by embedding the vertices following the recursive structure of T: the vertices of G corresponding to the children of a node are embedded into the vertex representing this node (see Figure 1 (B)). A vertex (leaf or super-node) X is *contained* in another vertex Z if $X \neq Z$ and the node corresponding to X is a descendant of the one corresponding to Z in T (hence Z is a super-node); as a consequence, all the strings generated by X are substrings of those generated by Z.

We now describe how to augment the graph representation G_1 of a PQ-tree T_1 using another permutation s_2. It turns out that this will be the basis for us to handle the ancestral genome analysis when a phylogeny is given. We start with G_1, and then add an edge, called a *blue* edge, (x, y) in G_1 for every adjacency xy in s_2. We denote this new graph G'_1 (note that G'_1 conserves the embedding structure we defined on G_1: only blue edges are added). The *degree* of a super-node X in G'_1 is the number of edges that connects a marker inside X to a marker outside X. See Figure 1 and Figure 4.

At this point, it is easy to see that the One-Sided MBP-PQ Problem is closely related to the classical Minimum Path Cover Problem.

An FPT algorithm for the One-Sided MBP-PQ Problem. We first state an easy lemma that describes constraints on the blue edges that can be conserved in an optimal solution of the problem.

Lemma 5. *An optimal solution for One-Sided MBP-PQ can be obtained by performing the following operations on G'_1.*

1. *If a marker x is in the middle of a Q-node Y which contains x, then one can delete all the blue edges incident to x to obtain an optimal solution.*
2. *If a marker x is of degree greater than two, then an optimal solution could be obtained by allowing at most two blue edges connecting to x.*
3. *If a super-node X is of degree greater than two, then an optimal solution could be obtained by allowing at most two blue edges connecting to some markers inside X.*

Let r be the maximum degree of a super node, after all edge deletion operations at Step 1 of Lemma 5 have been performed. (If $r \leq 2$ the problem is trivially solvable. So we assume that $r \geq 3$.) The principle of the FPT algorithm is to use a bounded search tree [8] that considers super nodes of degree at least three and, for such a node X, conserves 2 blue edges that link a marker inside X and a marker outside X. Let K be the optimal solution value for One-Sided MBP-PQ, and $f(K)$ be the size (number of nodes) of the search tree. It is sufficient to keep deleting edges such that the resulting nodes have degree at most two, so we have the following recurrence relation

$$f(K) = \begin{cases} 0 & \text{if } K = 0, \\ 1 & \text{if } K = 1, \\ \leq \binom{r}{r-2} f(K - r + 2) & \text{if } K > 1. \end{cases}$$

The main recurrence can be simplified as

$$f(K) \leq \binom{r}{2} f(K - r + 2) = \frac{r(r-1)}{2} f(K - r + 2).$$

This recurrence achieves its maximum value when $r = 3$. Therefore,

$$f(K) \leq 3^K.$$

Once K blue edges are deleted from G', all we need to do is to check whether the resulting graph on Σ defined by the markers and the remaining black and blue edges is composed of paths. If less than K blue edges are deleted and there is no vertex of degree at least three left, we can check whether there are still any (disjoint) cycles left, if so, then delete the blue edges accordingly to break these cycles. If, after K blue edges are deleted and no valid solution is found, then we report 'No solution of size K'. This can be easily done in $O(n)$ time as at this point the maximum degree of any vertex is at most two. Therefore, we can use this bounded search tree method to obtain an algorithm which runs in $O(3^K n)$ time, once G'_1 is computed.

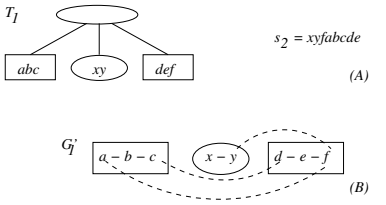


Fig. 4. An example for the FPT algorithm for One-Sided MBP-PQ

In Figure 4, we show a simple example for the algorithm. An example of T_1 and s_2 is illustrated in Figure 4 (A). The augmented graph G'_1 is shown in Figure 4 (B). The optimal solution value is $K = 1$. According to the algorithm, we will have to delete one blue (or dashed) edge in G'_1. The algorithm has the choice of deleting either (a, f), (y, f), or (c, d). Clearly, deleting (a, f) gives us the optimal solution with $s_1 = abcdefyx$ and exactly one breakpoint between s_1 and $s_2 = xyfabcde$. Deleting (y, f) or (c, d) alone both leads to infeasible solutions.

Theorem 2. *One-Sided MBP-PQ can be solved in $O(3^K n)$ time for uni-chromosomal signed and unsigned permutations, where n is the number of markers and K is the number of breakpoints in the optimal solution.*

Solving the p-MBM-PQ Problem. It is easy to see that p-MBM-PQ can be solved in $O(3^K n)$ time as well. The idea is to compute the graph G for the input PQ-tree T and then add blue edges from adjacencies in s_i, for $i = 1, \ldots, p$. Now a blue edge (x, y) is weighted, with the weight corresponding to the total number of adjacencies xy or yx in s_i, for $i = 1, \ldots, p$. So such a weight can be an integer in [1,p]. Let this augmented (weighted) graph be G''. Then the problem is clearly equivalent to deleting blue edges with a total weights of $K' \leq K$ from G'' such that the resulting graph is composed of paths. If there are K'' such paths, then adjacencies need to be added to transform them into a single path, and arbitrary adjacencies can be used, each contributing p to the breakpoint distance, that is then $K' + p(K'' - 1)$. This leads to the following result.

Corollary 2. *p-MBM-PQ can be solved in $O(3^K n)$ time for uni-chromosomal signed and unsigned permutations.*

Note that the actual running time of the FPT algorithm we described is in general much faster than $O(3^K n)$ as any adjacency in one of the genomes s_i that is discarded following Lemma 5.(1) increases the breakpoint distance by 1 but is not considered in the computation. More formally, if d is the number of edges discarded due to Lemma 5.(1), the running time is in fact $O(3^{K-d} n)$. This has been confirmed in our initial computational results. We can also immediately apply our algorithm to the variant where the median is constrained to contain only adjacencies that appear in at least one permutation s_i, which is also NP-hard for the classical Breakpoint-Median Problem [5]. Indeed, it suffices to forbid deleting blue edges that disconnects the augmented graph, which is obviously connected at first.

Handling multi-chromosomal permutations. We need here to account for two things: the set of generated permutations is different (larger in fact) and the breakpoint distance requires to consider common telomeres. To deal with both of these issues, we add in the augmented graph a vertex W, that represents telomeres, and a blue edge (W, a) for every telomere a in the s_i's. Then, a set of blue edges defining a valid permutation implies that, once edges (W, a) are discarded, the resulting edges comprise of a set of paths. Finally, as common telomeres contribute to half the weight of common adjacencies in the breakpoint distance formula, when the bounded search discards a blue edge (W, a), it increases the distance by 1/2 instead of 1. This proves the following result.

Corollary 3. *p-MBM-PQ can be solved in* $O(3^{2K}n)$ *time for multi-chromosomal signed and unsigned permutations.*

5 Application to Real Datasets

We present here preliminary computational results on some mammalian and yeast genomes to illustrate the ability of our FPT algorithm to handle real datasets, using a regular Lenovo laptop and C++. Precise data and results are available at the URL http://www.cs.montana.edu/bhz/PQ-TREE.html. In both cases, we change a multi-choromosomal genome into a uni-chromosomal signed permutation; as a consequence, we do not compute exactly the breakpoint distance as defined in [17], as we might create conserved adjacencies and we ignore common telomeres in the computation of the distance. But these results are presented to illustrate the ability of our algorithm to handle datasets with PQ-trees generating a large number of permutations. The running times are still high (varying from two days to about a week), but they are already better than what the theoretical results imply (for the three cases, we have $K = 69, 108$, and 348).

The mammalian dataset we use is from the following simple phylogenetic tree of five species, (((Human,Macaca)I,(Mouse,Rat)II)III,Dog), given in Newick format, and we are interested in the ancestors of Human and Macaca (node I) and Mouse and Rat (node II). Permutations and PQ-trees at nodes I and II were generated using methods as described as in [15]. In this case, $n = 689$. In the companion webpage, we show in detail the dataset and the sequences generated using the FPT algorithm for 2-MBM-PQ, for node I and node II. For node I, we found that the optimal breakpoint distance is 69, and for node II, the optimal distance is larger, at 108. Notice that these solutions are not unique (in fact in both cases there are about 10! permutations which minimizes $d_b(s, s_1)+d_b(s, s_2)$, due to that the roots of the trees are both P-nodes). So an exhaustive search would not work to generate an optimal permutation for node III.

The yeast data is from [13], the PQ-tree has a root which is a Q-node with 34 children (which are all Q-nodes or leaves). Among these 34 children, 8 of them are leaves. We found an optimal distance of 348. If we wanted to enumerate all generated permutations, we would have to try 2^{26} different permutations.

6 Conclusion

In this paper, we make the first step in comparing the similarity of PQ-trees, with application to comparative genomics. While the general problem is NP-complete (not a surprise!), we show that several interesting cases, that are relevant from an applied point of view, are in FPT, parameterized by the optimal breakpoint distance. We also present some preliminary computational results.

Our first open question is how to construct a general graph or hypergraph incorporating all the information regarding two PQ-trees T_1 and T_2. Without such a (hyper?) graph, it seems difficult to design approximation and FPT algorithms for the optimization version of MBP-PQ (and possibly some other ways to compare the similarity of T_1 and T_2). A related question would be to find an FPT algorithm for MBP-PQ whose parameter is the breakpoint distance. When this distance is zero, the problem is in fact easy to solve: it is easy to decide if T_1 and T_2 can generate the same permutation [4,2].

How to improve the efficiency of the FPT algorithms for One-Sided MBP-PQ and p-MBM-PQ also makes interesting questions. The only other FPT algorithm for a breakpoint median problem, described in [11], has complexity $O(2.15^K n)$, and it remains to see how the ideas used in that algorithm can be translated to the case where the median is constrained to be generated by a given PQ-tree.

Regarding p-MBM-PQ, it is recently proved in [17] that the Breakpoint Median Problem for signed multi-chromosomal genomes is tractable if the median is allowed to have circular chromosomes; it can indeed be solved by a simple maximum weight matching algorithm. In the case of the p-MBM-PQ, the corresponding problem would allow that, in the median, the leaves of one or more subtree rooted at children of the root form a circular chromosome. The complexity of this problem is open.

Finally, what if we consider the problems under other distances such as the DCJ (Double-Cut-and-Join) distance? Intuitively, we can expect that such problems are hard too. For example, comparing two PQ-trees of height 2 (every path between a leaf and the root contains at most two edges) whose internal nodes are all P-nodes is equivalent to computing the syntenic distance [9] between two genomes represented by the gene content of their chromosomes and with no gene order information, which is NP-hard [7].

Acknowledgments

This research is partially supported by NSF grant DMS-0918034, by NSF of China under grant 60928006, and by NSERC Discovery Grant 249834-2006.

References

1. Alizadeh, F., Karp, R., Weisser, D., Zweig, G.: Physical mapping of chromosomes using unique probes. J. Comp. Biol. 2, 159–184 (1995)
2. Bergeron, A., Blanchette, M., Chateau, A., Chauve, C.: Reconstructing ancestral gene orders using conserved intervals. In: Jonassen, I., Kim, J. (eds.) WABI 2004. LNCS (LNBI), vol. 3240, pp. 14–25. Springer, Heidelberg (2004)

3. Blin, G., Blais, E., Guillon, P., Blanchette, M., ElMabrouk, N.: Inferring Gene Orders from Gene Maps Using the Breakpoint Distance. In: Bourque, G., El-Mabrouk, N. (eds.) RECOMB-CG 2006. LNCS (LNBI), vol. 4205, pp. 99–102. Springer, Heidelberg (2006)
4. Booth, K., Lueker, G.: Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms. J. Computer and System Sciences 13, 335–379 (1976)
5. Bryant, D.: The complexity of the breakpoint median problem. Technical Report CRM-2579. Centre de Recherches en Mathématiques, Université de Montréal (1998)
6. Chauve, C., Tannier, E.: A methodological framework for the reconstruction of contiguous regions of ancestral genomes and its application to mammalian genome. PLoS Comput. 4:e1000234 (2008)
7. DasGupta, B., Jiang, T., Kannan, S., Li, M., Sweedyk, E.: On the Complexity and Approximation of Syntenic Distance. Discrete Appl. Math. 88(1–3), 59–82 (1998)
8. Downey, R., Fellows, M.: Parameterized Complexity. Springer, Heidelberg (1999)
9. Feretti, V., Nadeau, J.H., Sankoff, D.: Original synteny. In: Hirschberg, D.S., Meyers, G. (eds.) CPM 1996. LNCS, vol. 1075, pp. 159–167. Springer, Heidelberg (1996)
10. Fu, Z., Jiang, T.: Computing the breaking distance between partially ordered genomes. In: APBC 2007, pp. 237–246 (2007)
11. Gramm, J., Niedermeier, R.: Breakpoint medians and breakpoint phylogenies: A fixed-parameter approach. Bioinformatics 18(Suppl. 2), S128–S139 (2002)
12. Hannenhalli, S., Pevzner, P.: Transforming cabbage into turnip: polynomial algorithm for sorting signed permutations by reversals. J. ACM 46(1), 1–27 (1999)
13. Jean, G., Sherman, D.M., Nikolski, M.: Mining the semantic of genome super-blocks to infer ancestral architectures. J. Comp. Biol. 16(9), 1267–1284 (2009)
14. Landau, G., Parida, L., Weimann, O.: Gene proximity analysis across whole genomes via PQ-trees. J. Comp. Biol. 12, 1289–1306 (2005)
15. Ouangraoua, A., McPherson, A., Tannier, E., Chauve, C.: Insight into the structural evolution of amniote genomes. In: Preliminary version in Cold Spring Harbor Laboratory Genome Informatics Meeting 2009, poster 137 (2009)
16. Pe'er, I., Shamir, R.: The median problems for breakpoints are NP-complete. Elec. Colloq. Comput. Complexity, TR-98-071 (1998)
17. Tannier, E., Zheng, C., Sankoff, D.: Multichromosomal median and halving problems under different genomic distances. BMC Bioinformatics 10, 120 (2009)
18. Zheng, C., Lennert, A., Sankoff, D.: Reversal distance for partially ordered genomes. Bioinformatics 21(Suppl. 1), i502–i508 (2005)

On the Parameterized Complexity of Some Optimization Problems Related to Multiple-Interval Graphs*

Minghui Jiang

Department of Computer Science, Utah State University, Logan, UT 84322, USA
mjiang@cc.usu.edu

Abstract. We show that for any constant $t \geq 2$, k-INDEPENDENT SET and k-DOMINATING SET in t-track interval graphs are W[1]-hard. This settles an open question recently raised by Fellows, Hermelin, Rosamond, and Vialette. We also give an FPT algorithm for k-CLIQUE in t-interval graphs, parameterized by both k and t, with running time $\max\{t^{O(k)}, 2^{O(k \log k)}\} \cdot \mathrm{poly}(n)$, where n is the number of vertices in the graph. This slightly improves the previous FPT algorithm by Fellows, Hermelin, Rosamond, and Vialette. Finally, we use the W[1]-hardness of k-INDEPENDENT SET in t-track interval graphs to obtain the first parameterized intractability result for a recent bioinformatics problem called MAXIMAL STRIP RECOVERY (MSR). We show that MSR-d is W[1]-hard for any constant $d \geq 4$ when the parameter is either the total length of the strips, or the total number of adjacencies in the strips, or the number of strips in the optimal solution.

1 Introduction

The *intersection graph* $\Omega(\mathcal{F})$ of a family of sets $\mathcal{F} = \{S_1, \ldots, S_n\}$ is the graph with $\mathcal{F}$ as the vertex set and with two different vertices S_i and S_j adjacent if and only if $S_i \cap S_j \neq \emptyset$. The family $\mathcal{F}$ is called a *representation* of the graph $\Omega(\mathcal{F})$.

Let t be an integer at least two. A *t-interval* is the union of t disjoint intervals in the real line. A *t-track interval* is the union of t disjoint intervals in t disjoint parallel lines called tracks, one interval on each track. A *t-interval graph* is the intersection graph of a family of t-intervals. A *t-track interval graph* is the intersection graph of a family of t-track intervals. If all intervals in the representation of a t-interval graph have unit lengths, then the graph is called a *unit t-interval graph*. Similarly for unit t-track interval graphs.

As generalizations of the ubiquitous interval graphs, multiple-interval graphs such as t-interval graphs and t-track interval graphs have wide applications, traditionally to scheduling and resource allocation [3,5], and more recently to bioinformatics [17,2,19,7,14,1]. In particular, 2-interval graphs and 2-track interval graphs are natural models for the similar regions of DNA sequences [17,2,1] and for the helices of RNA secondary structures [19,7,14].

Fellows, Hermelin, Rosamond, and Vialette [9] recently initiated the study of the parameterized complexity of multiple-interval graph problems. In general graphs, the

* Supported in part by NSF grant DBI-0743670.

A. Amir and L. Parida (Eds.): CPM 2010, LNCS 6129, pp. 125–137, 2010.

three classical optimization problem k-VERTEX COVER, k-INDEPENDENT SET, and k-DOMINATING SET, parameterized by the optimal solution size k, are exemplary problems in parameterized complexity theory [8]: it is well-known that k-VERTEX COVER is in FPT, k-INDEPENDENT SET is W[1]-hard, and k-DOMINATING SET is W[2]-hard. Since t-interval graphs are a special class of graphs, all FPT algorithms for k-VERTEX COVER in general graphs immediately carry over to t-interval graphs. On the other hand, the parameterized complexity of k-INDEPENDENT SET in t-interval graphs is not at all obvious. Indeed, in general graphs, k-INDEPENDENT SET and k-CLIQUE are essentially the same problem, but in t-interval graphs, they manifest different parameterized complexities. Fellows et al. [9] showed that k-INDEPENDENT SET in t-interval graphs is W[1]-hard for any constant $t \geq 2$, then, in sharp contrast, gave an FPT algorithm for k-CLIQUE in t-interval graphs parameterized by both k and t. Similarly, the parameterized complexity of k-DOMINATING SET in t-interval graphs is not obvious either. Fellows et al. [9] showed that k-DOMINATING SET in t-interval graphs is also W[1]-hard for any constant $t \geq 2$.

At the end of their paper, Fellows et al. [9] raised three open questions. First, are k-INDEPENDENT SET and k-DOMINATING SET in 2-track interval graphs W[1]-hard? Second, is k-DOMINATING SET in t-interval graphs W[2]-hard? Third, can the parametric time-bound of their FPT algorithm for k-CLIQUE in t-interval graphs be improved?

The t disjoint tracks for a t-track interval graph can be viewed as t disjoint "host" intervals in the real line for a t-interval graph. Thus the class of t-track interval graphs is contained in the class of t-interval graphs. The containment is proper because the complete bipartite graph $K_{t^2+t-1,t+1}$ is a t-interval graph but not a t-track interval graph [21]. It is also known that for any $t \geq 1$, t-interval graphs are a proper subclass of $(t+1)$-interval graphs, and unit t-interval (resp. unit t-track interval) graphs are a proper subclass of t-interval (resp. t-track interval) graphs; see [18,12,13,10]. Fellows et al. [9] proved that k-INDEPENDENT SET and k-DOMINATING SET in unit 2-interval graphs are both W[1]-hard, hence k-INDEPENDENT SET and k-DOMINATING SET in t-interval graphs are both W[1]-hard for all $t \geq 2$. The main result of this paper is the following theorem that answers the first open question of Fellows et al. [9] and strengthens their W[1]-hardness results to encompass even the most basic subclass of multiple-interval graphs:

Theorem 1. *k-INDEPENDENT SET and k-DOMINATING SET in unit 2-track interval graphs are W[1]-hard.*

Given a graph G and a vertex-coloring $\kappa : V(G) \rightarrow \{1, 2, \ldots, k\}$, the problem k-MULTICOLORED CLIQUE is that of deciding whether G has a clique of k vertices containing exactly one vertex of each color. Fellows et al. [9] proved that k-MULTICOLORED CLIQUE is W[1]-complete, then proved that both k-INDEPENDENT SET and k-DOMINATING SET in unit 2-interval graphs are W[1]-hard by FPT reductions from k-MULTICOLORED CLIQUE. Our proof of Theorem 1 follows the same strategy. We note that this k-MULTICOLORED CLIQUE technique [9] is quickly becoming a standard tool for FPT reductions. We are unable to answer the second open question of Fellows et al. [9] on the possible W[2]-hardness of k-DOMINATING SET in

t-interval graphs, but believe that any new techniques developed for this problem would also have far-reaching influence in parameterized complexity theory.

Let's move on to the third open question. Fellows et al. [9] presented an FPT algorithm for k-CLIQUE in t-interval graphs parameterized by both k and t. They estimated that the running time of their algorithm is $t^{O(k \log k)} \cdot \mathrm{poly}(n)$, where n is the number of vertices in the graph, and asked whether the parametric time-bound of $t^{O(k \log k)}$ can be improved. Our next theorem makes some small progress on this open question:

Theorem 2. *For any constant $c \geq 3$, there is an algorithm for k-CLIQUE in t-interval graphs with running time $O(t^{ck}) \cdot O(n^c)$ if $k \leq \frac{1}{4} \cdot n^{1-1/c}$, where n is the number of vertices in the graph. In particular, there is an FPT algorithm for k-CLIQUE in t-interval graphs with running time $\max\{t^{O(k)}, 2^{O(k \log k)}\} \cdot \mathrm{poly}(n)$.*

Finally, we extend the W[1]-hardness results in Theorem 1 to a bioinformatics problem. In comparative genomic, the first step of sequence analysis is usually to decompose two or more genomes into syntenic blocks that are segments of homologous chromosomes. For the reliable recovery of syntenic blocks, noise and ambiguities in the genomic maps need to be removed first. A genomic map is a sequence of gene markers. A gene marker appears in a genomic map in either positive or negative orientation. Given d genomic maps as signed sequences of gene markers, MAXIMAL STRIP RECOVERY (MSR-d) [22,6] is the problem of finding d subsequences, one subsequence of each genomic map, such that the total length ℓ of the strips in these subsequences is maximized. Here a *strip* is a maximal string of at least two markers such that either the string itself or its signed reversal appears contiguously as a substring in each of the d subsequences in the solution. Without loss of generality, we can assume that all markers appear in positive orientation in the first genomic map, as in [22,15]. For example, the two genomic maps (the markers in negative orientation are underlined)

1	2	3	4	5	6	7	8	9	10	11	12
$\underline{8}$	$\underline{5}$	$\underline{7}$	$\underline{6}$	4	1	3	2	$\underline{12}$	$\underline{11}$	$\underline{10}$	9

have two subsequences

1	3		6	7	8		10	11	12
$\underline{8}$	$\underline{7}$	$\underline{6}$		1	3		$\underline{12}$	$\underline{11}$	$\underline{10}$

of the maximum total strip length 8. The strip $\langle 1, 3\rangle$ is positive and forward in both subsequences; the other two strips $\langle 6, 7, 8\rangle$ and $\langle 10, 11, 12\rangle$ are positive and forward in the first subsequence, but are negative and backward in the second subsequence. The four markers $2, 4, 5, 9$ are deleted. Intuitively, the strips are syntenic blocks, and the deleted markers are noise and ambiguities in the genomic maps.

A strip of length $l \geq 2$ has exactly $l-1$ adjacencies between consecutive markers. In general, m strips of total length l have $l - m$ adjacencies. Besides the total strip length, the total number of adjacencies in the strips is also a natural objective function of MSR-d. For both objective functions, it is known that MSR-d is APX-hard for any $d \geq 2$ [15], and moreover is NP-hard to approximate within $\Omega(d/\log d)$ [16]. On the other hand, for any constant $d \geq 2$, MSR-d admits a polynomial-time $2d$-approximation [6]. See also [20,4] for some related results. Our following theorem gives the first parameterized intractability result for MSR-d:

Theorem 3. MSR-d *for any constant* $d \geq 4$ *is W[1]-hard when the parameter is either the total length of the strips, or the total number of adjacencies in the strips, or the number of strips in the optimal solution. This holds even if all gene markers are distinct and appear in positive orientation in each genomic map.*

2 k-Independent Set

In this section we show that k-INDEPENDENT SET in unit 2-track interval graphs is W[1]-hard. We first review the previous FPT reduction from k-MULTICOLORED CLIQUE in general graphs to k-INDEPENDENT SET in unit 2-interval graphs [9], then show how to modify it into an FPT reduction from k-MULTICOLORED CLIQUE in general graphs to k-INDEPENDENT SET in unit 2-track interval graphs.

Previous Reduction. Let (G, κ, k) be an instance of k-MULTICOLORED CLIQUE. The construction consists of $k+\binom{k}{2}$ groups of unit intervals occupying disjoint regions of the real line. Among the $k+\binom{k}{2}$ groups, k groups are vertex gadgets, one for each color, and $\binom{k}{2}$ groups are edge gadgets, one for each pair of distinct colors. The vertex gadgets and the edge gadgets are then linked together, according to the incidence relation between the vertices and the edges, by the validation gadget. Each vertex gadget selects a vertex of a particular color. Each edge gadget selects an edge of a particular pair of colors. The validation gadget ensures the consistency of the selections.

Vertex selection: For each color i, $1 \leq i \leq k$, let V_i be the set of vertices with color i. The vertex gadget for the color i consists of a group of intervals that can viewed as a table[1] with $|V_i|$ rows and $k+1$ columns. Each row of the table corresponds to a distinct vertex $u \in V_i$: the first interval and the last interval together form a *vertex 2-interval* $\widehat{u_i}$; the other intervals, each associated with a distinct color $j \in \{1, \ldots, k\} \setminus \{c\}$ and denoted by $\overline{u_i * j}$, and are used for validation. The intervals in the table are arranged in a parallelogram formation with slanted columns: the intervals in each row are disjoint; the intervals in each column intersect at a common point; the intervals in lower rows have larger horizontal offsets such that each interval also intersects all intervals in higher rows in the next column.

Edge selection: For each pair of distinct colors i and j, $1 \leq i < j \leq k$, let E_{ij} be the set of edges uv such that u has color i and v has color j. The edge gadget for the pair of colors ij consists of a group of intervals that can viewed as a table with $|E_{ij}|$ rows and 4 columns. Each row of the table corresponds to a distinct edge $uv \in E_{ij}$: the first interval and the fourth interval together form an *edge 2-interval* $\widehat{u_i v_j}$; the second and the third intervals, denoted by $\overline{u_i v_j}$ and $\overline{v_j u_i}$, respectively, are used for validation. Again the intervals in the table are arranged in a parallelogram formation.

Validation: For each edge uv such that u has color i and v has color j, the validation gadget includes two *validation 2-intervals* $\overleftarrow{u_i v_j}$ and $\overrightarrow{u_i v_j}$: the 2-interval $\overleftarrow{u_i v_j}$ consists of the interval $\overline{u_i v_j}$ and the interval $\overline{u_i * j}$; the 2-interval $\overrightarrow{u_i v_j}$ consists of the interval $\overline{v_j u_i}$

[1] The table is of course only a visualization device; in reality the intervals in all rows of the table are in the same line.

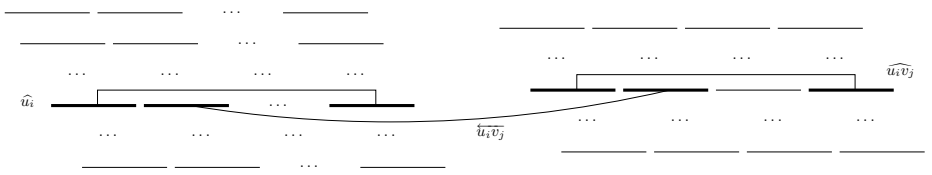


Fig. 1. Construction for k-INDEPENDENT SET. On the left is a vertex gadget. On the right is an edge gadget. The vertex 2-interval $\widehat{u_i}$ selects the vertex u for the color i. The edge 2-interval $\widehat{u_i v_j}$ selects the edge uv for the pair of colors ij. The validation 2-interval validates the selections.

and the interval $\overline{v_j *_i}$. Note that each validation 2-interval consists of an interval from an edge gadget and an interval from a vertex gadget.

In summary, the following family $\mathcal{F}$ of 2-intervals are constructed:

$$\mathcal{F} = \big\{\widehat{u_i} \mid u \in V_i,\, 1 \le i \le k\big\} \cup \big\{\widehat{u_i v_j}, \overleftarrow{u_i v_j}, \overrightarrow{u_i v_j} \mid uv \in E_{ij},\, 1 \le i < j \le k\big\}.$$

Refer to Figure 1 for an example. Now set the parameter $k' = k + 3\binom{k}{2}$. It remains to show that G has a k-multicolored clique if and only if $\mathcal{F}$ has a k'-independent set.

For the direct implication, it is easy to verify that if $K \subseteq V(G)$ is a k-multicolored clique, then the following subset of 2-intervals is a k'-independent set in $\mathcal{F}$:

$$\big\{\widehat{u_i} \mid u \in K,\, i = \kappa(u)\big\} \cup \big\{\widehat{u_i v_j}, \overleftarrow{u_i v_j}, \overrightarrow{u_i v_j} \mid u, v \in K,\, i = \kappa(u),\, j = \kappa(v)\big\}.$$

For the reverse implication, suppose that $\mathcal{I}$ is a k'-independent set in $\mathcal{F}$. By construction, $\mathcal{I}$ can include at most one vertex 2-interval for each color, and at most one edge 2-interval plus at most two validation 2-intervals for each pair of distinct colors. Since $k' = k + 3\binom{k}{2}$, $\mathcal{I}$ must include exactly one vertex 2-interval for each color, and exactly one edge 2-interval plus two validation 2-intervals for each pair of distinct colors. It follows that the $2\binom{k}{2} = (k-1)k$ validation 2-intervals in $\mathcal{I}$ have exactly two intervals in each edge gadget, and exactly $k-1$ intervals in each vertex gadget. Moreover, in each vertex gadget, the intervals of the vertex 2-interval and the $k-1$ validation 2-intervals in $\mathcal{I}$ must be in the same row. Similarly, in each edge gadget, the intervals of the edge 2-interval and the two validation 2-intervals in $\mathcal{I}$ must be in the same row. Since all intervals in the same row of a vertex gadget are associated with the same vertex, and all intervals in the same row of an edge gadget are associated with the same edge, the vertex selection and the edge selection must be consistent. Thus the k vertex 2-intervals in $\mathcal{I}$ corresponds to a k-multicolored clique in G.

This completes the review of the previous reduction. Before we present the new reduction, let's pause for a moment and ponder why this reduction works. You may have noticed that the central idea behind the construction is essentially a geometric packing argument. Consider each vertex 2-interval as a container of capacity $k-1$, each edge 2-interval as a container of capacity 2, and the validation 2-intervals as items to be packed. Then, in order to pack each container to its full capacity, the items in each container must be arranged in a regular pattern, that is, all intervals in each vertex or edge gadget must be in the same row.

New Reduction. We now modify the previous construction to transform each 2-interval into a 2-track interval. Move all vertex gadgets to track 1, and move all edge gadgets to track 2. Then all validation 2-intervals are immediately transformed into 2-track intervals. It remains to fix the vertex 2-intervals on track 1 and the edge 2-intervals on track 2.

We first fix the vertex 2-intervals on track 1. Consider the vertex gadget for the vertices V_i with color i. To fix the vertex 2-intervals in this gadget, we replace each 2-interval $\widehat{u}_i$ by two 2-track intervals $\widehat{u}_{i\,\mathrm{left}}$ and $\widehat{u}_{i\,\mathrm{right}}$ as follows:

- On track 1, let the intervals of $\widehat{u}_{i\,\mathrm{left}}$ and $\widehat{u}_{i\,\mathrm{right}}$ be the left and the right intervals, respectively, of $\widehat{u}_i$.
- On track 2, put the intervals of $\widehat{u}_{i\,\mathrm{left}}$ and $\widehat{u}_{i\,\mathrm{right}}$ for all $u \in V_i$ in a separate region, and arrange them in a parallelogram formation with $|V_i|$ rows and 2 columns: $\widehat{u}_{i\,\mathrm{left}}$ in the right column, $\widehat{u}_{i\,\mathrm{right}}$ in the left column. As usual, the intervals are disjoint in each row and are pairwise intersecting in each column, moreover the columns are slanted such that each interval in the left column intersects all intervals in higher rows in the right column.

Refer to Figure 2 for an illustration of the vertex 2-track intervals on the two tracks. In a similar way (with the roles of track 1 and track 2 reversed), we replace each edge 2-interval $\widehat{u_iv_j}$ by two 2-track intervals $\widehat{u_iv_j}_{\,\mathrm{left}}$ and $\widehat{u_iv_j}_{\,\mathrm{right}}$. Then all 2-interval are transformed into 2-track intervals. The following family $\mathcal{F}$ of 2-track intervals are constructed:

$$\mathcal{F} = \big\{\widehat{u}_{i\,\mathrm{left}}, \widehat{u}_{i\,\mathrm{right}} \mid u \in V_i,\, 1 \le i \le k\big\} \cup \big\{\widehat{u_iv_j}_{\,\mathrm{left}}, \widehat{u_iv_j}_{\,\mathrm{right}}, \overleftarrow{u_iv_j}, \overrightarrow{u_iv_j} \mid uv \in E_{ij},\, 1 \le i < j \le k\big\}.$$

Now set the parameter $k' = 2k + 4\binom{k}{2}$. It remains to show that G has a k-multicolored clique if and only if $\mathcal{F}$ has a k'-independent set.

For the direct implication, it is easy to verify that if $K \subseteq V(G)$ is a k-multicolored clique, then the following subset of 2-track intervals is a k'-independent set in $\mathcal{F}$:

$$\big\{\widehat{u}_{i\,\mathrm{left}}, \widehat{u}_{i\,\mathrm{right}} \mid u \in K,\, i = \kappa(u)\big\} \cup \big\{\widehat{u_iv_j}_{\,\mathrm{left}}, \widehat{u_iv_j}_{\,\mathrm{right}}, \overleftarrow{u_iv_j}, \overrightarrow{u_iv_j} \mid u, v \in K,\, i = \kappa(u),\, j = \kappa(v)\big\}.$$

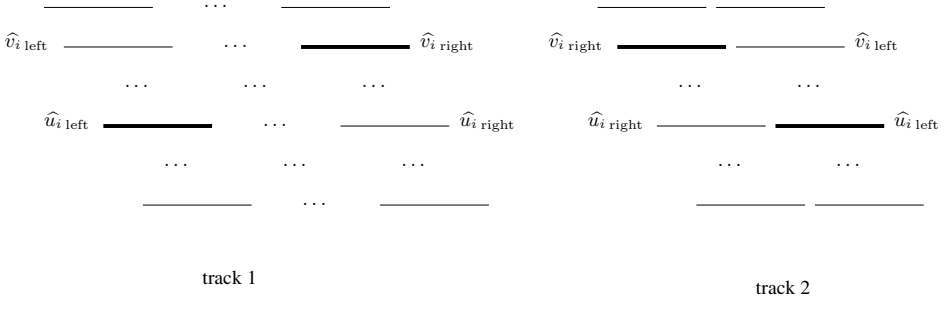


Fig. 2. Transforming vertex 2-intervals into 2-track intervals for k-INDEPENDENT SET

For the reverse implication, suppose $\mathcal{I}$ is a k'-independent set in $\mathcal{F}$. The same argument as before shows that $\mathcal{I}$ must include exactly *two* vertex 2-track intervals for each color, and exactly *two* edge 2-track intervals plus two validation 2-track intervals for each pair of distinct colors. Refer back to Figure 2. Let $\widehat{u_i}_{\text{ left}}$ and $\widehat{v_i}_{\text{ right}}$ be the two vertex 2-track intervals in $\mathcal{I}$ for some color i. The intersection pattern of the vertex 2-track intervals for V_i on track 2 ensures that the row of u must not be higher than the row of v. Without loss of generality, we can assume that they are in the same row, i.e., $u = v$, so that the set of validation intervals in the middle columns on track 1 that are dominated by $\widehat{u_i}_{\text{ left}}\,\widehat{v_i}_{\text{ right}}$ is minimal (or, in terms of geometric packing, this gives the container $\widehat{u_i}_{\text{ left}}\,\widehat{v_i}_{\text{ right}}$ the largest capacity on track 1). Thus we can assume that the two vertex 2-track intervals for each color i form a pair $\widehat{u_i}_{\text{ left}}\,\widehat{u_i}_{\text{ right}}$ for the same vertex u. Similarly, we can assume that the two edge 2-track intervals for each pair of colors ij form a pair $\widehat{u_iv_j}_{\text{ left}}\,\widehat{u_iv_j}_{\text{ right}}$ for the same edge uv. Then the same argument as before shows that the k pairs of vertex 2-track intervals in $\mathcal{I}$ corresponds to a k-multicolored clique in G.

3 k-Dominating Set

In this section we show that k-DOMINATING SET in unit 2-track interval graphs is W[1]-hard. We first review the previous FPT reduction from k-MULTICOLORED CLIQUE in general graphs to k-DOMINATING SET in unit 2-interval graphs [9], then show how to modify it into an FPT reduction from k-MULTICOLORED CLIQUE in general graphs to k-DOMINATING SET in unit 2-track interval graphs.

Previous Reduction. Let (G, κ, k) be an instance of k-MULTICOLORED CLIQUE. The reduction again constructs k vertex gadgets, one for each color, and $\binom{k}{2}$ edge gadgets, one for each pair of distinct colors. The vertex gadgets and the edge gadgets are then linked together by the validation gadget.

Vertex selection: For each color i, $1 \le i \le k$, let V_i be the set of vertices with color i. The vertex gadget for the color i includes one interval $\overline{*_i}$ for the color i and one interval $\overline{u_i}$ for each vertex $u \in V_i$. The interval $\overline{*_i}$ is combined with each interval $\overline{u_i}$ to form a *vertex 2-interval* $\widehat{u_i}$. The vertex gadget for V_i also includes two disjoint dummy 2-intervals that contain the left and the right endpoints, respectively, of the interval $\overline{*_i}$.

Edge selection: For each pair of distinct colors i and j, $1 \le i < j \le k$, let E_{ij} be the set of edges uv such that u has color i and v has color j. The edge gadget for the pair of colors ij includes a group of intervals that can viewed as a table with $|E_{ij}|$ rows and 3 columns. Each row of the table corresponds to a distinct edge $uv \in E_{ij}$: the left interval and the right interval together form an *edge 2-interval* $\widehat{u_iv_j}$; the middle interval, denoted by $\overline{u_iv_j}$, is used for validation. Again the intervals in the table are arranged in a parallelogram formation. The edge gadget for E_{ij} also includes two disjoint dummy 2-intervals that intersect the left intervals and the right intervals, respectively, of all edge 2-intervals $\widehat{u_iv_j}$.

Validation: For each edge uv such that u has color i and v has color j, the validation gadget includes two *validation 2-intervals* $\overleftarrow{u_iv_j}$ and $\overrightarrow{u_iv_j}$: the 2-interval $\overleftarrow{u_iv_j}$ consists

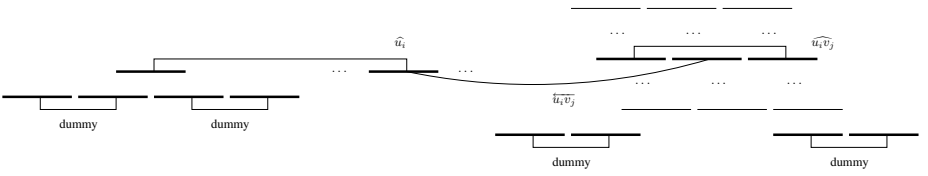


Fig. 3. Construction for k-DOMINATING SET. On the left is a vertex gadget. On the right is an edge gadget. The vertex 2-interval $\widehat{u_i}$ selects the vertex u for the color i. The edge 2-interval $\widehat{u_i v_j}$ selects the edge uv for the pair of colors ij. The validation 2-interval validates the selections.

of the interval $\overline{u_i v_j}$ and the interval $\overline{u_i}$; the 2-interval $\overrightarrow{u_i v_j}$ consists of the interval $\overline{u_i v_j}$ and the interval $\overline{v_j}$.

In summary, the following family $\mathcal{F}$ of 2-intervals are constructed:

$$\mathcal{F} = \left\{\widehat{u_i} \mid u \in V_i,\, 1 \le i \le k\right\} \cup \left\{\widehat{u_i v_j}, \overleftarrow{u_i v_j}, \overrightarrow{u_i v_j} \mid uv \in E_{ij},\, 1 \le i < j \le k\right\} \cup \textit{DUMMIES},$$

where *DUMMIES* is the set of $2k + 2\binom{k}{2}$ dummy 2-intervals, two in each vertex or edge gadget. Refer to Figure 3 for an example. Now set the parameter $k' = k + \binom{k}{2}$. It remains to show that G has a k-multicolored clique if and only if $\mathcal{F}$ has a k'-dominating set.

For the direct implication, it is easy to verify that if $K \subseteq V(G)$ is a k-multicolored clique, then the following subset of 2-intervals is a k'-dominating set in $\mathcal{F}$:

$$\left\{\widehat{u_i} \mid u \in K,\, i = \kappa(u)\right\} \cup \left\{\widehat{u_i v_j} \mid u, v \in K,\, i = \kappa(u),\, j = \kappa(v)\right\}.$$

For the reverse implication, suppose that $\mathcal{I}$ is a k'-dominating set in $\mathcal{F}$. Because every dummy 2-interval can be replaced by an adjacent vertex or edge 2-interval in a dominating set, we can assume without loss of generality that $\mathcal{I}$ does not include any dummy 2-intervals. Then, to dominate the dummy 2-intervals[2], $\mathcal{I}$ must include at least one vertex 2-interval for each color, and at least one edge 2-interval for each pair of distinct colors. Since $k' = k + \binom{k}{2}$, $\mathcal{I}$ must include exactly one vertex 2-interval for each color, and exactly one edge 2-interval for each pair of distinct colors. It follows that for each pair of distinct colors ij, the two validation 2-intervals $\overleftarrow{u_i v_j}$ and $\overrightarrow{u_i v_j}$ must be dominated by the two vertex 2-intervals $\widehat{u_i}$ and $\widehat{v_j}$, respectively. Therefore the vertex selection and the edge selection are consistent, and the k vertex 2-intervals in $\mathcal{I}$ corresponds to a k-multicolored clique in G.

New Reduction. We now modify the previous construction to transform each 2-interval into a 2-track interval. To transform the vertex 2-intervals into 2-track intervals, move the intervals $\overline{u_i}$ to track 1, and move the intervals $\overline{*_i}$ to track 2. Then, to transform

[2] We remark that the construction can be simplified by including only one dummy 2-interval for each vertex or edge gadget. Nevertheless we keep two dummy 2-intervals for each gadget in this presentation, partly for truthfulness to the original reduction, and partly for convenience in our new reduction (when we split each edge 2-interval into two 2-track intervals later, we don't have to add new dummies).

the validation 2-intervals into 2-track intervals, move all edge gadgets to track 2. The dummy 2-intervals can be fixed accordingly. It remains to fix the edge 2-intervals now on track 2.

Consider the edge gadget for the edges E_{ij} with colors ij. To fix the edge 2-intervals in this gadget, we replace each 2-interval $\widehat{u_iv_j}$ by two 2-track intervals $\widehat{u_iv_j}_{\text{ left}}$ and $\widehat{u_iv_j}_{\text{ right}}$ as follows:

- On track 2, let the intervals of $\widehat{u_iv_j}_{\text{ left}}$ and $\widehat{u_iv_j}_{\text{ right}}$ be the left and the right intervals, respectively, of $\widehat{u_iv_j}$.
- On track 1, put the intervals of $\widehat{u_iv_j}_{\text{ left}}$ and $\widehat{u_iv_j}_{\text{ right}}$ for all $uv \in E_{ij}$ in a separate region, then arrange them, together with $|E_{ij}|$ additional dummy intervals, in a parallelogram formation with $|E_{ij}|$ rows and 3 columns: $\widehat{u_i}_{\text{ left}}$ in the right column, $\widehat{u_i}_{\text{ right}}$ in the left column, and dummies in the middle column. As usual, the intervals are pairwise intersecting in each column, and the columns are slanted. But in each row the three intervals are not all disjoint: the left interval and the middle interval slightly overlap, and are both disjoint from the right interval. Now each interval in the right column intersects all intervals in lower rows in the middle column, and each interval in the left column intersects all intervals in the same or higher rows in the middle column. Finally, each of the $|E_{ij}|$ dummy intervals in the middle column is combined with an isolated dummy interval on track 2 to form a dummy 2-track interval.

Refer to Figure 4 for an illustration of the edge 2-track intervals on the two tracks. The following family $\mathcal{F}$ of 2-track intervals are constructed:

$$\mathcal{F} = \big\{\widehat{u_i} \mid u \in V_i,\, 1 \le i \le k\big\} \cup \big\{\widehat{u_iv_j}_{\text{ left}}, \widehat{u_iv_j}_{\text{ right}}, \overleftarrow{u_iv_j}, \overrightarrow{u_iv_j} \mid uv \in E_{ij},\, 1 \le i < j \le k\big\} \cup \textit{DUMMIES},$$

where *DUMMIES* is the set of $2k + 2\binom{k}{2} + |E(G)|$ dummy 2-track intervals, two in each vertex or edge gadget as before, and one more for each edge (recall the middle column of each edge gadget on track 1). Now set the parameter $k' = k + 2\binom{k}{2}$. It remains to show that G has a k-multicolored clique if and only if $\mathcal{F}$ has a k'-dominating set.

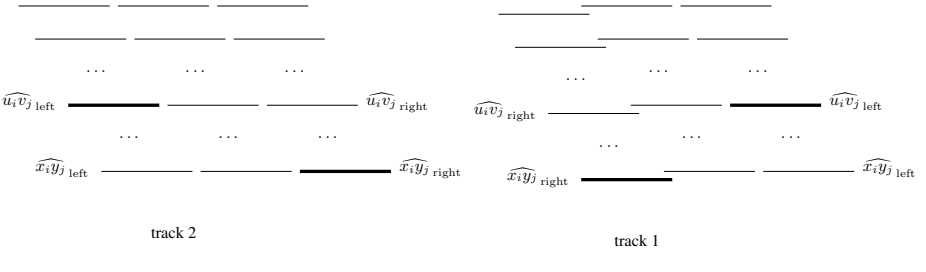


Fig. 4. Transforming edge 2-intervals into 2-track intervals for k-DOMINATING SET

For the direct implication, it is easy to verify that if $K \subseteq V(G)$ is a k-multicolored clique, then the following subset of 2-track intervals is a k'-dominating set in $\mathcal{F}$:

$$\{\widehat{u_i} \mid u \in K,\, i = \kappa(u)\} \cup \{\widehat{u_i v_j}_{\text{ left}}, \widehat{u_i v_j}_{\text{ right}} \mid u, v \in K,\, i = \kappa(u),\, j = \kappa(v)\}.$$

For the reverse implication, suppose that $\mathcal{I}$ is a k'-dominating set in $\mathcal{F}$. Note that any one of the (original) two dummy 2-track intervals in each vertex or edge gadget can be replaced by an adjacent vertex or edge 2-interval in a dominating set. Thus we can assume without loss of generality that $\mathcal{I}$ includes none of these $2k + 2\binom{k}{2}$ dummies. Then, to dominate these dummies, $\mathcal{I}$ must include at least one vertex 2-track interval for each color, and at least two edge 2-track intervals for each pair of distinct colors. Since $k' = k + 2\binom{k}{2}$, $\mathcal{I}$ must include exactly one vertex 2-track interval for each color, and exactly two edge 2-track intervals for each pair of distinct colors. Refer back to Figure 4. Let $\widehat{u_i v_j}_{\text{ left}}$ and $\widehat{x_i y_j}_{\text{ right}}$ be the two edge 2-track intervals in $\mathcal{I}$ for some pair of colors ij. The intersection pattern of the edge 2-track intervals for E_{ij} on track 1 ensures that, in order to dominate all the (new) dummies in the middle column, the row of xy must not be higher than the row of uv. Without loss of generality, we can assume that they are in the same row, i.e., $uv = xy$, so that the set of validation intervals in the middle column on track 2 that are dominated by $\widehat{u_i v_j}_{\text{ left}}\, \widehat{x_i y_j}_{\text{ right}}$ is maximal. Thus the two edge 2-track intervals for each pair of colors ij form a pair $\widehat{u_i v_j}_{\text{ left}}\, \widehat{u_i v_j}_{\text{ right}}$ for the same edge uv. Then the same argument as before shows that the k vertex 2-track intervals in $\mathcal{I}$ corresponds to a k-multicolored clique in G.

4 k-Clique

In this section we prove Theorem 2. Fellows et al. [9] presented the following algorithm CLIQUE(G, k) that decides whether a given t-interval graph G has a k-clique:

CLIQUE(G, k):

1. If $|V(G)| < k$, then return NO.
2. Let v be a vertex of minimum degree in G.
3. If $\deg(v) \geq 2tk$, then return YES.
4. If v is in a k-clique of G, then return YES.
5. Return CLIQUE$(G - v, k)$.

The crucial step of this algorithm, step 3, is justified by a structural lemma [9, Lemma 2]: "if G is a t-interval graph with no k-cliques then G has a vertex of degree less than $2tk$." Step 4 can be implemented in $O(k^2 \cdot \binom{2tk}{k})$ time by brute force; all other steps have running time polynomial in n. Since the total number of recursive calls, in step 5, is at most n, the overall time complexity of the algorithm is $O(k^2 \cdot \binom{2tk}{k}) \cdot \text{poly}(n)$. Fellows et al. [9] estimated that

$$O(k^2 \cdot \binom{2tk}{k}) = t^{O(k \log k)}, \tag{1}$$

and asked whether this parametric time-bound can be improved.

Fellows et al. [9] suggested that "a possible good place to start is to consider the problem for constant values of t, and to attempt to obtain a parametric time-bound of $2^{O(k)}$." This suggestion is little misleading because for constant values of t, the algorithm CLIQUE(G, k) *already* attains a parametric time-bound of $2^{O(k)}$. Note that $\binom{2tk}{k} \leq 2^{2tk}$. Thus if $t = O(1)$ then $O(k^2 \cdot \binom{2tk}{k}) = O(2^{2\log k} \cdot 2^{2tk}) = 2^{O(k)}$.

Anyway, can we improve the parametric time-bound of $t^{O(k \log k)}$? We next describe such an FPT algorithm. Our FPT algorithm has two components. The first component is the following algorithm CLIQUE$^*(G, k)$ slightly modified from CLIQUE(G, k):

CLIQUE$^*(G, k)$:

1. If $|V(G)| < k$, then return NO.
2. Let v be a vertex of minimum degree in G.
3. If $\deg(v) \geq 2tk$, then return YES.
4. If CLIQUE$^*(\text{neighbors}(v), k-1)$ returns YES, then return YES.
5. Return CLIQUE$^*(G - v, k)$.

Note that CLIQUE$^*(G, k)$ is identical to CLIQUE(G, k) except step 4. The following recurrence on the time bound $f(k) \cdot g(n)$ captures the recursive behavior of CLIQUE$^*(G, k)$:

$$f(k) \cdot g(n) \leq f(k-1) \cdot g(2tk) + f(k) \cdot g(n-1) + O(n^2).$$

Lemma 1. *For any constant $c \geq 3$, if $k \leq \frac{1}{4} \cdot n^{1-1/c}$, then the running time of* CLIQUE$^*(G, k)$ *is $O(t^{ck}) \cdot O(n^c)$.*

The second component of our FPT algorithm is the obvious brute-force algorithm that enumerates and checks all k-subsets of vertices for k-cliques.

Lemma 2. *For any constant $c \geq 3$, if $k > \frac{1}{4} \cdot n^{1-1/c}$, then the running time of the brute-force algorithm is $2^{O(k \log k)}$.*

Finally, for any constant $c \geq 3$, by choosing the algorithm CLIQUE$^*(G, k)$ when $k \leq \frac{1}{4} \cdot n^{1-1/c}$, and choosing the brute-force algorithm when $k > \frac{1}{4} \cdot n^{1-1/c}$, we obtain an FPT algorithm with a parametric time-bound of

$$\max\{t^{O(k)}, 2^{O(k \log k)}\}. \tag{2}$$

Compare our bound (2) with the previous bound (1). It appears that we have obtained an improvement[3], but asymptotically this improvement is negligible. Check that the estimate in (1) is not tight:

$$O(k^2 \cdot \binom{2tk}{k}) = O(k^2(2tk)^k) = t^{O(k)}2^{O(k \log k)}$$
$$= \max\{(t^{O(k)})^2, (2^{O(k \log k)})^2\} = \max\{t^{O(k)}, 2^{O(k \log k)}\}.$$

In light of this delicate distinction, perhaps the open question on k-CLIQUE in t-interval graphs [9] could be stated more precisely as follows:

[3] Under the condition that $k \leq \frac{1}{4} \cdot n^{1-1/c}$ for some constant $c \geq 3$, CLIQUE$^*(G, k)$ clearly improves CLIQUE(G, k): in particular, for $t = \Theta(\log k)$, the parametric bound of CLIQUE$^*(G, k)$ is $2^{O(k \log \log k)}$, and the parametric bound of CLIQUE(G, k) is $2^{O(k \log k)}$.

Question 1. Is there an FPT algorithm for k-CLIQUE in t-interval graphs with a parametric time-bound of $t^{O(k)}$?

Note that a parametric time-bound of $2^{O(k \log k)}$ alone is beyond reach. This is because every graph of n vertices is a t-interval graph for $t \geq n/4$ [11]. If the parameter t does not appear in the bound, then we would have an FPT algorithm for the W[1]-hard problem of k-CLIQUE in general graphs.

5 Maximal Strip Recovery

In this section we prove Theorem 3. Let ℓ-MSR-d be the problem MSR-d parameterized by the total length ℓ of the strips in the optimal solution. We first prove that ℓ-MSR-4 is W[1]-hard by an FPT-reduction from k-INDEPENDENT SET in 2-track interval graphs.

Let $(\mathcal{F}, k)$ be an instance of k-INDEPENDENT SET in 2-track interval graphs, where $\mathcal{F} = \{I_1, \ldots, I_n\}$ is a set of n 2-track intervals. We construct four genomic maps $G_\rightarrow, G_\leftarrow, G_1, G_2$, where each map is a permutation of $2n$ distinct markers $\overset{i}{\subset}$ and $\overset{i}{\supset}$, $1 \leq i \leq n$, all in positive orientation. $G_\rightarrow$ and $G_\leftarrow$ are concatenations of the n pairs of markers with ascending and descending indices, respectively:

$$G_\rightarrow : \overset{1}{\subset}\ \overset{1}{\supset} \quad \cdots \quad \overset{n}{\subset}\ \overset{n}{\supset}$$
$$G_\leftarrow : \overset{n}{\subset}\ \overset{n}{\supset} \quad \cdots \quad \overset{1}{\subset}\ \overset{1}{\supset}$$

To construct G_1 and G_2, we first modify the representation of the 2-track interval graph for $\mathcal{F}$ until the $2n$ endpoints of the n intervals on each track are all distinct. This can be done in polynomial time by a standard procedure for interval graphs. Then, on each track, mark the left and the right endpoints of the interval for I_i by the left and the right markers $\overset{i}{\subset}$ and $\overset{i}{\supset}$, respectively. Thus we obtain two sequences of markers for the two genomic maps G_1 and G_2. This completes the construction.

Now set the parameter $\ell = 2k$. By the following two observations, it is easy to check that $\mathcal{F}$ has a k-independent set if and only $G_\rightarrow, G_\leftarrow, G_1, G_2$ have four subsequences of total strip length ℓ:

1. $G_\rightarrow$ and $G_\leftarrow$ ensure that each strip must be a pair of markers.
2. G_1 and G_2 encode the intersection pattern of the 2-track intervals.

Therefore ℓ-MSR-d is W[1]-hard.

Since the length of each strip is exactly 2 in our construction, the total number of adjacencies in the strips and the number of strips are both equal to half the total strip length. Therefore MSR-d remains W[1]-hard when the parameter is changed to either the total number of adjacencies in the strips or the number of strips. For any two constants d and d' such that $d' > d \geq 2$, the problem MSR-d is a special case of the problem MSR-d' with $d' - d$ redundant genomic maps. Thus the W[1]-hardness of MSR-4 implies the W[1]-hardness of MSR-d for all constants $d \geq 4$.

References

1. Alcón, L., Cerioli, M.R., de Figueiredo, C.M.H., Gutierrez, M., Meidanis, J.: Tree loop graphs. Discrete Applied Mathematics 155, 686–694 (2007)
2. Bafna, V., Narayanan, B., Ravi, R.: Nonoverlapping local alignments (weighted independent sets of axis-parallel rectangles). Discrete Applied Mathematics 71, 41–53 (1996)
3. Bar-Yehuda, R., Halldórsson, M.M., Naor, J(S.), Shachnai, H., Shapira, I.: Scheduling split intervals. SIAM Journal on Computing 36, 1–15 (2006)
4. Bulteau, L., Fertin, G., Rusu, I.: Maximal strip recovery problem with gaps: hardness and approximation algorithms. In: Dong, Y., Du, D.-Z., Ibarra, O. (eds.) ISAAC 2009. LNCS, vol. 5878, pp. 710–719. Springer, Heidelberg (2009)
5. Butman, A., Hermelin, D., Lewenstein, M., Rawitz, D.: Optimization problems in multiple-interval graphs. In: Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2007), pp. 268–277 (2007)
6. Chen, Z., Fu, B., Jiang, M., Zhu, B.: On recovering syntenic blocks from comparative maps. Journal of Combinatorial Optimization 18, 307–318 (2009)
7. Crochemore, M., Hermelin, D., Landau, G.M., Rawitz, D., Vialette, S.: Approximating the 2-interval pattern problem. Theoretical Computer Science 395, 283–297 (2008)
8. Downey, R.G., Fellows, M.R.: Parameterized Complexity. Springer, Heidelberg (1998)
9. Fellows, M.R., Hermelin, D., Rosamond, F., Vialette, S.: On the parameterized complexity of multiple-interval graph problems. Theoretical Computer Science 410, 53–61 (2009)
10. Gambette, P., Vialette, S.: On restrictions of balanced 2-interval graphs. In: Brandstädt, A., Kratsch, D., Müller, H. (eds.) WG 2007. LNCS, vol. 4769, pp. 55–65. Springer, Heidelberg (2007)
11. Griggs, J.R.: Extremal values of the interval number of a graph, II. Discrete Mathematics 28, 37–47 (1979)
12. Griggs, J.R., West, D.B.: Extremal values of the interval number of a graph. SIAM Journal on Algebraic and Discrete Methods 1, 1–7 (1980)
13. Gyárfás, A., West, D.B.: Multitrack interval graphs. Congressus Numerantium 109, 109–116 (1995)
14. Jiang, M.: Approximation algorithms for predicting RNA secondary structures with arbitrary pseudoknots. IEEE/ACM Transactions on Computational Biology and Bioinformatics, doi:10.1109/TCBB.2008.109 (to appear)
15. Jiang, M.: Inapproximability of maximal strip recovery. In: Dong, Y., Du, D.-Z., Ibarra, O. (eds.) ISAAC 2009. LNCS, vol. 5878, pp. 616–625. Springer, Heidelberg (2009)
16. Jiang, M.: Inapproximability of maximal strip recovery: II (Submitted)
17. Joseph, D., Meidanis, J., Tiwari, P.: Determining DNA sequence similarity using maximum independent set algorithms for interval graphs. In: Nurmi, O., Ukkonen, E. (eds.) SWAT 1992. LNCS, vol. 621, pp. 326–337. Springer, Heidelberg (1992)
18. Trotter Jr., W.T., Harary, F.: On double and multiple interval graphs. Journal of Graph Theory 3, 205–211 (1979)
19. Vialette, S.: On the computational complexity of 2-interval pattern matching problems. Theoretical Computer Science 312, 223–249 (2004)
20. Wang, L., Zhu, B.: On the tractability of maximal strip recovery. In: Proceedings of the 6th Annual Conference on Theory and Applications of Models of Computation (TAMC 2009), pp. 400–409 (2009)
21. West, D.B., Shmoys, D.B.: Recognizing graphs with fixed interval number is NP-complete. Discrete Applied Mathematics 8, 295–305 (1984)
22. Zheng, C., Zhu, Q., Sankoff, D.: Removing noise and ambiguities from comparative maps in rearrangement analysis. IEEE/ACM Transactions on Computational Biology and Bioinformatics 4, 515–522 (2007)

Succinct Representations of Separable Graphs

Guy E. Blelloch[1] and Arash Farzan[2]

[1] Computer Science Department, Carnegie Mellon University
[2] Max-Planck-Institut für Informatik, 66123 Saarbrücken, Germany
blelloch@cs.cmu.edu, afarzan@mpi-inf.mpg.de

Abstract. We consider the problem of highly space-efficient representation of separable graphs while supporting queries in constant time in the RAM with logarithmic word size. In particular, we show constant-time support for adjacency, degree and neighborhood queries. For any monotone class of separable graphs, the storage requirement of the representation is optimal to within lower order terms.

Separable graphs are those that admit a $O(n^c)$-separator theorem where $c < 1$. Many graphs that arise in practice are indeed separable. For instance, graphs with a bounded genus are separable. In particular, planar graphs (genus 0) are separable and our scheme gives the first succinct representation of planar graphs with a storage requirement that matches the information-theory minimum to within lower order terms with constant time support for the queries.

We, furthers, show that we can also modify the scheme to succinctly represent the combinatorial planar embedding of planar graphs (and hence encode planar maps).

1 Introduction

Many applications use graphs to model connectivity information and relationship between different objects. As the size of these graphs grow, the space efficiency becomes increasingly important. The structural connectivity of the Web modeled as the Web graph is an example which presently contains billions of vertices and the number is growing [1]. As a result, compact representation of such graphs for use in various algorithms has been in interest [2,3,4,5]. Planar (and almost planar) graphs which capture various structural artifacts such as road networks, form another example of graphs whose space-efficient representation is crucial due to their massive size. For all these applications, it is desirable to represent the graph compactly and be able to answer dynamic queries on the graph quickly.

A succinct representation of a combinatorial object is a compact representation of that object such that its storage requirement matches the information-theoretic space lower bound to within lower order terms, and it supports a reasonable set of queries in constant time. Succinct data structures perform under the uniform-cost word RAM-model with $\Theta(\lg n)$ word size [6][1]. Hence, the main distinction between succinct and compact representations of an object

[1] $\lg n$ denotes $\log_2 n$.

A. Amir and L. Parida (Eds.): CPM 2010, LNCS 6129, pp. 138–150, 2010.

is that unlike compact representations, the storage requirement of a succinct representation cannot be a constant factor away from the optimal and moreover, queries must perform in constant time.

Unstructured graphs are highly incompressible (see section 1.1). Fortunately however, most types of graph that arise in practice have some structural properties. A most common structural property that graphs in practice have is that they have small separators. A graph has small separators if its induced subgraphs can be partitioned into two parts of roughly the same size by removing a small number of vertices (to be defined precisely in section 2). Planar graphs (such as 2-dimensional meshes), almost planar graphs (such as road networks, distribution networks) [7,8], and most 3-dimensional meshes [9] have indeed small separators.

In this paper, we study the problem of succinct representations of separable undirected and unlabeled graphs (as defined precisely in definition 1). We present a succinct representation with a storage requirement which achieves the information-theoretic bound to within lower order terms and show constant time support for the following set of queries: adjacency queries, neighborhood queries, and degree queries. Adjacency queries on a pair of vertices x, y determines whether (x, y) is an edge. Neighborhood queries iterate through the neighbors of a given vertex x. Finally, the degree query outputs the number of incident edges to a given vertex x. A representation that supports these queries in constant time has the functionality of both an adjacency list and an adjacency matrix at the same time.

Analogous to Fredrickson's partitioning scheme for planar graphs [8], our succinct representation is based on recursive decomposition of graphs into smaller graphs. We repeatedly separate the given graph into smaller graphs to obtain small graphs of poly-logarithmic size which we refer to as by *mini-graphs*. These mini-graphs are further separated into yet smaller graphs of sub-logarithmic size which we refer to as by *micro-graphs*. Micro-graphs have small enough sizes to be catalogued and listed in a look-up table. Micro-graphs are encoded by a reference to within the look-up table. At each step that a graph is repeatedly split into two smaller subgraphs, the vertices in the separator are copied into both subgraphs. Therefore there are duplicate vertices in mini-graphs and micro-graphs. The main difficulty is to be able to represent the correspondence between duplicate vertices and the original graph vertices.

The time to construct the representation is dominated by the time needed to recursively decompose the graph into mini-graphs and micro-graphs and also by the time needed to assemble the look-up table for micro-graphs. The time for finding the separators and decomposing the graph recursively varies significantly from a family of graphs to another. For instance, there are linear time algorithms for finding separators in planar graphs and well-shaped meshes in arbitrary dimensions [7,9]. For our purposes a poly-logarithmic approximation of the separator size suffices and therefore we use Leighton-Rao's polynomial time construction [10]. The time required to assemble the look-up table depends on the maximum size of micro-trees, since we need to exhaustively list all separable

graphs modulo their isomorphism up to that size. We have a large degree of freedom on the choice of maximum size of a micro-graph, choice of $\sqrt{\frac{\lg n}{\lg \lg n}}$ as the maximum micro-graph size ensures a sub-linear look-up table construction time. Albeit, for simplicity of presentation of this paper, we will use $\frac{\lg n}{\lg \lg n}$ as the maximum micro-graph size.

1.1 Related Work

As mentioned previously, unstructured graphs are highly incompressible. A simple counting argument shows that a random graph with n vertices and m edges requires $\left\lceil \lg \binom{\binom{n}{2}}{m} \right\rceil$ bits. Blandford *et al.* [11] achieves this bound within a constant multiplicative factor for sparse graphs. Raman *et al.* [12] give a representation with a storage requirement which is roughly twice the information theory minimum and supports adjacency and neighborhood queries in constant time. Farzan and Munro [13] prove the infeasibility of achieving the information-theoretic space lower bound to within lower order terms and constant-time query support. and give a representation with a storage requirement that is a factor of $1 + \epsilon$ away from the minimum (for any constant $\epsilon > 0$).

Hence, space efficient representations of graphs with a certain combinatorial structure has been of interest: *e.g.* bounded-genus graphs [14], graphs with limited arboricity, and c-decomposable graphs [15]. A strong line of research has been on compressing planar graphs. Given a planar graph with n vertices, Turán [16] gives a $O(n)$-bit representation. Keeler and Westbrook [17] improve the space by a constant factor. He *et al.* [18] improve the first order term of space to the information-theory minimum. However, none of these consider fast support for queries.

Jacobson [19] gives a linear-space representation for planar graphs which supports adjacency queries in logarithmic time. Munro and Raman [20] gives a linear-space encoding for planar graphs in which supports queries in constant time. Chuang *et al.* [21] and subsequently Chiang *et al.* [22] improve the constant on the high order term for space. There is a vast literature on encoding subfamilies of planar graphs. Two important subfamilies are tri-connected planar graphs and triangulated planar graphs for which in a culminating work Castelli Aleardi *et al.* [23] show a succinct representation. This representation, used for general planar graphs, has a storage requirement which is a constant factor away from the optimal (and therefore is not succinct).

One important aspect in representing planar graphs has been to also represent the associated planar embedding together with the graph (*i.e.* to represent planar maps). We demonstrate our scheme yields a succinct representation for both general planar graphs and planar maps.

Blandford *et al.* [11] study space-efficient representations of separable graphs with constant time support for adjacency, degree, and neighborhood queries. However their representation is not succinct and can have a storage which is a multiplicative factor away from the optimal. We present a succinct representation for separable graphs that supports the same set of queries in constant time.

2 Preliminaries

A *separator* S in a graph $G = (V, E)$ with n vertices is a set of vertices that divides V into non-empty parts $A \subset V$ and $B \subset V$ such that $\{A, S, B\}$ is a partition of V, and no edge in G joins a vertex in A to a vertex in B.

Definition 1. *A family of graphs $\mathcal{G}$ that is closed under taking the vertex-induced subgraphs satisfies the $f(.)$-*separator theorem *[7] if there are constants $\alpha < 1$ and $\beta > 0$ such that each member graph $G \in \mathcal{G}$ with n vertices has a separator S of size $|S| < \beta f(n)$ which divides the vertices into parts A, B each of which contains at most αn vertices ($|A| \leq \alpha n, |B| \leq \alpha n$). We define a family of graphs as* separable *if it satisfies the n^c-separator theorem for some constant $c < 1$. A graph is separable if it belongs to a separable family of graphs.*

Lipton, Rose, and Tarjan [24] prove that a in family of graphs satisfying a $\left(n/(\log n)^{1+\epsilon}\right)$-separator theorem for some $\epsilon > 0$, the number of edges of a graph is linear in the number of vertices. Since separable graphs satisfy a stronger separator theorem, a separable graph has linear number of edges.

We use the dictionary data structures heavily in this work. The first data structure we need in our tool set is an indexable dictionary (ID) to represent a subset of a universe supporting membership, rank, and select queries on member elements in constant time. A membership query on a given element X determines whether x is present in the subset. A rank query on an element x reports the number of present elements less than x in the subset. Finally, a select query (which are reverse to rank queries) for a given number i reports element at rank i in the increasing order in the subset.

Lemma 1 ([12]). *Given a set S of size s which is a subset of a universe $U = \{1, \ldots, u\}$, there is an indexable dictionary (ID) on S that requires $\lg \binom{u}{s} + o(s) + O(\log \log u)$ bits and supports rank/select on elements of S in constant time (rank/select on non-members is not supported).*

Unlike IDs, *fully indexable dictionaries* (FID) support membership, rank, and select queries on both members and non-members. These are very powerful structures, as they can support predecessor queries in constant time. As a result, they are not as space-efficient as IDs.

Lemma 2 ([12]). *Given a subset S of a universe U, there is a* fully indexable dictionary **(FID)** *structure which requires $\lg \binom{|U|}{|S|} + O(|U| \log \log |U| / \log |U|)$ bits and supports rank and select queries both on members and nonmembers of S in constant time.*

3 Succinct Representation

Analogous to the compact representation of separable graphs [11], we find and remove separators recursively to decompose the graph. Given a graph G with n vertices, we find a small separator S ($|S| < \beta n^c$) whose removal divides G

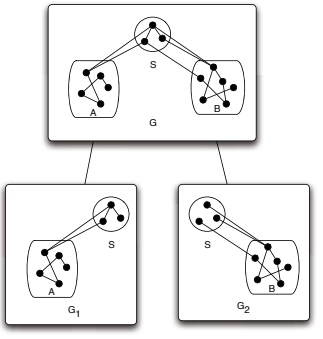


Fig. 1. Decomposition of a separable graph G into G_1, G_2

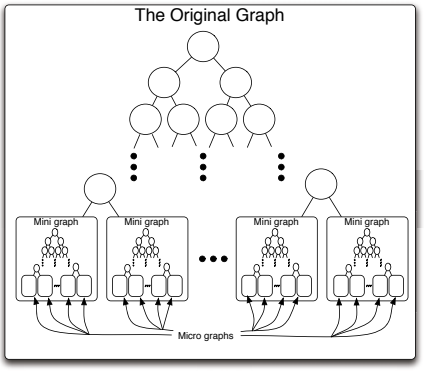


Fig. 2. A schematic view of the decomposition of a separable graph to mini-graphs and then to micro-graphs

into two parts A, B with at most αn vertices each. We obtain two induced subgraphs $G_1 = A \cup S$ and $G_2 = B \cup S$. We remove internal edges of S from G_1 (and retain them in G_2). Therefore, we obtain two subgraphs G_1, G_2 from G. Figure 1 illustrates the decomposition of an example graph G into G_1 and G_2.

We decompose G_1 and G_2 to obtain smaller subgraphs. Smaller subgraphs are in turn decomposed similarly into yet smaller subgraphs. We define a constant $\delta = 2/(1-c)$ where there are n^c-separators (definition 1). We repeat the separator-based decomposition till the subgraphs have at most $(\lg n)^\delta$ vertices where n is the number of vertices in the initial graph. We refer to these subgraphs with at most $(\lg n)^\delta$ vertices as *mini-graphs*.

Mini-graphs are further decomposed in the same fashion. Each mini-graph is decomposed repeatedly until the number of vertices in subgraphs is at most $\lg n / \lg \lg n$. We refer to these subgraphs with at most $\lg n / \lg \lg n$ vertices as *micro-graphs*. Figure 2 illustrates the decomposition into mini and micro graphs.

The graph representation consists of the representations of mini-graphs which in turn consist of the representations of micro-graphs. Micro-graphs are small enough to be catalogued by a look-up table. Vertices in separators are duplicated by each iteration of the decomposition and therefore there can be many occurrences of a single vertex of the original graph across different mini-graphs and/or micro-graphs.

Each occurrence of a vertex receives three labels: a label within the containing micro-graph which we refer to as by *micro-graph label*, a label within the containing mini-graph which we refer to as by *mini-graph label*, and finally a label in the entire graph which we refer to as by *graph label* and is visible from outside our abstract data type for the graph. Queries indicate vertices using their graph labels. Dictionary structures of lemmas 1 and 2 are used to maintain the relationship between duplicates of a single vertex.

Combining representations of mini-graphs. We assume mini-graphs are encoded (using a scheme to be discussed shortly), we explain here how these encodings can be combined to represent the entire graph. We start by bounding the number of vertices of an individual mini-graph and their accumulative size (proof omitted due to space constrainsts):

Lemma 3. *The number of mini-graphs is* $\Theta\left(n/(\log n)^{\delta}\right)$. *The total number of duplicates among mini-graphs (*i.e. *sum of multiplicities greater than one) is* $O\left(n/\log^2 n\right)$. *The sum of the number of vertices of mini-graphs together is* $n + O\left(n/\log^2 n\right)$. □

As discussed previously the given graph is unlabeled and we pick (graph) labels for vertices. Labels of vertices with no duplicates precede labels of vertices with duplicates. Mini-graphs are scanned in order and non-duplicate vertices are assigned graph labels consecutively. Duplicate vertices in the original graph are assigned graph labels arbitrarily using the remaining labels.

To translate graph labels to/from mini-graph labels, we build a bit vector `Translate` with length equal to the sum of the number of vertices in mini-graphs. This vector spans across all mini-graphs in order containing an entry for each vertex of a mini-graph. The entry is set to zero if the corresponding vertex has no duplicates and is set to one if it has a duplicate. The fully indexable dictionary (FID) of lemma 2 is used to represent one entries over the universe of all entries in `Translate`. Support for rank and select on both zeros and ones allows us to translate between locations in `Translate` and graph labels. The space of this structure by lemmas 2,3 is $o(n)$. Figure 3 depicts an overview of these structures.

`Boundaries` is another bit vector which is encoded also using a FID. It marks the boundaries of mini-graphs in `Translate`. `Translate` and `Boundaries` together enable us to translate labels of non-duplicate vertices. Given the graph label of such a vertex, we find the corresponding location in `Translate` by a select query and then perform rank on `Boundary` to obtain the mini-graph number and the offset from the predecessor one which is the mini-graph label of that vertex. Conversely, given the mini-graph label of a non-duplicate vertex, we perform select on `boundaries` to find the start location of the mini-graph in `Translate` and add to it the mini-graph label to find the corresponding location in there. Now a rank over non-duplicates gives us the graph label.

For translating labels of duplicate vertices, we maintain other structures. `Duplicates` has a list for each duplicate vertex which contains all duplicates of the vertex as positions in `Translate`. `Duplicates` empowers us to iterate through duplicates of a vertex. `Real-names` is an array with length equal to the sum of multiplicities of duplicates vertices. Its entries contain in order the graph label of each occurrence of a duplicate vertex in `Translate`. `Real-names` allows us to determine the graph label of an occurrence of a duplicate vertex. Using these structures we can translate between graph labels and mini-graph labels of duplicate vertices. To account for the space of these structures, we note that $\Theta(\log n)$ bits are used for any occurrence of duplicate vertices of which there are $\Theta\left(n/\log^2 n\right)$ by lemma 3, and therefore the space is $\Theta(n/\log n)$ bits.

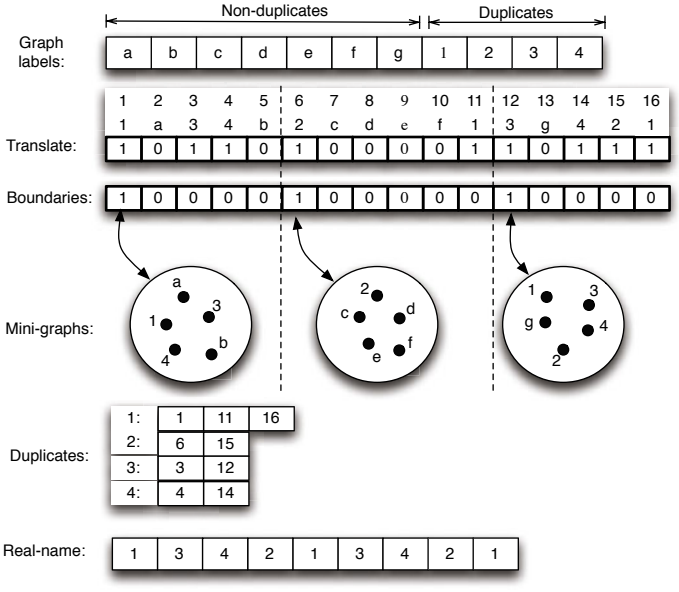


Fig. 3. Indexing structures used to translate between graph labels and mini-graph labels

Combining representations of micro-graphs. The representation of a mini-graph is composed of those of micro-graphs in the same manner as the representation of the entire graph is composed out of mini-graphs. The same set of structures are built and stored. The technical lemma in this construction is analogous to lemma 3. The details of this construction and the proof of lemma is omitted due to space constraints.

Lemma 4. *Within a particular mini-graph of m vertices, the number of micro-graphs is $\Theta((m \log\log n)/\log n)$. The total number of duplicates (*i.e. *sum of multiplicities) is $O\left((m \log\log^{1-c} n)/\log^{1-c} n\right)$. Sum of the number of vertices of micro-graphs together is $m + O\left((m \log\log^{1-c} n)/\log^{1-c} n\right)$.* □

Representations of micro-graphs. Micro-graphs have $\Theta(\log n/\log\log n)$ vertices and are encoded by an `Index` to a look-up table. The look-up table lists all possible micro-graphs with $\Theta(\log n/\log\log n)$ vertices ordered according to their numbers of vertices. The table also stores pre-computed answers to all queries of interest.

`Index` fields account for the dominant space term. Since we enumerate micro-graphs to list them in the look-up table, the length of the `Index` field matches the entropy bound for each micro-tree. Since a family of separable graphs has a linear entropy ($\mathcal{H}() = O(n)$) [11], the sum of the lengths of `Index` fields over all micro-graphs is $\mathcal{H}(\Sigma) + o(n)$ where Σ is the sum of the number of vertices of micro-graphs ($o(n)$ comes from the round-up for individual indices). Lemmas 3 and 4

show that $\Sigma = n + o(n)$ and thus the length of the encoding is $\mathcal{H}(n) + o(n)$. Since all indexes built to combine micro-graphs into mini-graphs and combine mini-graphs into an entire graph is $o(n)$ as shown, and the storage requirement of the look-up table is $o(n)$, the entire representation requires $\mathcal{H}(n) + o(n)$ bits.

We now turn to showing support for queries in constant time. The two main queries of interest are neighborhood and adjacency queries and support for degree queries is straightforward.

3.1 Neighborhood Queries

We now explain how neighbors of a vertex can be reported in constant time per neighbor. Given a vertex v by its graph label, we first determine if it has duplicates by a simple comparison. If there is no duplicates then the corresponding mini-graph and the mini-graph label are determined. If there are duplicates, we use `Duplicates` array to look-up each occurrence of the vertex one by one. Each occurrence leads us to a particular vertex in a mini-graph.

Once confined to a mini-graph and a particular vertex u therein, we determine analogously if u has duplicates across micro-graphs. If no duplicate exists, then we find the micro-graph and the micro-graph label therein and the query is answered using the pre-computed neighbors in the look-up table. In case duplicates exist, array `Duplicates` is used and each occurrence is handled analogously.

Each neighbor vertex name is a micro-graph label and should be translated to a graph label which is performed by a conversion to mini-graph label and subsequently to a graph label using `Translate, Boundaries` structures.

3.2 Adjacency Queries

We use the same approach as in [11] and direct the edges such that in the resulting graph each vertex has a bounded out-degree:

Lemma 5 ([11]). *The edges of a separable graph can be directed in linear time such that each vertex has out-degree at most b for some constant $b > 0$.*

In order to answer the adjacency query $q(u, v)$, it suffices to show how outgoing edges of a vertex can be looked-up in constant time as the (possible) edge between u, v is either directed from u to v or vice versa.

We cannot store the directed graph as the space requirement would exceed our desirable bound. We only store the direction of a sub-linear number of edges. The look-up table remains undirected and intact, and thus it does not reflect the direction of any edge.

We add the following structures to enable constant time look-up for out-going edges. In a mini-graph, for each vertex v with duplicates, we store b vertices that are endpoints of edges going out of v ($\Theta(\log \log n)$ bits each). Similarly, in the entire graph, for each vertex u with duplicates we explicitly store b endpoints of edges going out of u ($\Theta(\log n)$ bits each).

More importantly, for each vertex with duplicates across different mini-graphs, we store, in Structure `Duplicate-components`, the mini-graph numbers in which

it has a duplicate . We cannot simply list mini-graphs in `Duplicate-components` as we must support membership queries. We use the indexable dictionary structure (lemma 1) over the universe of mini-graphs. Internal to each mini-graph, we build the same structure as `Duplicate components` which captures the micro-graph numbers of duplicates of the same vertex across different micro-graphs. The extra space added by using these structures can be proved to be $o(n)$.

Given a query $q(u, v)$ on two vertices u, v. We state the procedure for vertex u, however the same procedure must be repeated for vertex v afterwards. We first determine if u has duplicates in different mini-graphs or micro-graphs. If it does so, then endpoints of its outgoing edges are explicitly listed which we compare against v by translating mini-graph and/or micro-graph labels of the listed vertices. In case u has duplicates neither across micro-graphs within the mini-graph nor across different mini-graphs, u appears in only one mini-graph and one micro-graph therein. For v to have and edge to u, it must appear in the same micro and mini-graph. We use structure `Duplicate-components` to determine if v has a duplicate in the same micro-graph as u. As otherwise, there cannot be an edge uv. We now use a rank query in `Duplicate-components` to index to `Duplicates` and retrieve the micro-graph label of the proper duplicate of v. Within a micro-graph, we use the look-up table, to determine if they are adjacent in constant time.

Theorem 1. *Any family of separable graphs (definition 1) with entropy $\mathcal{H}(n)$ where n is the number of vertices, can be succinctly encoded in $\mathcal{H}(n) + o(n)$ bits such that adjacency, neighborhood, and degree queries are supported in constant time.* □

4 Representing Planar Maps: Encoding the Combinatorial Embedding

A planar drawing of a planar graph is a drawing of the graph in $\mathbb{R}^2$ with no edge crossings. There is infinitely many planar drawings for any fixed planar graphs G. Two such planar drawings are *equivalent* if for all vertices the clockwise cyclic ordering of neighbors is the same in both graphs. An equivalency class of planar drawings specifies a clockwise cyclic order of neighbors for all vertices which is known as the *combinatorial planar embedding*. A *planar map* is a planar graph together with a fixed combinatorial planar embedding.

In this section, we address the issue of representing (unlabeled) planar maps. The underlying planar graphs of a planar map is separable and therefore the representation of section 3 can encode them succinctly to support adjacency, degree, and neighborhood queries in constant time. In planar maps representations, we not only need to encode the planar graph, but also we need to store the combinatorial planar embedding. Hence, we enhance the definition of neighborhood queries to report neighbors of a fixed vertex according to the combinatorial planar embedding: *i.e.* neighbors should be reported in the clockwise cyclic order in constant time per neighbor.

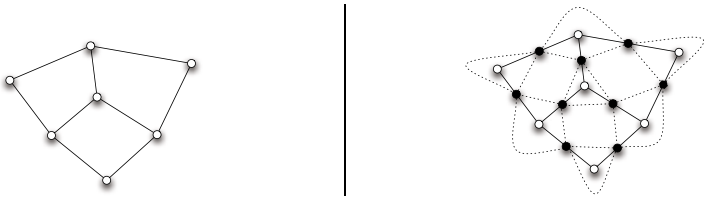


Fig. 4. A planar map (left) and the resulting graph where edges are subdivided and connected according to the combinatorial planar embedding (right)

We first note that we can easily achieve a planar map encoding by increasing the storage requirement by a constant factor. Given a planar map G, we subdivide all edges by introducing a dummy vertex of degree two on each edge and connect these dummy vertices circularly around each vertex (as depicted in figure 4. Since the number of edges of a planar graph is linear, the number of vertices is increased by a constant factor. It is easy to verify that the resulting graph is planar and therefore separable. We can encode this graph using any of the compact planar graph representations referred to in section 1.1 using $O(n)$ bits. Using the dummy vertices, we can produce neighbors of a vertex in the circular order according to the combinatorial embedding. Moreover, we explicitly store a bit for each dummy node which distinguishes the immediate clockwise and counter-clockwise neighbor (*e.g.* we set the bit to zero if the neighbor with a higher label is the clockwise one). Using these bits we can produce the neighbors in the actual clockwise circular order for any fixed node. This encoding proves that the entropy $\mathcal{H}_p(n)$ of planar maps is linear in the number of vertices n.

Although the simple encoding scheme achieves the entropy to within a constant factor, a succinct representation that achieves the entropy tightly to within lower order terms is desired and we will give such representation in this section.

Theorem 2. *A planar map G with n vertices can be encoded succinctly in $\mathcal{H}_p(n) + o(n)$ bits where n is the number of vertices of G. The encoding supports queries adjacency queries, degree queries, and neighborhood queries (according to combinatorial planar embedding of G) in constant time.*

We subdivide edges of G by introducing dummy vertices of degree two on each edge as described before to obtain graph G'. Since G' is planar and separable, we use the succinct separable graph representation of section 3 to represent it. This representation in its current form requires a space which is a constant factor away from entropy $\mathcal{H}_p(n)$. We will make modifications to lessen the space to $\mathcal{H}_p(n) + o(n)$. We will also show constant-time support for queries.

The succinct separable representation of section 3 divides G into mini-graphs and micro-graphs and creates duplicate vertices which are repeated in more than one mini/micro-graphs. Among dummy vertices, we retain all that are duplicates and discard all that are not. A non-duplicate dummy vertex d is discarded by a contraction which deletes the vertex and connects the endpoints of the edge d stood for. We refer to by the resulting graph as $\hat{G}$. By lemmas 3, and, 4 the

total number of dummy vertices that are retained is $o(n)$ and therefore the total number of vertices in the graph is $n + o(n)$. Using a bit vector which is stored as in lemma 2, we explicitly store whether a vertex is a dummy vertex.

Micro-graphs are stored by references into a look-up table as before. The micro-graph is a subgraph of G' and therefore is planar. Furthermore, the combinatorial planar embedding of G' induces a combinatorial planar embedding for micro-graphs. The table stores the combinatorial planar embedding of micro-graphs together with the structure of the graphs. Theorem 1 implies that the storage requirement of the representation is $\mathcal{H}_p(n) + o(n)$ bits.

It only remains to show constant-time support for queries. As the degrees of original vertices in G remain unchanged supporting *degree queries* is trivial. Support for *adjacency queries* is more complicated since we have introduced dummy vertices on edges of G. Nevertheless, the adjacency queries in G are handled in the same manner as adjacency queries in the representation (section 3.2). To show support for adjacency queries in section 3.2, we first oriented the edges of the graph such that each vertex has a bounded out-degree (lemma 5). To orient G', we orient the underlying graph G according to lemma 5 and if edge uv in G has a dummy vertex d in G', we orient edges of G' as $u \to d$ and $d \to v$. We orient edges between dummy vertices according to the clockwise cyclic order. It is easy to verify that all vertices have a constant out-degree in $\hat{G}$, and therefore we can repeat the same procedure as in section 3.2. However, the procedure guarantees that we can discover edges between immediate neighbors and in $\hat{G}$ there could be a dummy vertex on an edge. We first note that this is not an issue within a micro-graph as using the look-up table we can easily test if two vertices are connected through a degree-two vertex (which we must also verify to be a dummy vertex). For vertices that have a duplicate across mini/micro-graphs, we explicitly listed out-neighbors in section 3.2; here we list explicitly out-neighbors through dummy vertices as well (*i.e.* if node u is a duplicate and there are edges $u \to d \to v$ where d is a dummy vertex, we explicitly store v in the list). Response to adjacency queries can be computed as in section 3.2.

We now demonstrate how *neighborhood queries* are supported. Given an edge uv between two vertices u and v of graph G, the neighborhood query is to report the next neighbor of v in the circular order according to the combinatorial planar embedding. Let us denote by d the dummy vertex in G' that resides on edge uv of G. Also we denote by w the next neighbor of u in the circular order in G, and d' the dummy vertex that resides on edge uw in G'. Either of dummy vertices d, d' in G' may or may not be present in $\hat{G}$. Refer to figure 5.

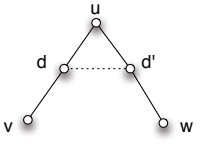


Fig. 5. Supporting neighborhood queries on vertex u: vertex w is reported after vertex v regardless of existence of d or d'

We distinguish two cases according to whether the edge dd' is present or absent in $\hat{G}$. If $dd' \in G'$ (both d and d' are present in $\hat{G}$), then clearly we can discover the next neighbor of u by taking the edge dd' and arriving at vertex d' which leads us to vertex w.

Therefore, the more interesting case is where edge dd' is absent. In this case, neither of d or d' could be a duplicate vertex (as otherwise, since we retain duplicate dummy vertices and their immediate neighbors, they both would be present and therefore edge dd' would exist). Since d and d' do not have duplicates and they are immediately connected (by edge dd'), they belong to the same micro-graph of G'. Moreover, since u, v, w are immediate neighbors of vertices d, d', these vertices or a duplicate of them must also belong to the same micro-graph. Since edges of G are not repeated in more than one micro-graph, the micro-graph is the one containing the (possibly subdivided) edge uv in $\hat{G}$. Hence, the edge uw can be read from the look-up table as the next neighbor of uv in the circular order. □

5 Conclusion and Discussion

We studied the problem of succinctly encoding separable graphs while supporting degree, adjacency, and neighborhood queries in constant time. For each family of separable graphs (*e.g.* planar graphs). The storage is the information-theoretic minimum to within lower order terms. We achieve the entropy bound for any monotone family of separable graphs with no knowledge of the actual entropy for that family of graphs since we use look-up tables for tiny graphs. Namely, when used for planar graphs, our representation requires a space which is the entropy of the planar graphs to within lower order terms while supporting queries in constant time. This is when the actual entropy (or equivalently the number of unlabeled planar graphs) is still unknown [25]. This is an improvement in the heavily-studied compact encoding of planar graphs. Moreover, we showed that our approach yields a succinct representation for planar maps (*i.e.* planar graphs together with a given embedding).

One interesting direction for future work is to extend the idea of this paper to represent dynamic separable graphs. These are graphs under updates in form of insertion and deletion of vertices and edges while the graphs remains separable.

References

1. Gulli, A., Signorini, A.: The indexable web is more than 11.5 billion pages. In: WWW 2005: Special interest tracks and posters of the 14th international conference on World Wide Web, pp. 902–903. ACM, New York (2005)
2. Claude, F., Navarro, G.: A fast and compact web graph representation. In: Ziviani, N., Baeza-Yates, R. (eds.) SPIRE 2007. LNCS, vol. 4726, pp. 118–129. Springer, Heidelberg (2007)
3. Broder, A., Kumar, R., Maghoul, F., Raghavan, P., Rajagopalan, S., Stata, R., Tomkins, A., Wiener, J.: Graph structure in the web. Comput. Netw. 33(1-6), 309–320 (2000)
4. Adler, M., Mitzenmacher, M.: Towards compressing web graphs. In: DCC 2001: Proceedings of the Data Compression Conference, Washington, DC, USA, p. 203. IEEE Computer Society, Los Alamitos (2001)
5. Suel, T., Yuan, J.: Compressing the graph structure of the web. In: DCC 2001: Data Compression Conference, p. 213. IEEE, Los Alamitos (2001)

6. Munro, J.I.: Succinct data structures. Electronic Notes in Theoretical Computer Science 91, 3 (2004)
7. Lipton, R.J., Tarjan, R.E.: A separator theorem for planar graphs. SIAM Journal on Applied Mathematics 36(2), 177–189 (1979)
8. Frederickson, G.N.: Fast algorithms for shortest paths in planar graphs, with applications. SIAM J. Comput. 16(6), 1004–1022 (1987)
9. Miller, G.L., Teng, S.H., Thurston, W., Vavasis, S.A.: Separators for sphere-packings and nearest neighbor graphs. J. ACM 44(1), 1–29 (1997)
10. Leighton, T., Rao, S.: An approximate max-flow min-cut theorem for uniform multicommodity flow problems with applications to approximation algorithms. In: FOCS 1988: Foundations of Computer Science, pp. 422–431. IEEE, Los Alamitos (1988)
11. Blandford, D.K., Blelloch, G.E., Kash, I.A.: Compact representations of separable graphs. In: SODA: ACM-SIAM Symposium on Discrete Algorithms (2003)
12. Raman, R., Raman, V., Satti, S.R.: Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. ACM Trans. Algorithms 3(4), 43 (2007)
13. Farzan, A., Munro, J.I.: Succinct representations of arbitrary graphs. In: Halperin, D., Mehlhorn, K. (eds.) ESA 2008. LNCS, vol. 5193, pp. 393–404. Springer, Heidelberg (2008)
14. Lu, H.I.: Linear-time compression of bounded-genus graphs into information-theoretically optimal number of bits. In: SODA 2002: Proceedings of ACM-SIAM symposium on Discrete algorithms, pp. 223–224 (2002)
15. Kannan, S., Naor, M., Rudich, S.: Implicit representation of graphs. SIAM J. Discrete Math. 5(4), 596–603 (1992)
16. Turán, G.: On the succinct representation of graphs. Discrete Applied Mathematics 8, 289–294 (1984)
17. Keeler, W.: Short encodings of planar graphs and maps. DAMATH: Discrete Applied Mathematics and Combinatorial Operations Research and Computer Science 58 (1995)
18. He, X., Kao, M.Y., Lu, H.I.: A fast general methodology for information-theoretically optimal encodings of graphs. SIAM Journal on Computing 30(3), 838–846 (2000)
19. Jacobson, G.: Space-efficient static trees and graphs. In: 30th Annual Symposium on Foundations of Computer Science, 1989, October 30 – November 1, pp. 549–554 (1989)
20. Munro, J.I., Raman, V.: Succinct representation of balanced parentheses, static trees and planar graphs. In: IEEE Symposium on Foundations of Computer Science, pp. 118–126 (1997)
21. Chuang, R.C.N., Garg, A., He, X., Kao, M.Y., Lu, H.I.: Compact encodings of planar graphs via canonical orderings and multiple parentheses. In: Larsen, K.G., Skyum, S., Winskel, G. (eds.) ICALP 1998. LNCS, vol. 1443, pp. 118–129. Springer, Heidelberg (1998)
22. Chiang, Y.T., Lin, C.C., Lu, H.I.: Orderly spanning trees with applications to graph encoding and graph drawing. In: SODA 2001: ACM-SIAM symposium on Discrete algorithms, pp. 506–515 (2001)
23. Devillers, L.C.A.O., Schaeffer, G.: Succinct representations of planar maps. Theor. Comput. Sci. 408(2-3), 174–187 (2008)
24. Lipton, R.J., Rose, D.J., Tarjan, R.E.: Generalized nested dissection. SIAM Journal on Numerical Analysis 16, 346–358 (1979)
25. Liskovets, V.A., Walsh, T.R.: Ten steps to counting planar graphs. Congressus Numerantium 60, 269–277 (1987)

Implicit Hitting Set Problems and Multi-genome Alignment

Richard M. Karp

University of California at Berkeley and
International Computer Science Institute

Let U be a finite set and S a family of subsets of U. Define a hitting set as a subset of U that intersects every element of S. The optimal hitting set problem is: given a positive weight for each element of U, find a hitting set of minimum total weight. This problem is equivalent to the classic weighted set cover problem.We consider the optimal hitting set problem in the case where the set system S is not explicitly given, but there is an oracle that will supply members of S satisfying certain conditions; for example, we might ask the oracle for a minimum-cardinality set in S that is disjoint from a given set Q. The problems of finding a minimum feedback arc set or minimum feedback vertex set in a digraph are examples of implicit hitting set problems. Our interest is in the number of oracle queries required to find an optimal hitting set. After presenting some generic algorithms for this problem we focus on our computational experience with an implicit hitting set problem related to multi-genome alignment in genomics. This is joint work with Erick Moreno Centeno.

A. Amir and L. Parida (Eds.): CPM 2010, LNCS 6129, p. 151, 2010.

Bounds on the Minimum Mosaic of Population Sequences under Recombination

Yufeng Wu

Department of Computer Science and Engineering
University of Connecticut
Storrs, CT 06269, U.S.A.
ywu@engr.uconn.edu

Abstract. We study the minimum mosaic problem, an optimization problem originated in population genomics. We develop a new lower bound, called the C bound. The C bound is provably higher and significantly more accurate in practice than an existing bound. We show how to compute the exact C bound using integer linear programming. We also show that a weaker version of the C bound is also more accurate than the existing bound, and can be computed in polynomial time. Simulation shows that the new bounds often match the exact optimum at least for the range of data we tested. Moreover, we give an analytical upper bound for the minimum mosaic problem.

1 Introduction

Recombination is a key genetic process that creates *mosaic* population sequences during meiosis. Throughout this paper, we assume population sequences are *binary*. This is justified by the current interests in single nucleotide polymorphisms (SNPs). A SNP is a single nucleotide site where exactly two (of four) different nucleotides occur in a large percentage of the population, and thus can be represented as a binary number. The input data is a binary matrix M with n rows (sequences) and m columns (SNPs). Recombination plays an important role in the evolutionary history of these sequences. Conceptually, recombination takes two equal length sequences and generates a new sequence of same length by concatenating a prefix of one sequence and a suffix of the other sequence. The position between the prefix and the suffix is called a *breakpoint*. Studying recombination in populations needs genealogical models. In this paper, we focus on a model called the **mosaic** model. The mosaic model [10] assumes that current population sequences are descendants of a *small* number of *founder* sequences. Due to recombination, an extant sequence consists of multiple segments from the founders, where breakpoints separate the segments. We ignore point mutations in this paper. That is, extant sequences contain *exact* copies of founder segments. See Figure 1 for an illustration of the mosaic structure.

The mosaic model is a *recurring* formulation in population genomics with applications including study of recombinant inbred mouses [13], hidden Markov models (HMMs) in inferring haplotypes from genotypes [2,5,8] and genotype

A. Amir and L. Parida (Eds.): CPM 2010, LNCS 6129, pp. 152–163, 2010.

Fig. 1. A minimum mosaic found by program *RecBlock* containing 55 breakpoints for the given twenty sequences of length forty. Note that only the sequences are given as input: we do not know the founders nor the positions of the breakpoints. The top four sequences are founders inferred by program *RecBlock*, each with a distinct color. A different mosaic for these sequences with 56 breakpoints was shown in [6].

imputation methods (e.g. [3]). A main technical challenge is that even when sequences of M are formed as a mosaic, the breakpoints are *invisible*. Thus inference needs to be performed to reconstruct the mosaic pattern (and the founder sequences) from the given sequences. In [10], Ukkonen formulated an optimization problem for the mosaic model based on parsimony.

The Minimum Mosaic Problem. For a binary matrix M with n rows (sequences) and m columns as well as an integer 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 at matching positions. Such a mosaic is called the minimum mosaic for M and the number of breakpoints needed in the minimum mosaic is denoted as $B_{min}(M)$. Note that no shifting is allowed: segments of founders in the input sequences must retain their original positions in the founders.

The minimum mosaic problem can be viewed as a sequence coloring problem as shown in Figure 1. We need to assign colors (one per each founder) to the given sequences (i.e. sequences are colored by founders). The total number of color changes (i.e. breakpoints) in the minimum mosaic is the smallest among all possible coloring. Note at a column, if two sequences are assigned the same color, then they must have *identical* values at this column. Also, K_f is usually much smaller than n.

The minimum mosaic problem has a simple polynomial time algorithm for binary data [10,12] or even for the so-called genotype data (a mixed description of two binary sequences) [12] when $K_f = 2$. There are two exact methods for the general minimum mosaic problem, which are implemented in program *Haplovisual* [10] and program *RecBlock* [12]. None of these exact methods runs in

polynomial-time. Our experience suggests that program *RecBlock* outperforms program *Haplovisual* for many datasets. The basic idea of program *RecBlock* is to scan from left to right, and enumerate all possible founders at each column. For example, suppose $K_f = 3$. Then there are 6 configurations for founder settings at column 1: 001, 010, 100, 011, 101 and 110 (000 and 111 are excluded because we can preprocess the input sequences to remove any columns with only 0s or only 1s, and such preprocessing does not change the solution). Now for column 2, we need to enumerate all 6 founder configurations for each of the configurations at column 1. Full enumeration quickly becomes infeasible. To obtain a practical method, program *RecBlock* uses several techniques to prune the search space. Empirical study shows that program *RecBlock* seems to work well for small number of founders (say three to five) when the data is of medium size (say 50 by 50). However, its performance degrades as K_f and/or the size of matrix increase.

Since there exists no known polynomial-time algorithm for the minimum mosaic problem, heuristic methods for handling larger datasets are also developed [12,7]. For example, program *RecBlock* can run in a non-optimal mode to find relatively good solutions for larger datasets when exact solutions are difficult to obtain. Although these methods are fast, a *major* problem is the lack of knowledge on the optimality of their solutions. Moreover, little is known on theoretical quantification of the mosaic structure in general.

This motivates computing *lower bounds* for the minimum mosaic problem. A lower bound gives an estimate on the necessary breakpoints. Lower bounds can be useful in the following two ways.

1. A lower bound quantifies the *range* of the solution, together with the upper bounds found by heuristic methods. In the ideal case, the lower bound matches the upper bound, and thus certifies the optimality of the heuristic solution. Even when the lower bound is smaller than the upper bound, the range of solution given by the bounds can still be useful.
2. A good lower bound can speed up the branch and bound search for the optimal solution.

A known lower bound. There is an existing lower bound (called the *CH* bound) in [12] for the minimum mosaic problem, which is inspired by [4]. The *CH* bound considers each segment $[i, j]$ of M (i.e. the sub-matrix $M_{i,j}$ with columns between i and j inclusive, where $i < j$). We let set $S_{i,j}$ contain the *distinct* rows of $M_{i,j}$, and we denote the multiplicity of a sequence $s^k_{i,j} \in S_{i,j}$ in $M_{i,j}$ as $n^k_{i,j}$. Each $s^k_{i,j}$ represents a "cluster" of $n^k_{i,j}$ identical sequence within the interval. For example, in Figure 2, there are four clusters between the first and the third columns: 001, 100, 010 and 000 (which are numbered from 1 to 4 in this order). For the cluster $s^1_{1,3} = 001$, $n^1_{1,3} = 2$, while $n^k_{1,3} = 1$ for $k = 2, 3, 4$. We order sequences $s^k_{i,j}$ so that the list of $n^k_{i,j}$ is *non-increasing*. Now if $|S_{i,j}| \leq K_f$, then the lower bound on $B_{min}(M_{i,j})$ is simply 0. Otherwise, $B_{i,j} = \sum_{k=K_f+1}^{|S_{i,j}|} n^k_{i,j}$ is a lower bound on $B_{min}(M_{i,j})$. For the data in Figure 2, when $K_f = 3$, $B_{1,3} = 1$. We say a cluster is cut if *each* identical sequence of the

cluster contains breakpoints. Otherwise, the cluster is un-cut if some sequence of the cluster contains no breakpoints. The correctness of $B_{i,j}$ as a lower bound on $B_{min}(M_{i,j})$ is due to the fact that there are only K_f colors and Lemma 1.

Lemma 1. *Within an interval, two distinct sequences containing no breakpoints can not be colored by the same founder.*

$B_{i,j}$ for each $[i, j]$ is often much smaller than $B_{min}(M)$. But Myers and Griffiths [4] introduced a general method to *combine* $B_{i,j}$ from all intervals to get a much higher overall bound. Intuitively, we consider a straight horizontal line. We need to place the *minimum* number of breakpoints along the horizontal line, so that for each interval $[i, j]$, there is at least $B_{i,j}$ breakpoints that are properly contained inside $[i, j]$. The *CH* bound is equal to the minimum number of breakpoints needed. A breakpoint located at position x within $[i, j]$ contributes to the interval if $i < x < j$. Note that breakpoints are not placed at integer points. The *CH* bound is simple and efficiently computable using a greedy approach [4]. However, practical experience shows that the *CH* bound often significantly underestimates $B_{min}(M)$ (see Section 4), which greatly limits its use.

Contributions. In this paper, we present a new *lower bound* (called the C bound) for the minimum mosaic problem. We show that the C bound is provably higher than (or equal to) the *CH* bound, and often significantly higher in practice. For a large portion of datasets we simulate, the C bound matches $B_{min}(M)$. Thus, the C bound can be useful in quantifying the tighter range of optimal solution. In terms of efficiency, the C bound can be computed exactly for many datasets, and a variation of the C bound (which leads to only a little loss of accuracy) can be computed in polynomial time. We also evaluate the performance of the lower bounds in branch and bound search through simulation, where we do observe, albeit modest, speedup. On the theoretical side, we give an analytical upper bound on $B_{min}(M)$.

2 The *C* Bound: A New Lower Bound

We now present a new lower bound, called the clique bound (or simply C bound) for the minimum mosaic problem. Clique here refers to segments of input sequences used in the definition of the C bound that are pairwise incompatible (see blow). If we create a graph of sequence segments where there is an edge between two segments if they are incompatible, the C bound corresponds to cliques with some special property in this graph.

2.1 Breakpoint Placement

The breakpoint placement by the *CH* bound is a two-stage approach: first estimate the number of needed breakpoints inside each interval, and then place breakpoints along a horizontal line. A breakpoint placed this way is not associated with any particular sequence. Our first idea is to adopt a one-stage

approach: place necessary breakpoints directly in M. That is, each breakpoint belongs to a particular sequence in M. Formally,

Placement of necessary breakpoints. Place the *smallest* number (denoted as n_b) of breakpoints in M (i.e. a breakpoint is placed inside some sequence of M) such that for each interval $[i, j]$, no more than K_f clusters remain un-cut. Our first version of the C bound is equal to n_b, which is a lower bound on $B_{min}(M)$. This is due to Lemma 2, which is implied by Lemma 1.

Lemma 2. $n_b \leq B_{min}(M)$.

We say a lower bound x beats a lower bound y if x is always higher than or equal to y. First note that the C bound always beats the *CH* bound. This is because we can place breakpoints created by the C bound along a horizontal line by projecting these breakpoints onto the line. These are at least $B_{i,j}$ breakpoints for $[i, j]$ on the horizontal line. In practice, the C bound is usually higher (sometime significantly higher) than the *CH* bound.

We do not know a polynomial-time algorithm to compute the exact C bound. To compute the C bound in practice, we use the following integer linear programming (ILP) formulation. We define a binary variable $C_{r,c}$ for each row r and column c, where $C_{r,c} = 1$ if there is a breakpoint in row r *between* columns c and $c + 1$. For each interval $[i, j]$ with $|S_{i,j}|$ clusters within $[i, j]$, we create a binary variable $U_{i,j,k}$ for the k-th cluster within $[i, j]$, where $U_{i,j,k} = 1$ if this cluster is un-cut within $[i, j]$, and 0 otherwise.

Objective: minimize $\sum_{1 \leq r \leq n, 1 \leq k < m} C_{r,k}$.
Subject to

1 $U_{i,j,k} + \sum_{i \leq i' < j} C_{r,i'} \geq 1$, for each $1 \leq i < j \leq m$ and each row r in cluster k within $[i, j]$.

2 $\sum_{k=1}^{|S_{i,j}|} U_{i,j,k} \leq K_f$, for each $1 \leq i < j \leq m$.

For each $1 \leq k < m$ and $1 \leq r \leq n$, there is a binary variable $C_{r,k}$.
For each $1 \leq i < j \leq m$, and $1 \leq k \leq |S_{i,j}|$, there is a binary variable $U_{i,j,k}$.

Briefly, constraint (1) says that cluster k is un-cut if any sequence in the cluster contains no breakpoints. Constraint (2) says within each interval $[i, j]$, there is no more than K_f un-cut clusters. This ILP formulation can be solved relatively efficiently in practice for many datasets.

A polynomial-time computable bound. The C bound beats the *CH* bound both in theory and in practice. Still, we do not have a polynomial-time algorithm for computing it. This may limit its use for larger datasets. We now show a weaker version of the C bound (denoted as the C_w bound) that is polynomial-time computable and provably beats the *CH* bound. This provides more evidence on the strength of the exact C bound. The C_w bound is computed by solving the linear programing (LP) *relaxation* of the ILP formulation of the C bound. Briefly, linear programming relaxation treats each variable in the ILP formulation to be a general real variable (between 0 and 1 for a binary variable). The C_w bound is a legal lower bound on $B_{min}(M)$ because the objective of the LP relaxation

can not be higher than that of the original C bound. The number of variables and constraints in the ILP formulation for the C bound is $O(nm^2)$. Thus, the C_w bound can be computed in polynomial-time.

We now show that the C_w bound beats the *CH* bound.

Proposition 1. $C_w \geq CH$.

Proof. Conceptually, the *CH* bound can be computed by the following ILP formulation (called the *CH* formulation).

Objective: minimize $\sum_{1 \leq k < m} C_k$.
* Subject to: $\sum_{x=i}^{j-1} C_x \geq B_{i,j}$, for each $1 \leq i < j \leq m$.
For each $1 \leq k < m$, there is a *general* integer variable C_k.

C_i refers to the number of breakpoints between columns i and $i+1$. We will first show that C_w beats the objective of the LP *relaxation* of the *CH* formulation. Then we will show the LP relaxation of the *CH* formulation has an integer optimal solution. By combining these two observations, we have $C_w \geq CH$.

First, we let $C_{r,k}$ be an optimal solution for C_w (where $C_{r,k}$ is a real value between 0 and 1). We now let $C_k = \sum_{1 \leq r \leq n} C_{r,k}$. We claim C_k is a legal solution for the relaxed *CH* formulation (with the *identical* objective value as the C_w formulation). To show this, we only need to show C_k satisfies $\sum_{i \leq x < j} C_x \geq B_{i,j}$ for each interval $[i, j]$ when $B_{i,j} > 0$. Since $\sum_{i \leq x < j} C_x = \sum_{i \leq x < j, 1 \leq r \leq n} C_{r,x}$, and from constraint [1] in the C bound formulation,

$$\sum_{i \leq x < j} C_x \geq n - \sum_{k=1}^{|S_{i,j}|} n_{i,j}^k U_{i,j,k}$$

Since $0 \leq U_{i,j,k} \leq 1$, and $\sum_{k=1}^{|S_{i,j}|} U_{i,j,k} \leq K_f$ and note that $n_{i,j}^k$ is ordered non-decreasingly, we have: $\sum_{i \leq x < j} C_{i'} \geq n - \sum_{k=1}^{K_f} n_{i,j}^k = B_{i,j}$. We achieve the minimum value by making each $U_{i,j,k} = 1$ for $k \leq K_f$.

We now show the LP relaxation of the *CH* formulation has an integer optimal solution. For contradiction, we assume no integer solutions exists for the *CH* relaxed formulation. We consider an optimal solution to the LP relaxation of the *CH* formulation so that its first C_i with non-integer value occurs at a column p where $p < m$ is the *largest* among all such solutions. That is, C_p is not an integer, while C_k is an integer when $k < p$, and there is no optimal solutions with integer values for $C_1 \ldots C_p$. Let $C_p = v + f$, where v is an integer and $0 < f < 1$. Then we create a new solution C'_k by letting $C'_k = C_k$ when $k \neq p$ and $p+1$. Then $C'_p = v$ and $C'_{p+1} = C_{p+1} + f$. We need to show the changed solution is legal (i.e. satisfying constraint "*"). Note that only intervals $[i, j]$ overlapping breakpoint p need to be checked. This contains two cases: (a) intervals $[i, p+1]$; (b) intervals $[i, j]$, where $i \leq p$ and $p + 1 < j$. The type (a) intervals' bounds are satisfied because the bounds are integers and thus discarding f still satisfies the bounds since all previous C'_k are integer. The type (b) intervals are satisfied since the summation of C'_k values is the same as summation of C_i values. This contradicts

our previous assumption that there is no optimal solutions with integer values for $C_1 \ldots C_p$. Therefore, there exists an integer solution to the LP relaxation of the *CH* bound. □

2.2 Improving the *C* Bound

Simulation shows that the C bound (and the C_w bound) is usually higher than the *CH* bound. For the datasets we simulate, the gap between the C and the *CH* bounds is usually 10-20%, but the C bound can still be significantly lower than $B_{min}(M)$. We now describe techniques that significantly improve the C bound.

We start by strengthening Lemma 1. A moment's thought suggests Lemma 1 can be extended to *overlapping* segments of different intervals (instead of within a single interval). We say segment $[a_1, b_1]$ of row r_1 (denoted as $r_1[a_1, b_1]$) and segment $[a_2, b_2]$ of row r_2 (i.e. $r_2[a_2, b_2]$) are incompatible if $[a_1, b_1]$ and $[a_2, b_2]$ overlap (i.e. with non-empty intersection) and $r_1[a_1, b_1]$ and $r_2[a_2, b_2]$ are *not* identical within the overlapping region. Here, $[a_1, b_1]$ and $[b_1, c_1]$ are considered to overlap at $[b_1, b_1]$. For example, in Figure 2, $r_1[1, 2]$ and $r_2[2, 3]$ are compatible while $r_1[3, 4]$ and $r_2[2, 3]$ are incompatible. Then we have:

Lemma 3. *Two incompatible segments can not be colored by the same founder.*

Note that Lemma 1 is a special case of Lemma 3, and Lemma 3 can give a higher bound than the original C bound. This motivates an improved C bound as follows. For each interval $[i, j]$, we search for *pairwise* incompatible segments $[i_k, j_k]$, each for one of the $|S_{i,j}|$ clusters. Here, $i \leq i_k < j_k \leq j$. It is desired that the total length of the segments is small. The shorter the incompatible segments are, the more *restrictive* the placement of breakpoints is: fewer breakpoints can contribute to shorter segments and so more breakpoints may be needed. Then, we require no more than K_f incompatible segments remain un-cut: $\sum_{k=1}^{|S_{i,j}|} U_{i_k,j_k,c_{i,j,k}(i_k,j_k)} \leq K_f$. Here, $c_{i,j,k}(i_k, j_k)$ is the cluster index of the $[i_k, j_k]$ part within the k-th cluster of $[i, j]$ (since $[i_k, j_k]$ may have a different set of distinct rows from $[i, j]$). From Lemma 4, the new C bound beats the original C bound.

	c_1	c_2	c_3	c_4
r_1	0	0	1	1
r_2	1	0	0	1
r_3	0	0	1	0
r_4	0	1	0	1
r_5	0	0	0	1

Fig. 2. An example dataset

Lemma 4. *Using incompatible segments gives a higher C bound.*

Proof. For $[i, j]$, if $U_{i,j,k} = 1$, then there is *no* breakpoint between columns i and j. Since $[i_k, j_k]$ is contained inside $[i, j]$, this implies $U_{i_k,j_k,k'} = 1$ (where $k' = c_{i,j,k}(i_k, j_k)$). Thus, $U_{i,j,k} \leq U_{i_k,j_k,k'}$. Since the C bound prefers smaller $U_{i,j,k}$ values (by constraint 2 of the C bound formulation), using incompatible segments leads to a higher bound. □

As an example, we consider the dataset in Figure 2, where we assume $K_f = 3$. We first consider the ILP constraints of each interval from the original C bound.

For example, for interval $[1, 4]$, we have $U_{1,4,1} + U_{1,4,2} + U_{1,4,3} + U_{1,4,4} + U_{1,4,5} \leq 3$, and for $r_1 = 0011$ we have $U_{1,4,1} + C_{1,1} + C_{1,2} + C_{1,3} \geq 1$. Suppose as shown in Figure 2, we place a breakpoint between c_1 and c_2 in r_2 (i.e. $C_{2,1} = 1$), and a breakpoint between c_3 and c_4 for r_4 (i.e. $C_{4,3} = 1$). This would satisfy constraints of all intervals. But these two breakpoints will not be enough when we consider the following five pairwise incompatible segments: $r_1[3, 4]$, $r_2[1, 3]$, $r_3[3, 4]$, $r_4[2, 3]$ and $r_5[1, 3]$. These segments impose an ILP constraint: $U_{3,4,1} + U_{1,3,2} + U_{3,4,3} + U_{2,3,3} + U_{1,3,5} \leq 3$. Clearly, setting $C_{2,1}$ and $C_{4,3}$ to be 1 will not satisfy this constraint since $C_{4,3}$ does not contribute to $r_4[2, 3]$ (recall $C_{4,3}$ refers to the breakpoint between columns 3 and 4, and $r_4[2, 3]$ is the 3rd cluster within $[2, 3]$). So only $U_{1,3,2} = 0$ and the other four terms are equal to 1.

Finding incompatible segments. Ideally, we would like to find incompatible segments whose total length is minimized (called the shortest incompatible segments or SISs). It is not known whether there is a polynomial-time algorithm for finding the SISs. We also want to find more than one set of SISs to obtain higher lower bounds. Thus, we use the following heuristic algorithm for finding approximate SISs that works reasonably well in practice. This heuristic greedily finds incompatible segments for each cluster. To find multiple SISs, we choose different initial positions in the first cluster.

1 Order the clusters (e.g. in the order of their appearance in the dataset).
2 Set $pos_1 \leftarrow 1$.
3 while $pos_1 < m$
3a Initialize the first segment for row 1 as $r_1[pos_1, pos_1 + 1]$.
3b For each remaining cluster row r_i, let its segment be the shortest segment such that the segment is incompatible with all previous segments.
3c Set $pos_1 \leftarrow pos_1 + 1$.

Now with ILP constraints imposed on incompatible segments, higher C bounds can be obtained. Moreover, the C bound can also be computed faster than the original C bound by ILP. Simulation results show that the CPLEX ILP solver usually takes only a few seconds for fairly large datasets (say with 100 rows and 100 columns with $K_f = 10$), which is much faster than computing the original C bound. The speedup is likely due to the reduction of the size of the ILP formulation: *same* segments are often chosen for different overlapping intervals, which reduces the number of needed variables. Thus, the number of needed $U_{i,j,k}$ is often much smaller than that in the original formulation. Our experience shows that the new C formulation with overlapping segments can use only 10% (or fewer) variables as in the original formulation for larger datasets.

Other improvements. The C bound can be further improved by the following observations. (a) We can find different SISs by picking a small number of different cluster orders. (b) We can avoid enumeration of all $\binom{m}{2}$ intervals for a matrix with m columns in searching for incompatible segments. Suppose for an interval $[a, b]$ the found incompatible segments within $[a, b]$ are all between $[a_1, b_1]$. Then we can skip all intervals $[x, y]$ where $a \leq x \leq a_1$ and $b_1 \leq y \leq b$.

2.3 Application in Finding the Exact Minimum Mosaic Using Branch and Bound

The C bound can speedup program *RecBlock* in finding the exact minimum mosaic with the branch and bound scheme. We have experimented with the following straightforward approach. Briefly, we use the approximate C bound to determine whether a current search path can lead to a better solution. We first compute the C bounds for the segment of input data between column i and m, for each $1 \leq i \leq m-1$. For the purpose of efficiency, we only compute the C_w bound by solving linear programming relaxation. Program *RecBlock* builds a partial mosaic from column 1 to i during its search for the minimum mosaic. If the lower bound on the minimum number of breakpoints for the sub-matrix (from the current site to the right end of the input matrix) plus the currently needed breakpoints in the partial solution is no smaller than the known upper bound, then this search path will not lead to a better solution and can thus be dropped.

Other more advanced strategies are also possible. For example, we can make the lower bounds more effective by switching to heuristic mode with promising search paths at each column. This may help because it may find the optimal solution earlier whose optimality may be certified by the lower bound.

3 An Analytical Upper Bound

A natural question is how many breakpoints we may need in a minimum mosaic for an arbitrary (i.e. unknown) n by m matrix. An answer to this question gives an upper bound and also an estimate on the range of $B_{min}(M)$. It was stated in [6] that there can be as many as $(m-1)n/2$ breakpoints needed in a minimum mosaic when $K_f = 2$. We now extend to the general case.

Proposition 2. *$B_{min} \leq (1 - \frac{1}{K_f})(\frac{m}{\lceil log_2(K_f) \rceil} - 1)n$, for any K_f.*

Proof. First note that when $K_f = 2$, this reduces to the $(m-1)n/2$ bound. Our approach is similar to [11]. We divide M into $m/\lceil log_2(K_f) \rceil$ non-overlapping intervals, each with $\lceil log_2(K_f) \rceil$ columns. There are at most K_f distinct binary sequences within each interval. We then pick each of these unique sequences as founders within the interval. Thus, no breakpoints is needed inside intervals. Since there are $m/\lceil log_2(K_f) \rceil$ intervals for each of the n rows, we need no more than $(m/\lceil log_2(K_f) \rceil - 1)n$ breakpoints between intervals.

We can further improve this bound by carefully picking the colors for neighboring intervals and removing some breakpoints between the intervals in a way similar to [10,9]. More specifically, we construct a weighted bipartite graph for two neighboring intervals IV and IV'. A vertex F_i corresponds to a founder and also a sequence cluster colored with this particular founder. There is an edge of weight $n_{i,j}$ between vertices F_i within IV and F_j within IV' if there are $n_{i,j}$ input sequences as concatenation of founder F_i within IV and founder F_j within IV'. We add a weight 0 edge between two founders if there is no such combination in the input sequences. When choosing the coloring for IV and IV', we use

the maximum weighted *matching* of the graph: if F_i is matched with F_j, F_i and F_j are colored by the same founder. We avoid $n_{i,j}$ breakpoints between IV and IV' since there is no color change within these $n_{i,j}$ rows.

Since the input matrix M is unknown, we can not explicitly construct the bipartite graph and compute the maximum weighted matching. Nonetheless, we can still rely on the matching to improve the upper bound as follows. First, the total weight of the bipartite graph is n. Also, there exists a maximum weighted matching which is also a perfect matching since the bipartite graph is complete. We claim that there exists a perfect matching in the bipartite graph with at least n/K_f weight. To see this, note that there are $K_f!$ perfect matchings for a bipartite graph with K_f nodes on one side, and each edge appears in exactly $(K_f-1)!$ of these perfect matchings. Thus, the sum of the weights of all perfect matchings is $(K_f-1)!n$. The maximum weighted matching has weight of at least $(K_f-1)!n/K_f! = n/K_f$. Therefore, we can remove at least n/K_f breakpoints at each interval boundary by properly selecting how founders are matched for the two neighboring intervals. So we need no more than $(1-1/K_f)(m/\lceil log_2(K_f)\rceil - 1)n$ breakpoints for any input dataset with K_f founders. □

4 Simulation Results

We have implemented the lower bound method in program *RecBlock* with either CPLEX (a commercial ILP solver) or GNU GLPK ILP solver (mainly a demo of the functionalities for users without a CPLEX license). The CPLEX version is often much faster. The simulation results in this section are for the CPLEX version. We test our method for simulated data on a 3192 MHz Intel Xeon workstation. We use Hudson's program ms [1] to generate binary population sequences. We fix the scaled population mutation rate to 10, and the scaled recombination rate to 10, and generate 100 datasets for 20, 30, 40 and 50 sequences. We then remove from datasets any columns that have more than 95% or less than 5% 1s. This helps to remove more recent mutations and focus on the underlying mosaic structures.

Performance of lower bounds. To demonstrate the usefulness of the new C bound, we compare the performance of the *CH* bound and the C bound. We use program *RecBlock* to compute the exact $B_{min}(M)$. We also evaluate the approximate C bound (i.e. the C_w bound) obtained by solving LP relaxation. This is useful since the C_w bound can be more scalable when data size grows. The performance of the lower bounds is shown in Table 1. We use three statistics: (a) percent of datasets where the lower bound matches $B_{min}(M)$; (b) average gap between the lower bound and $B_{min}(M)$ (normalized by $B_{min}(M)$); (c) average running time. From Table 1, we can see that the C bound is very accurate for the range of data we test: for a large percentage of datasets, the C bound matches $B_{min}(M)$, and the average gap between the C bound and $B_{min}(M)$ is very small (within 2%). Moreover, the C bound remains accurate for larger K_f. In terms of efficiency, computing the C bound scales well with large K_f: the larger K_f is, the faster computation is. This is likely because the size of integer programs decreases

when K_f increases since fewer intervals contain more than K_f clusters when K_f increases. This is very useful to obtain an estimate of $B_{min}(M)$: program *RecBlock* gets increasingly slower when K_f increases. We also note that the C_w bound is very accurate, and slightly faster to compute than the original C bound. On the other hand, the *CH* bound performs poorly in all the cases with much larger gaps, although the *CH* bound is often much faster to compute.

Also, it appears that the GLPK version can be slow in computing the exact C bound, while it performs relatively well in computing the approximate C bound. For example, the GLPK version takes on average 208 seconds in computing the exact C bound (excluding one dataset where GLPK runs for more than one day but does not find the optimal solution), and 61 seconds for computing the approximate C bounds with the datasets with $n = 30$ and $K_f = 5$. As a comparison, the CPLEX version takes on average 24 seconds and 18 seconds respectively for computing the exact and the approximate C bounds for the same datasets. Thus, when using the GLPK version, computing the approximate C bounds may be more practical for some more difficult datasets.

Application in finding the exact minimum mosaic using branch and bound. We also evaluate how the C bound performs in speeding up program *RecBlock* with the branch and bound scheme. Simulation results are shown in Table 1. When K_f becomes larger, branch and bound are more likely to be effective. Although reduction of running time is modest in these simulations, greater speed-up could be achieved if better heuristics are used for finding upper bounds since the C bound is often close to the optimal solution. Moreover, since branch and bound needs to compute the C bounds for many segments, using a more powerful integer programming solver (e.g. CPLEX) may have a significant impact on the running time.

Table 1. Performance of lower bounds. Exact: compute $B_{min}(M)$ by program *RecBlock*. Exact (C): the branch and bound mode of program *RecBlock* using the C_w bound. %Opt: percentage of datasets where the lower bound matches $B_{min}(M)$. Gap: percentage of difference between $B_{min}(M)$ and the lower bound (divided by $B_{min}(M)$). T: time (in seconds). n: number of sequences. K_f: number of founders.

n		20				30				40				50			
K_f		5	6	7	8	5	6	7	8	5	6	7	8	5	6	7	8
Exact	T	1	7	39	255	8	95	848	3620	8	61	418	-	13	123	2024	-
Exact (C)	T	3	7	38	196	14	48	304	2152	13	46	361	-	21	73	1671	-
CH	*%Opt*	6	21	43	67	1	2	10	27	1	7	10	-	0	1	5	-
	Gap	34	27	18	9	42	37	31	24	38	32	27	-	43	39	33	-
	T	<1	<1	<1	<1	<1	<1	<1	<1	<1	<1	<1	<1	<1	<1	<1	<1
C	*%Opt*	85	94	98	100	81	82	85	90	72	73	77	-	63	69	68	-
	Gap	1	1	<1	0	1	2	1	1	2	2	2	-	2	2	2	-
	T	3	3	2	2	24	19	19	14	18	18	15	12	40	37	29	21
C_w	*%Opt*	83	94	98	100	77	81	83	90	71	70	77	-	59	67	66	-
	Gap	1	1	<1	0	1	2	1	1	2	2	2	-	2	2	2	-
	T	3	3	2	2	18	18	15	13	17	15	14	12	26	24	23	21

Acknowledgment. This work is supported by National Science Foundation [IIS-0803440]. I am also supported by the Research Foundation of University of Connecticut.

References

1. Hudson, R.: Generating Samples under the Wright-Fisher neutral model of genetic variation. Bioinformatics 18(2), 337–338 (2002)
2. 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)
3. Marchini, J., Howie, B., Myers, S., McVean, G., Donnelly, P.: A new multipoint method for genome-wide associationstudies by imputation of genotypes. Nature Genetics 39, 906–913 (2007)
4. Myers, S.R., Griffiths, R.C.: Bounds on the minimum number of recombination events in a sample history. Genetics 163, 375–394 (2003)
5. Rastas, P., Koivisto, M., Mannila, H., Ukkonen, E.: A Hidden Markov Technique for Haplotype Reconstruction. In: Casadio, R., Myers, G. (eds.) WABI 2005. LNCS (LNBI), vol. 3692, pp. 140–151. Springer, Heidelberg (2005)
6. Rastas, P., Ukkonen, E.: Haplotype Inference Via Hierarchical Genotype Parsing. In: Giancarlo, R., Hannenhalli, S. (eds.) WABI 2007. LNCS (LNBI), vol. 4645, pp. 85–97. Springer, Heidelberg (2007)
7. Roli, A., Blum, C.: Tabu Search for the Founder Sequence Reconstruction Problem: A Preliminary Study. In: Proceedings of Distributed Computing, Artificial Intelligence, Bioinformatics, Soft Computing, and Ambient Assisted Living (IWANN 2009), pp. 1035–1042 (2009)
8. 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)
9. Schwartz, R., Clark, A., Istrail, S.: Methods for Inferring Block-Wise Ancestral History from Haploid Sequences. In: Guigó, R., Gusfield, D. (eds.) WABI 2002. LNCS, vol. 2452, pp. 44–59. Springer, Heidelberg (2002)
10. Ukkonen, E.: Finding Founder Sequences from a Set of Recombinants. In: Guigó, R., Gusfield, D. (eds.) WABI 2002. LNCS, vol. 2452, pp. 277–286. Springer, Heidelberg (2002)
11. Wu, Y.: Analytical Upper Bound on the Minimum Number of Recombinations in the History of SNP Sequences in Populations, Info. Proc. Letters 109, 427–431 (2009)
12. Wu, Y., Gusfield, D.: Improved Algorithms for Inferring the Minimum Mosaic of a Set of Recombinants. In: Ma, B., Zhang, K. (eds.) CPM 2007. LNCS, vol. 4580, pp. 150–161. Springer, Heidelberg (2007)
13. Zhang, Q., Wang, W., McMillan, L., Prins, J., de Villena, F.P., Threadgill, D.: Genotype Sequence Segmentation: Handling Constraints and Noise. In: Crandall, K.A., Lagergren, J. (eds.) WABI 2008. LNCS (LNBI), vol. 5251, pp. 271–283. Springer, Heidelberg (2008)

The Highest Expected Reward Decoding for HMMs with Application to Recombination Detection

Michal Nánási, Tomáš Vinař, and Broňa Brejová

Faculty of Mathematics, Physics, and Informatics, Comenius University, Mlynská Dolina, 842 48 Bratislava, Slovakia

Abstract. Hidden Markov models are traditionally decoded by the Viterbi algorithm which finds the highest probability state path in the model. In recent years, several limitations of the Viterbi decoding have been demonstrated, and new algorithms have been developed to address them (Kall et al., 2005; Brejova et al., 2007; Gross et al., 2007; Brown and Truszkowski, 2010). In this paper, we propose a new efficient highest expected reward decoding algorithm (HERD) that allows for uncertainty in boundaries of individual sequence features. We demonstrate usefulness of our approach on jumping HMMs for recombination detection in viral genomes.

Keywords: hidden Markov models, decoding algorithms, recombination detection, jumping HMMs.

1 Introduction

Hidden Markov models (HMMs) are an important tool for modeling and annotation of biological sequences and other data, such as natural language texts. The goal of sequence annotation is to label each symbol of the input sequence according to its meaning or a function. For example, in gene finding, we seek to distinguish regions of DNA that encode proteins from non-coding sequence. An HMM defines a probability distribution $\Pr(A|X)$ over all annotations A of sequence X. Typically, one uses the well-known Viterbi algorithm (Forney Jr., 1973) or its variants for more complex models (Brejova et al., 2007) to find the annotation with the highest overall probability $\arg\max_A \Pr(A|X)$. In this paper, we design an efficient HMM decoding algorithm that finds the optimal annotation for a different optimization criterion that is more appropriate in many applications.

In recent years, several annotation strategies were shown to achieve better performance than the Viterbi decoding in particular applications (Kall et al., 2005; Gross et al., 2007; Brown and Truszkowski, 2010). Generally, they can be expressed in the terminology of *gain functions* introduced in the context of stochastic context-free grammars (Hamada et al., 2009). In particular, we choose a gain function $G(A, A')$ which characterizes similarity between a proposed annotation A and the (unknown) correct annotation A'. The goal is then to find the

A. Amir and L. Parida (Eds.): CPM 2010, LNCS 6129, pp. 164–176, 2010.

annotation A with the highest expected value of $G(A, A')$ over the distribution of A' defined by the HMM, conditioning on sequence X. That is, we maximize $E_{A'|X}[G(A, A')] = \sum_{A'} G(A, A')P(A'|X)$.

Intuitively, the gain function should characterize the measure of prediction accuracy appropriate for a particular application domain. If the sequences and the true annotations are generated from the HMM, the decoding algorithm optimizing the expected gain will on average reach higher prediction accuracy, measured by $G(A, A')$, than any other decoding.

In this framework, the Viterbi decoding optimizes the identity gain function $G(A, A') = [A = A']$, that is the gain is 1 if we predict the whole annotation exactly correctly, and 0 otherwise. There may be many high-probability annotations besides the optimal one, and they are disregarded by this gain function, even though their consensus may suggest a different answer that is perhaps more accurate locally. On the other hand, the posterior decoding (Durbin et al., 1998) predicts at each position a label that has the highest posterior probability at that position, marginalizing over all annotations. Therefore, it optimizes the expected gain under the gain function that counts the number of correctly predicted labels in A with respect to A'.

These two gain functions are extremes: the Viterbi decoding assigns a positive gain to the annotation only if it is completely correct, while the posterior decoding gain function rewards every correct label. It is often appropriate to consider gain functions in between these two extremes. For example, in the context of gene finding, Gross et al. (2007) use a gain function that assigns a score $+1$ for each correctly predicted coding region boundary and score $-\gamma$ for predicted boundary that is a false positive. Indeed, one of the main objectives of gene finding is to find exact positions of these boundaries, since even a small error may change the predicted protein significantly. Parameter γ in the gain function controls the trade-off between sensitivity and specificity.

While the coding region boundaries are well defined in gene finding, and it is desirable to locate them precisely, in other applications, such as transmembrane protein topology prediction, we only wish to infer the approximate locations of feature boundaries. The main reason is that the underlying HMMs do not contain enough information to locate the boundaries exactly, and there are typically many annotations of similar probability with slightly different boundaries. This issue was recently examined by Brown and Truszkowski (2010) in a Viterbi-like setting, where we assign gain to an annotation, if all feature boundaries in A are within some distance W from the corresponding boundary in the correct annotation A'. Unfortunately, the problem has to be addressed by heuristics, since it is NP-hard even for $W = 0$.

In this paper, we propose a new gain function in which each feature boundary in A gets score $+1$ if it is within distance W from the corresponding boundary in A', and score $-\gamma$ otherwise. Our definition allows to consider nearby boundary positions as equivalent, as in Brown and Truszkowski (2010), yet it avoids the requirement that the whole annotation needs to be essentially correct to receive

any gain at all. Another benefit is that our gain function can be efficiently optimized in time linear in the length of the input sequence.

We apply our algorithm to the problem of detecting recombination in the genome of the human immunodeficiency virus (HIV) with jumping HMMs (Schultz et al., 2006). A jumping HMM consists of a profile HMM (Durbin et al., 1998) for each known subtype of HIV. Recombination events are represented by a special jump transitions between different profile HMMs. The goal is to determine for a new HIV genome whether it comes from one of the known subtypes or whether it is a recombination of several subtypes. However, the exact position of a breakpoint can be difficult to determine, particularly if the two recombining strains were very similar near the recombination point. Our gain function corresponds to this problem very naturally: it scores individual predicted recombination points, but allows some tolerance in their exact placement.

2 HERD: The Highest Expected Reward Decoding

In this section, we propose a new gain function and describe an algorithm for finding the annotation with the highest expected gain. Our algorithm is a non-trivial extension of the maximum expected boundary accuracy decoding (Gross et al., 2007).

Hidden Markov models and notation. A *hidden Markov model (HMM)* is a generative probabilistic model with a finite set of states V and transitions E. There is a single designated start state s and a final state t. The generative process starts in the start state, and in each round it emits a single symbol x_i from the *emission probability* distribution e_{v_i,x_i} of the current state v_i, and then changes the state to v_{i+1} according to the *transition probability* distribution $a_{v_i,v_{i+1}}$. The generative process continues until the final state is reached. Thus, the joint probability of generating a sequence $X = x_1, \ldots, x_n$ by a state path $\pi = s, v_1, \ldots, v_n, t$ is $\Pr(\pi, X) = a_{s,v_1} \cdot \prod_{i=1}^{n} e_{v_i,x_i} \cdot a_{v_i,v_{i+1}}$, where $v_{n+1} = t$. In other words, the HMM defines a probability distribution $\Pr(\pi, X)$ over all possible sequences X and state paths π, or perhaps more appropriately, for a given sequence X, the HMM defines a conditional distribution over all state paths $\Pr(\pi \mid X)$.

Our aim is to produce an *annotation* of an input sequence X, i.e. to label each symbol of X by a color corresponding to its function (e.g., coding or non-coding in the case of gene finding, or a virus subtype in case of recombination detection). Position i in the annotation $A = a_1 \ldots a_n$ is a *boundary*, if a_i and a_{i+1} are different colors. For convenience, we consider positions 0 and n as boundaries. A *feature* is a region between two consecutive boundaries.

To use HMMs for sequence annotation, we color each state v by a color $c(v)$. Every state path $\pi = s, v_1, \ldots, v_n, t$ thus implies an annotation $c(\pi) = c(v_1) \ldots c(v_n)$. In general, multiple states can have the same color, and several state paths may produce the same annotation. Thus, HMMs also define a probability distribution over annotations A, where $\Pr(A \mid X) = \sum_{\pi:c(\pi)=A} \Pr(\pi \mid X)$.

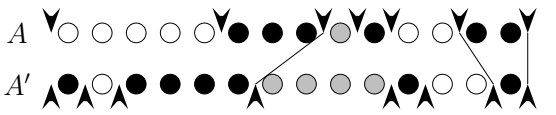


Fig. 1. Example of buddy pairs in two annotations over three colors (white, gray, black) for $W = 3$. Boundaries are shown by arrows, buddy pairs are connected by lines. The second boundary in A does not have a buddy pair due to condition (ii), whereas the fourth and fifth boundary due to condition (iii). In this example, $G(A, A') = 3 - 4\gamma$.

The highest expected reward decoding problem. To formally define our problem, we first define a gain function $G(A, A')$ characterizing similarity between any two annotations A and A' of the same sequence. We assign a positive score to a boundary in A if A' contains a corresponding boundary sufficiently close so that they can be considered equivalent. This notion of closeness is formalized in the following definition (see also Figure 1).

Definition 1. *Let A and A' be two annotations of the same sequence. Boundaries i in A and j in A' are called* buddies *if (i) both of them separate the same pair of colors c_1 and c_2, (ii) $|i - j| < W$, and (iii) there is no other boundary at positions $\min\{i, j\}, \ldots, \max\{i, j\}$ in either A or A'.*

The intuition behind condition (iii) is that the buddies should correspond to boundaries that are only slightly shifted from their correct position, but still separate essentially the same pair of features. In the extreme case, such as the boundaries between gray and black features in Figure 1, even a slight shift in the boundary causes the flanking black features to become non-overlapping. Condition (iii) in fact enforces that pairs of such non-overlapping features are not considered as corresponding to each other. Moreover, condition (iii) also enforces that each boundary in A' is a buddy to at most one boundary in A and vice versa.

Definition 2 (Highest expected reward decoding problem). *Let gain function $G(A, A')$ assign score +1 to each boundary in A if it has a buddy in A' and score $-\gamma$ to boundaries in A without a buddy. In the* highest expected reward decoding (HERD)*, we seek the annotation A maximizing the expected gain $E_{A'|X}[G(A, A')] = \sum_{A'} G(A, A') \Pr(A' \mid X)$, where the conditional probability $\Pr(A' \mid X)$ is defined by the HMM as $\sum_{\pi:c(\pi)=A'} \Pr(\pi, X)/\Pr(X)$.*

Note that our objective $E_{A'|X}[G(A, A')]$ can be further decomposed. In particular, from linearity of expectation, $E_{A'|X}[G(A, A')] = \sum_{i \in B(A)} R_\gamma(p_{A,i})$, where $B(A)$ is the set of all boundaries in A, $p_{A,i}$ is the posterior probability in the HMM that the boundary i in A has a buddy, and $R_\gamma(p) = p - \gamma \cdot (1 - p)$ is the expected score (reward) for a boundary with posterior probability p.

The HERD algorithm computes posterior probabilities and expected rewards for all possible boundaries and then uses dynamic programming to choose an annotation with the highest possible sum of expected rewards in its boundaries. The details of the algorithm are described below.

$$
\begin{array}{lll}
 & & i \\
A & \bullet\circ\circ\circ\circ\bullet\bullet\circ & p(i,\circ,\bullet,3,2) \\
\hline
 & \circ\bullet\bullet\bullet & = \Pr(a_{i-2\ldots i+1} = \circ\bullet^3 \mid X) \\
 & \circ\bullet\bullet & + \Pr(a_{i-1\ldots i+1} = \circ\bullet^2 \mid X) \\
 & \circ\bullet & + \Pr(a_{i\ldots i+1} = \circ\bullet \mid X) \\
 & \circ\circ\bullet & + \Pr(a_{i\ldots i+2} = \circ^2\bullet \mid X)
\end{array}
$$

Fig. 2. Illustration of annotations contributing probability to $p(i, c_1, c_2, w_L, w_R)$ for $W = 3$

Expected reward of a boundary. To compute the posterior probability $p_{A,i}$ that a boundary i in A has a buddy in A' sampled from the HMM, it is sufficient to examine only a local neighborhood of boundary i in A. In particular, let c_1 and c_2 be the two colors separated by this boundary and n_L and n_R be the lengths of the two adjacent features. If $n_L \leq W$, the leftmost possible position of the buddy in A' is $i - n_L + 1$, otherwise it is $i - W + 1$; a symmetric condition holds for the rightmost position. Therefore, if A has a buddy in A', it must be in the interval $[i - w_L + 1, i + w_R - 1]$, where $w_L = \min\{W, n_L\}$, and $w_R = \min\{W, n_R\}$. If we denote by $p(i, c_1, c_2, w_L, w_R)$ the sum of probabilities of all annotations A' that have a buddy for boundary i in the interval $[i - w_L + 1, i + w_R - 1]$ (see Figure 2), the expected reward of boundary i will be $R_\gamma(p(i, c_1, c_2, w_L, w_R))$.

Probability $p(i, c_1, c_2, w_L, w_R)$ can be expressed as a sum of simpler terms, one for each possible position j of the buddy in A':

$$
\begin{aligned}
p(i, c_1, c_2, w_L, w_R) = & \sum_{j=i-w_L+1}^{i} \Pr(a_{j\ldots i+1} = c_1 {c_2}^{i-j-1} \mid X) \\
& + \sum_{j=i+1}^{i+w_R-1} \Pr(a_{i\ldots j+1} = {c_1}^{j-i-1} c_2 \mid X).
\end{aligned}
$$

Note that if the buddy is at position $j \leq i$, this position needs to have color c_1 and all successive positions up to $i + 1$ need to have color c_2, otherwise there would be a different boundary between i and j in A'. However, positions outside of interval $[j, i+1]$ can be colored arbitrarily. Similarly for the buddy at position $j > i$, all positions from i up to j need to have color c_1 and position $j + 1$ color c_2. Also note that all terms in the sum represent disjoint sets of annotations, and therefore we are justified to compute the probability of the union of these sets by a sum. All terms of this sum can be computed efficiently, as described at the end of this section.

Finding the annotation with the highest expected reward. Once the expected rewards $R_\gamma(p(i, c_1, c_2, w_L, w_R))$ are known for all possible boundaries, we can compute the annotation A with the highest expected gain by dynamic programming. We can view the algorithm as the computation of the highest-weight

directed path between two vertices in a directed acyclic graph, where each path corresponds to one annotation and its weight to the expected gain.

In particular, the graph has a vertex (i, c, w) for each position i, color c, and window length $w \leq W$. This vertex represents a boundary at position i between an unspecified color on the left and the color c on the right, where the adjacent feature of color c has length exactly w if $w < W$, or at least W otherwise. If $w < W$, we will connect vertex (i, c, w) with vertices $(i + w, c', w')$ for all colors c' and lengths $w' \leq W$. Each such edge will have weight $R_\gamma(p(i+w, c, c', w, w'))$, representing the expected reward of boundary at position $i + w$. If $w = W$, we connect vertex (i, c, w) with vertices $(i + w'', c, c', w')$ for all $w'' \geq W$, $w' \leq W$ and color c' by *long-distance edges*. The weight of such edges will be $R_\gamma(p(i + w'', c, c', W, w'))$.

To finish the construction, we will assume that positions 0 and $n + 1$ are labeled by special colors c_s and c_f and that these two features have corresponding nodes in the graph. We also add a starting vertex $(-1, c_s, 1)$ and connect it to vertices $(0, c, w)$ according to normal rules. The annotation with the highest reward corresponds to the highest-weight path from vertex $(-1, c_s, 1)$ to vertex $(n, c_f, 1)$.

In this graph, the number of long-distance edges is quadratic in the length of sequence X, leading to an inefficient algorithm. Fortunately, the cost of a long-distance edge from (i, c, w) to $(i + w'', c, c', w')$ does not depend on index i, only on $i + w''$. Therefore, every long-distance edge can be replaced by a path through a series of special collector vertices of the form (i, c) for a position i and color c. There is an edge of weight 0 from (i, c, W) to $(i + W, c)$ for entering the collector path at an appropriate minimum distance from i, edge of weight 0 from (i, c) to $(i + 1, c)$ for continuing in the collector path, and an edge of weight $R_\gamma(p(i, c, c', W, w'))$ for leaving the collector path from vertex (i, c) to vertex (i, c', w'). This modified graph has $O(nWC)$ vertices and $O(nW^2C^2)$ edges, where n is the length of the sequence, W is the size of the window, and C is the number of different colors in the HMM.

Implementation details and running time. The only remaining detail is the computation of the posterior probabilities of the form $\Pr(a_{i\ldots i+w} = c_1 c_2{}^w \mid X)$ and $\Pr(a_{i\ldots i+w} = c_1^w c_2 \mid X)$ needed to compute $p(i, c, c', w, w')$. We will show how to compute the first of these two quantities, the second is analogous.

First, we use the standard forward algorithm (Durbin et al., 1998) to compute $F[i, v]$, the sum of the probabilities of all state paths ending in state v after generating the first i symbols from X. We use a modified backward algorithm (Durbin et al., 1998) to compute $B[i, v, w]$, the sum of the probabilities of all state paths generating symbols $x_i \ldots x_n$ that start in state v and generate the first w symbols in the states of color $c(v)$. Values $B[i, v, 1]$ are computed by the standard backward algorithm, and $B[i, v, w]$ for $1 < w \leq W$ is computed as follows:

$$B[i, v, w] = \sum_{\substack{v \to v' \\ c(v) = c(v')}} B[i + 1, v, w - 1] \cdot e_{v,x_i} \cdot a_{v,v'}.$$

Finally, the desired posterior probability is obtained by combining forward and backward probabilities over all transitions passing from color c_1 to color c_2 at position i:

$$\Pr(a_{i\ldots i+w} = c_1{c_2}^w \mid X) = \sum_{\substack{v \to v' \\ c(v)=c_1 \\ c(v')=c_2}} F[i,v] \cdot a_{v,v'} \cdot B[i+1,v',w] / \Pr(x).$$

The standard forward algorithm works in $O(n|E|)$ time, our extended backward algorithm takes $O(nW|E|)$ time. Posterior probabilities are summarized from these quantities also in $O(nW|E|)$ time. Finally, we construct and search the graph in $O(nW^2C^2)$ time. Thus the overall running time is $O(nW|E|+nW^2C^2)$. Note that the time is linear in the sequence length, which is very important for applications in genomics, where we analyze very long genomic sequences.

3 Application to Viral Recombination Detection

Most HIV infections are caused by HIV-1 group M viruses. These viruses can be classified by a phylogenetic analysis into several subtypes and sub-subtypes. However, some HIV genomes are a mosaic of sequences from different subtypes resulting from recombination between different strains (Robertson et al., 2000). Our goal is to classify whether a newly sequenced HIV genome comes entirely from one of the known subtypes or whether it is a recombination of different subtypes, which is important for monitoring the HIV epidemics.

Schultz et al. (2006) propose to detect recombination by jumping HMMs. In this framework, multiple sequence alignment of known HIV genomes is divided into parts corresponding to individual subtypes or sub-subtypes, and a profile HMM is built for each. A profile HMM (Durbin et al., 1998) represents one column of alignment by a match state, insert state and delete state. Emission probabilities of the match state correspond to the frequencies of symbols in that alignment column. The insert state represents sequences inserted immediately after the column, and the delete state is a silent state allowing to bypass the match state without emitting any symbols, thus corresponding to a deletion. A jumping HMM also contains low probability jump transitions between profile HMMs corresponding to individual subtypes, as shown in Figure 3.

To use a jumping HMM for recombination detection, we color each state by its subtype. Then, boundaries in the annotation correspond to recombination breakpoints. Schultz et al. (2006) use the Viterbi algorithm and report the annotation corresponding to the most probable state path. However, the same annotation can be obtained by many different state paths corresponding to different alignments of the input sequence to the profile HMMs. Even though in the latest version of their software (Schultz et al., 2009) they augment the output by displaing the posterior probabilities, they still output only a single annotation obtained by the Viterbi algorithm. Since we are not interested in the alignment,

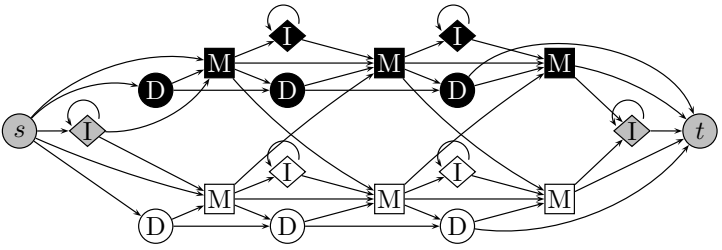


Fig. 3. A small example of a jumping HMM with two profile HMMs. For readability, jumping transitions between match states (M) and insert (I) or delete (D) states are not shown.

only in the annotation, it is more appropriate to use the most probable annotation instead of the most probable path. However, the problem of finding the most probable annotation is NP-hard for many HMMs (Brejova et al., 2007), and jumping HMMs, due to their complicated structure with many transitions between states of different color, are likely to belong to this class.

The HERD bypasses this computational difficulty by maximizing a different gain function that scores individual breakpoints rather than the whole annotation. Compared to the Viterbi algorithm, our algorithm considers all possible state paths (alignments) contributing to the resulting annotation. In addition, our algorithm considers nearby potential recombination points as equivalent, since in practice it is difficult to determine the exact recombination point, particularly in strongly conserved regions or between related subtypes.

The use of jumping HMMs on HIV genomes is relatively time consuming, as a typical HIV genome has the length of almost 10,000 bases, and the jumping HMM has 7,356,740 transitions. Schultz et al. (2006) use the beam search heuristic to speed up the Viterbi algorithm. Unfortunately, this heuristic is not applicable in our case, and our algorithm is also asymptotically slower than the Viterbi algorithm by a factor of W. To reduce the running time, we use a simple anchoring strategy, similar to the heuristics frequently used in the global sequence alignment (Kurtz et al., 2004). We have selected 19 well-conserved portions of the HIV multiple alignment as anchors, and align the consensus sequence of each anchor to the query sequence. In the forward and backward algorithm, we constrain the alignment of the query to the profile HMMs so that the position of the anchor in the query aligns to its known position in the profile HMM. We also extend the algorithm described above to handle silent states by modifying the preprocessing stage.

4 Experiments

A toy sequence annotation HMM. We have first tested our algorithm on data generated from a toy HMM in Figure 4. This HMM has multiple state paths for a given annotation, and we have previously demonstrated that the most probable annotation is more accurate than the annotation corresponding to the most probable state path found by the Viterbi algorithm (Brejova et al., 2007).

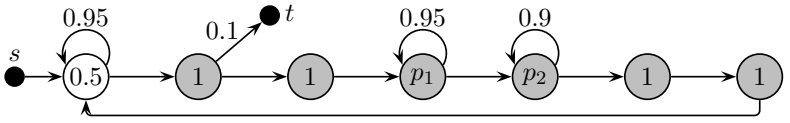


Fig. 4. A toy HMM emitting symbols over the binary alphabet, where the numbers inside states represent the emission probability of the symbol 1. States s and t are silent. The HMM outputs alternating white regions of mean length 20 and gray regions of mean length 34. The distribution of symbols is constant in the white regions, while in the gray regions it changes towards the end. The gray regions are flanked by a two-symbol signal 11 on both sides. The HMM was inspired by models of CT-rich intron tails in gene finding (Brejova et al., 2007).

Table 1 shows different measures of accuracy for several decoding algorithms on 5000 randomly generated sequences of mean length about 500. We report results for two sets of parameter values of the model, however, the trends observed in the table generally hold also for the other combinations of p_1 and p_2. As we have shown earlier, the extended Viterbi algorithm (EVA) (Brejova et al., 2007) for finding the most probable annotation generally outperforms the Viterbi algorithm. The HERD with parameters $W = 5$ and $\gamma = 1$ is more accurate when the performance is measured by its own gain function, which is not surprising, since the data and baseline predictions are generated from the same model as is used for annotation. On the other hand, the HERD colors fewer bases correctly and tends to place boundaries on average further away from the correct ones than the EVA. This is also not unexpected, as the HERD explicitly disregards small differences in the boundary position. We have also measured sensitivity and specificity in predicting individual features. Here the HERD works better than the EVA for some parameter settings (e.g. $p_1 = p_2 = 0.9$ in the table), but not for others. We have also run the HERD with $W = 1$, which is equivalent to maximum expected boundary accuracy decoding (Gross et al., 2007). The accuracy of this decoding is very poor for $\gamma = 1$, but markedly improves for lower penalty $\gamma = 0.1$. The reason is that for $W = 1$, we sum over fewer state paths and therefore the posterior probability of a boundary rarely reaches the threshold $1/2$ necessary to achieve positive expected reward at $\gamma = 1$.

HIV recombination detection. Table 2 shows the accuracy of the HERD on predicting recombination in HIV genomes. In all tests, we have used the sequence data and the jumping of Schultz et al. (2006), though in most tests we have increased the jump probability P_j from 10^{-9} to 10^{-5}. With the original value, the HERD rarely predicts any recombination, since the posterior probability of a breakpoint has to be at least $1/2$ for $\gamma = 1$ to receive a positive score, and with the lower jumping probability, we usually do not reach such a level of confidence. We have conducted the tests on a 1696 column region of the whole genomic alignment, starting at position 6925. This restriction allowed us to test higher number of sequences than Schultz et al. (2006) reasonably fast.

Table 1. The accuracy on synthetic data generated from the HMM in Figure 4. (i) Fraction of the bases colored by the same color by the algorithm and the correct annotation (baseline). (ii) Gain $G(A, A')$ of the prediction compared to the baseline. For evaluation, the parameters of the gain function were set to $W = 5$ and $\gamma = 1$, even though in some tests we have used different parameters in the algorithm. (iii) A feature is predicted correctly if there is a corresponding feature of the same color in the baseline with both boundaries within the distance of less than 5. Specificity (sp.) is the fraction of all predicted features that are correct, and sensitivity (sn.) is the fraction of baseline features that are correctly predicted. (iv) Mean distance between the baseline and predicted boundary for all correctly predicted features.

Algorithm	% bases correct$^{(i)}$	Gain $^{(ii)}$	Feature sp.$^{(iii)}$	Feature sn.$^{(iii)}$	Avg. dist.
HMM parameters $p_1 = 0.9$, $p_2 = 0.9$					
HERD $W = 5, \gamma = 1$	88.7%	12.7	75.9%	66.9%	1.8
HERD $W = 1, \gamma = 1$	47.5%	3.0	55.1%	17.8%	0.0
HERD $W = 1, \gamma = 0.1$	90.4%	2.4	51.8%	66.0%	0.9
Viterbi	89.4%	8.9	66.3%	47.3%	0.7
Extended Viterbi	91.2%	10.3	69.9%	56.2%	0.8
HMM parameters $p_1 = 0.7$, $p_2 = 0.8$					
HERD $W = 5, \gamma = 1$	77.6%	5.9	54.8%	39.3%	1.37
HERD $W = 1, \gamma = 1$	47.5%	3.0	55.0%	17.7%	0.0
HERD $W = 1, \gamma = 0.1$	79.6%	-2.7	38.2%	43.9%	0.9
Viterbi	75.0%	3.6	51.2%	25.7%	0.4
Extended Viterbi	79.7%	4.1	49.0%	31.3%	0.6

The first set of tests was done on 62 real HIV sequences without known recombination. These sequences were selected from the subtypes A1, B, C, D, F1 (10 sequences from each subtype) and G, A2, F2 (5, 3, and 4 sequences respectively) and omitted from the training set (except for the subtypes A2, F1 and F2 which have very few samples). As we can see in Table 2, the Viterbi algorithm always predicts the correct result. Our algorithm on the jumping HMM with the original low jumping probability $P_j = 10^{-9}$ also produces correct answer every time. However, the value of $P_j = 10^{-5}$ leads to spurious recombinations predicted in 11.3% of sequences, thus lowering the accuracy.

The second set of sequences contains artificial recombinants. Each of them was created as a combination of two sequences from two different subtypes by alternating regions of length 300. The set contains recombinants between subtype pairs A-B, A-C, A-G, B-C, B-G and C-G, 50 sequences from each pair. Our algorithm performs slightly better with respect to the total number of correctly labeled bases and average distance to the correct boundary, and also it finds individual features (recombinant regions) with much greater sensitivity and specificity if we allow some tolerance in the boundary placement. For $W = 1$, the HERD has a very low accuracy even for lowered penalty $\gamma = 0.1$. This suggests that our generalization of the maximum expected boundary accuracy decoding to the case $W > 1$ is crucial in this setting.

Table 2. The accuracy on the HIV recombination data. The meaning of the columns is the same as in Table 1, except that we use $W = 10$ and $\gamma = 1$ in the definition of the gain function and correctly predicted features.

Algorithm	% bases correct$^{(i)}$	Gain $^{(ii)}$	Feature sp.$^{(iii)}$	Feature sn.$^{(iii)}$	Avg. dist.
Sequences without recombination					
HERD, $W = 10, \gamma = 1, P_j = 10^{-9}$	100.0%	2.0	100.0%	100.0%	0.0
HERD, $W = 10, \gamma = 1, P_j = 10^{-5}$	93.7%	1.5	83.9%	83.9%	0.0
Viterbi	100.0%	2.0	100.0%	100.0%	0.0
Sequences with artificial inter-subtype recombination					
HERD $W = 10, \gamma = 1, P_j = 10^{-5}$	95.7%	2.61	63.1%	58.9%	2.4
HERD $W = 1, \gamma = 0.1, P_j = 10^{-5}$	81.6%	1.17	37.7%	30.2%	1.4
Viterbi	95.4%	2.1	53.4%	47.9%	1.8
Sequences with artificial intra-subtype recombination					
HERD $W = 10, \gamma = 1, P_j = 10^{-5}$	91.6%	1.7	46.5%	41.9%	2.7
Viterbi	88.0%	1.3	32.8%	26.1%	2.7

In the third test, we have used the same procedure to create 170 artificial recombinants between sequences of two sub-subtypes of the same subtype (A1 and A2, F1 and F2), and from the two subtypes (B and D) at a small phylogenetic distance that is more typical for sub-subtypes. The overall accuracy is lower in this test, because it is more difficult to distinguish recombination among more closely related sequences. The HERD is still much more accurate at the feature level and also more accurate than the Viterbi algorithm on the base level.

One issue with our tests is that we have used a lower jump probability $P_j = 10^{-9}$ for sequences without recombination and a higher value $P_j = 10^{-5}$ for sequences with recombination. This distinction is justified by the fact that although recombinant sequences are generally rare, suggesting a low jumping probability, they usually have several recombination points, whose detection then requires a higher value of P_j. In practice, when faced with a sequence of unknown origin we propose to first test whether the sequence is likely to be a recombinant, perhaps by a likelihood ratio test with nested models (Felsenstein, 2004) in which P_j is optimized for the input sequence in one model and set to 0 for the null model. If the sequence appears to contain recombination, we can then apply the HERD with the higher value of P_j to determine the breakpoints.

We have also run our algorithm on 12 naturally occurring recombinants, using $W = 10$, $\gamma = 1.5$, and $P_j = 10^{-5}$. Here, we have used the whole length of the sequence. Due to the small number of sequences and uncertain annotation, we do not report the accuracy statistics. Nonetheless, on six sequences, the HERD found the correct set of recombining subtypes (on annotated regions). Two of them the HERD annotated better than Viterbi (CRF08, CRF12). On the remaining six, the HERD predicted at least one erroneous subtype and often misplaced breakpoints or jumped frequently, but the Viterbi algorithm also made numerous mistakes on the two of these sequences.

5 Conclusion

In this paper, we have introduced a novel decoding algorithm for hidden Markov models seeking an annotation of the sequence in which boundaries of individual sequence features are at least approximately correct. This decoding is particularly appropriate in situations where the exact boundaries are difficult to determine, and perhaps their knowledge is not even necessary.

We apply our algorithm to the problem of recombination detection in HIV genomes. Here, the Viterbi decoding considers for a given annotation only a single alignment of the query to the profile HMMs and only one placement of breakpoints. In contrast, we marginalize the probabilities over all possible alignments and over nearby placements of recombination boundaries. As a result, we are able to predict individual recombinant regions with greater sensitivity and specificity.

Our experiments also suggest venues for future improvement. First of all, the accuracy results vary with the choice of parameters P_j, W, and γ. It remains an open question how to choose these parameters in a principled way. We have also observed that our algorithm does not perform as well as the Viterbi algorithm in finding the exact boundaries. Perhaps this could be solved by a gain function in which a boundary with a more distant buddy gets a smaller score. Similarly, our algorithm performs in some tests slightly worse in terms of base-level accuracy, and this shortcoming perhaps could be addressed by adding a positive score for every correctly colored nucleotide to the gain function. In general, the framework of maximum expected gain decoding is very promising, because it allows to tailor decoding algorithm to a specific application domain.

Acknowledgements. We would like to thank Dan Brown and Jakub Truszkowski for helpful discussion on related problems. Research of TV and BB is funded by European Community FP7 grants IRG-224885 and IRG-231025.

References

Brejova, B., Brown, D.G., Vinar, T.: The most probable annotation problem in HMMs and its application to bioinformatics. Journal of Computer and System Sciences 73(7), 1060–1077 (2007)

Brown, D.G., Truszkowski, J.: New decoding algorithms for hidden Markov models using distance measures on labellings. BMC Bioinformatics 11(S1), S40 (2010)

Durbin, R., Eddy, S., Krogh, A., Mitchison, G.: Biological sequence analysis: Probabilistic models of proteins and nucleic acids. Cambridge University Press, Cambridge (1998)

Felsenstein, J.: Inferring phylogenies. Sinauer Associates (2004)

Forney Jr., G.D.: The Viterbi algorithm. Proceedings of the IEEE 61(3), 268–278 (1973)

Gross, S.S., Do, C.B., Sirota, M., Batzoglou, S.: CONTRAST: a discriminative, phylogeny-free approach to multiple informant de novo gene prediction. Genome Biology 8(12), R269 (2007)

Hamada, M., Kiryu, H., Sato, K., Mituyama, T., Asai, K.: Prediction of RNA secondary structure using generalized centroid estimators. Bioinformatics 25(4), 465–473 (2009)

Kall, L., Krogh, A., Sonnhammer, E.L.L.: An HMM posterior decoder for sequence feature prediction that includes homology information. Bioinformatics 21(S1), i251–i257 (2005)

Kurtz, S., Phillippy, A., Delcher, A.L., Smoot, M., Shumway, M., Antonescu, C., Salzberg, S.L.: Versatile and open software for comparing large genomes. Genome Biology 5(2), R12 (2004)

Robertson, D.L., et al.: HIV-1 nomenclature proposal. Science 288(5463), 55–56 (2000)

Schultz, A.-K., Zhang, M., Bulla, I., Leitner, T., Korber, B., Morgenstern, B., Stanke, M.: jpHMM: improving the reliability of recombination prediction in HIV-1. Nucleic Acids Research 37(W), W647–W651 (2009)

Schultz, A.-K., Zhang, M., Leitner, T., Kuiken, C., Korber, B., Morgenstern, B., Stanke, M.: A jumping profile Hidden Markov Model and applications to recombination sites in HIV and HCV genomes. BMC Bioinformatics 7, 265 (2006)

Phylogeny- and Parsimony-Based Haplotype Inference with Constraints

Michael Elberfeld and Till Tantau

Institut für Theoretische Informatik
Universität zu Lübeck, 23538 Lübeck, Germany
{elberfeld,tantau}@tcs.uni-luebeck.de

Abstract. Haplotyping, also known as haplotype phase prediction, is the problem of predicting likely haplotypes based on genotype data. One fast computational haplotyping method is based on an evolutionary model where a perfect phylogenetic tree is sought that explains the observed data. In their CPM 2009 paper, Fellows et al. studied an extension of this approach that incorporates prior knowledge in the form of a set of candidate haplotypes from which the right haplotypes must be chosen. While this approach may help to increase the accuracy of haplotyping methods, it was conjectured that the resulting formal problem *constrained perfect phylogeny haplotyping* might be NP-complete. In the present paper we present a polynomial-time algorithm for it. Our algorithmic ideas also yield new fixed-parameter algorithms for related haplotyping problems based on the maximum parsimony assumption.

1 Introduction

In large-scale studies of the relation between genomic variation and phenotypic traits, low-cost sequencing methods are used to read out the DNA sequences of many individuals. For each individual the bases present on the two chromosomes at a large number of SNP (single nucleotide polymorphism) sites are determined, yielding the individual's *genotype* for the different sites. In order to study phenotypic traits that are related to the bases present on multiple loci on a single DNA strand, it is important to determine *haplotypes* rather than genotypes. They describe how bases are assigned to chromosomes (this assignment of bases to haplotypes is also known as *phasing*), but are expensive to determine directly. *Haplotype inference* or just *haplotyping* methods aim at predicting haplotypes from genotypes computationally by using biological insights into the haplotype distribution in a population. They either use statistics, pioneered in [11], or combinatorics, the two most common approaches being the *perfect phylogeny method* (haplotype evolution is assumed to take place with unique point mutation and without recombination) and the *maximum parsimony method* (haplotype evolution is assumed to produce only few haplotypes).

Most combinatorial algorithms ignore prior knowledge that we might have on which haplotypes may be permissible to explain a given genotype. In some situations a pool of haplotypes from prior studies is already known and we should

A. Amir and L. Parida (Eds.): CPM 2010, LNCS 6129, pp. 177–189, 2010.

only pick haplotypes out of this pool. We may even have more specific information about the permissible haplotypes for the genotypes of the individuals: the ethnicity of individuals may be known, allowing us to narrow the pool of permissible haplotypes for each individual. On the other hand, for some individuals no prior knowledge may be available.

In the present paper we study combinatorial haplotyping methods that take such *pool constraints* into account. For some or all genotypes we are given a pool of haplotypes that are allowed for this particular genotype. The task is to predict haplotypes for the genotypes such that all constraints are satisfied and the haplotypes form a perfect phylogeny or their number is minimal or both.

The above ideas lead to three mathematical problems, whose complexity we study in the present paper: $\text{C}_{\text{pools}}\text{PPH}$ is the *constrained perfect phylogeny haplotyping problem*, $\text{C}_{\text{pools}}\text{MH}$ is the *constrained maximum parsimony haplotyping problem*, and $\text{C}_{\text{pools}}\text{MPPH}$ is the combined problem (see Section 2 for formal definitions). The two problems $\text{C}_{\text{pools}}\text{PPH}$ and $\text{C}_{\text{pools}}\text{MH}$ are generalizations of the two problems $\text{C}_{\text{one pool for all}}\text{PPH}$ and $\text{C}_{\text{one pool for all}}\text{MH}$ recently studied by Fellows et al. [12]; the difference is that Fellows et al. require a single pool of haplotypes to be used for all genotypes while we allow pools to be specified individually for each genotype. We remark that, since we also allow that no constraints are imposed at all, the standard problems PPH, MH, and MPPH (without any constraints) are special cases of their constrained counterparts and the algorithms we present also work for them.

Our Results. Our first main result is a polynomial-time algorithm for $\text{C}_{\text{pools}}\text{PPH}$. It is based on an initial partition of the genotypes into independent subinstances and a subsequent recursive decomposition of the pool constraints. Since this algorithm also solves the simpler problem $\text{C}_{\text{one pool for all}}\text{PPH}$, we settle the main open problem of Fellow et al. [12]: $\text{C}_{\text{one pool for all}}\text{PPH}$ is polynomial-time solvable.

Our second set of results concerns maximum parsimony haplotyping. Both MH and $\text{C}_{\text{one pool for all}}\text{MH}$ are known to be NP-complete, but fixed-parameter tractable with respect to the number of distinct haplotypes in the solution [20,12]. We show that, in contrast, $\text{C}_{\text{pools}}\text{MH}$ is hard for the class W[2] for the same parameter and, therefore, unlikely to have a fixed-parameter algorithm. We prove this by showing that $\text{C}_{\text{pools for all}}\text{MH}$, where some pool *must* be specified for each genotype, is W[2]-complete. On the positive side we present a fixed-parameter algorithm for $\text{C}_{\text{pools}}\text{MH}$ where the parameter is the number of distinct haplotypes in the solution plus the number of times duplicated genotypes have incomparable pool constraints.

Our third main result is that the NP-complete problem $\text{C}_{\text{pools}}\text{MPPH}$ *is* fixed-parameter tractable with respect to the number of distinct haplotypes in the solution. So, $\text{C}_{\text{pools}}\text{MPPH}$ has the same complexity as $\text{C}_{\text{one pool for all}}\text{MH}$. As corollaries we obtain that MPPH and $\text{C}_{\text{one pool for all}}\text{MPPH}$ are both fixed-parameter tractable, which was not known before. Our algorithm is a combination of the algorithmic ideas for $\text{C}_{\text{pools}}\text{PPH}$ and $\text{C}_{\text{pools}}\text{MH}$.

We have implemented our polynomial-time algorithm for $\text{C}_{\text{pools}}\text{PPH}$. The implementation shows that the algorithm works very fast in practice. We have

also started applying it to real genotype data and plan to report on the results in a future publication. In the present paper, however, we concentrate on the algorithmic side.

Related Work. The study of the perfect-phylogeny haplotyping problem was initiated by the seminal paper of Gusfield [13], who showed that it is solvable in polynomial time. Subsequent papers presented conceptually simpler polynomial-time algorithms [2,10], linear-time algorithms [5,3,18,19], and fine-grained complexity-theoretic results [7,8] for it.

The problem MH is NP-complete, as remarked in [14], and a later publication sharpens this lower bound by showing that MH remains NP-complete if every given genotype has at most three heterozygous sites [17]. On the positive site Sharan, Halldórsson, and Istrail [20] devised a fixed-parameter algorithm for MH, where the parameter is the number of distinct haplotypes in the solution. Moreover, algorithms based on linear programming [4], branch-and-bound algorithms [22], and a recent combination of both methods [16] are known.

To increase the accuracy of the predicted haplotypes, the perfect phylogeny and the maximum parsimony assumptions have been combined, leading to the problem MPPH. It was shown to be NP-complete for instances with at most three heterozygous entries per genotypes by Bafna et al. [1] and later studied by Iersel et al. [21].

Another direction to increase prediction accuracy is to constrain the set of solution haplotypes: Fellows et al. [12] proposed the $\text{C}_{\text{one pool for all}}$PPH problem and presented polynomial-time algorithms for some special cases like the number of heterozygous entries in the genotypes and in the sites being bounded by small constants. They left open the complexity of $\text{C}_{\text{one pool for all}}$PPH and leaned towards the conjecture that it is NP-complete. The problem $\text{C}_{\text{one pool for all}}$MH is NP-complete by a reduction from MH with at most three heterozygous entries per genotypes (for each genotype put all its explaining haplotypes, of which there can be at most four, into the pool). Huang et al. [15] studied approximation algorithms for this problem, Fellows et al. [12] showed that it is fixed-parameter tractable with respect to the number of distinct haplotypes in the solution.

Organization of This Paper. We first give formal definitions of genotypes, haplotypes, and the computational problems we study. Sections 3, 4, and 5 are devoted to the algorithmic and complexity-theoretic studies of C_{pools}PPH, C_{pools}MH, and C_{pools}MPPH, respectively.

Due to lack of space, all proofs are omitted. They can be found in the technical report version of this paper [9].

2 Haplotyping Problems and Constraints

A *haplotype* describes the genetic information from a single chromosome at SNP sites. Since most SNP sites are biallelic, it is customary to encode a haplotype as a binary string $h \in \{0,1\}^n$, where 0 and 1 represent the two possible alleles. A genotype combines the genetic information of two haplotypes by joining

their entries to a sequence of sets. Following common conventions, instead of sets we write a 0 or a 1 when both underlying haplotypes have this value (these entries are called *homozygous*) and use the value 2 when the underlying haplotypes have different entries (these entries are called *heterozygous*). A pair of haplotypes $\{h, h'\} \subseteq \{0,1\}^n$ *explains* a genotype $g \in \{0,1,2\}^n$ if for every site $s \in \{1,\dots,n\}$ we have $g[s] = h[s] = h'[s]$ whenever $g[s] \in \{0,1\}$ and $h[s] \neq h'[s]$ whenever $g[s] = 2$. In a *genotype matrix* A each row is a genotype. If the matrix is clear from the context, we refer to the genotype in row i by g_i. Similar, we arrange haplotypes in a *haplotype matrix* B and refer to the haplotype in row i by h_i. A $2n \times m$ haplotype matrix B *explains* an $n \times m$ genotype matrix A if every genotype g_i is explained by the haplotype pair $\{h_{2i-1}, h_{2i}\}$. We use the term *site* to refer to a position in genotypes and haplotypes and to a column of genotype and haplotype matrices.

For a pair s and t of sites the *induced set* $\mathrm{ind}(B, s, t)$ contains all strings from $\{00, 01, 10, 11\}$ that appear in the sites s and t in the haplotype matrix B. We say that these strings are *induced* by s and t. The notion of induces can be extended to genotype matrices A: for two sites s and t the set $\mathrm{ind}(A, s, t)$ contains a string $xy \in \{00, 01, 10, 11\}$ if A has a genotype g with either $g[s] = x \wedge g[t] = y$ or $g[s] = x \wedge g[t] = 2$ or $g[s] = 2 \wedge g[t] = y$. This implies $\mathrm{ind}(A, s, t) \subseteq \mathrm{ind}(B, s, t)$ for every haplotype matrix B explaining A.

A haplotype matrix B *admits a perfect phylogeny* if there exists a tree T (an undirected acyclic graph), such that: (a) Each haplotype from B labels exactly one vertex of T; (b) each site $s \in \{1,\dots,m\}$ labels exactly one edge of T and each edge is labeled by at least one site; and (c) for every two haplotypes h_i and h_j from B and every site $s \in \{1,\dots,m\}$, we have $h_i[s] \neq h_j[s]$ if, and only if, s lies on the path from h_i to h_j in T. It is well-known that B admits a perfect phylogeny if, and only if, it satisfies the following *four gamete property: for every pair of sites s and t we have* $\{00, 01, 10, 11\} \neq \mathrm{ind}(B, s, t)$.

For the three problems PPH, MH, and MPPH the input is always a genotype matrix plus, for the last two problems, a number k. The questions are whether there exists a haplotype matrix B that explains A and admits a perfect phylogeny (PPH), has at most k different haplotypes (MH), or admits a perfect phylogeny and has at most k different haplotypes (MPPH).

Constrained Haplotyping Problems. For *constrained* haplotyping problems different kinds of *constraints* are specified along with the input genotype matrix. The first kind of constraints that we study are *pool constraints*. Let A be an $n \times m$ genotype matrix. A pool constraint specifies that, in the output haplotype matrix, the two explaining haplotypes for some particular genotype g_i should both be drawn from a pool $H_i \subseteq \{0,1\}^n$ of allowed haplotypes. We write such a constraint as $\mathrm{pool}(i, H_i)$. Clearly, it suffices to allow only one such constraint per genotype. Two pool constraints are *incomparable* if none of their pools is a subset of the other.

The second kind of constraints are restrictions on the phase of sites. For a genotype g with 2-entries in two sites s and t, the explaining haplotypes add either $\{00, 11\}$ or $\{01, 10\}$ to the induced set. If there is another genotype g'

with 2-entries in the sites s and t, then, in order to satisfy the four gamete property, it must choose the same pair for its explaining haplotypes. In the first case we say that s and t are *phased equally*, otherwise *phased unequally*. The constraints "equal-phase(s,t)" and "unequal-phase(s,t)" specify that a particular phasing must be chosen for the two sites s and t in a solution matrix. Formally, a haplotype matrix B satisfies equal-phase(s,t) if $\{01,10\} \not\subseteq \text{ind}(B,s,t)$; and unequal-phase$(s,t)$ if $\{00,11\} \not\subseteq \text{ind}(B,s,t)$.

We indicate constrained haplotyping problems by prefixing the haplotyping problems MH, PPH, and MPPH with a C whose index indicates which constraints are allowed to be specified as part of the input. The index "pools" means that arbitrary pool constraints are allowed; "pools for all" indicates that (possibly different) pools must be specified for all genotypes (and not only for some); and "one pool for all" indicates that, additionally, the same pool must be specified for all genotypes. The index "phase" indicates that phase constraints are permissible. For example, $\text{C}_{\text{pools,phase}}$MPPH is the MPPH problem where both haplotype and phase constraints are allowed as part of the input.

Haplotyping with phase constraints has not been defined formally in the literature, but many known algorithms implicitly handle phase constraints:

Fact 2.1 ([2,10]). *There exists an algorithm that, given an $n \times m$ genotype matrix with phase constraints, solves the problem* C_{phase}PPH *in time $O(nm^2)$.*

3 Constrained Perfect Phylogeny Haplotyping

In this section we prove the following theorem, which answers the main question of Fellows et al. [12] affirmatively: There is a polynomial-time algorithm for $\text{C}_{\text{one pool for all}}$PPH.

Theorem 3.1. *There exists an algorithm that solves* $\text{C}_{\text{pools,phase}}$PPH *in time $O(p(n+p)m^2)$, where the input genotype matrix has size $n \times m$ and p is the sum of the sizes of all pool constraints.*

The outline of the algorithm for $\text{C}_{\text{pools,phase}}$PPH, which we detail in the rest of this section, is as follows: Given an $n \times m$ genotype matrix A and a set K of pool and phase constraints, our algorithm uses procedure SOLVE-CPPH from Figure 1 to preprocesses the input and to partition the genotypes into at most m matrices A_s that can be solved independently. Each matrix A_s has the property that there is a site s, called the *2-site of A_s*, that has 2-entries in all genotypes from A_s. Each A_s along with its corresponding constraints is then solved by the procedure SOLVE-CPPH-2-SITE from Figure 1 via a recursive branch-and-reduce approach: For each of the two possible phasings between the 2-site and another site, it branches recursively, derives new phase constraints, and splits the pool constraints.

In the following we describe the four procedures that make up our algorithm: the two main procedures SOLVE-CPPH and SOLVE-CPPH-2-SITE, whose pseudo-code is depicted in Figure 1, and the simpler procedures SANITIZE-POOL-CONSTRAINTS and DEDUCE-PHASE-CONSTRAINTS for which no pseudo-code is

given. In the following, we say that a computational step *has the correctness property* if the following holds: *There exists a haplotype matrix that explains the genotype matrix and satisfies the four gamete property and the constraints before the step if, and only, if this holds for the instance after the step. Furthermore, whenever the step outputs "no", no solution exist for the current instance.*

Procedure SOLVE-CPPH(A, K).
Input: An $n \times m$ genotype matrix A and a set of constraints K
Output: An explaining haplotype matrix B for A that satisfies the four gamete property and the constraints K, if it exists; or "no", otherwise
Preprocessing:
ensure that column pairs with different entries induce 00
sort columns decreasingly by leaf count
update phase constraints with induces
call DEDUCE-PHASE-CONSTRAINTS
call SANITIZE-POOL-CONSTRAINTS
Solve independent subinstances:
for each site $s \in \{1, \ldots, m\}$ **do**
  $B_s \leftarrow$ **call** SOLVE-CPPH-2-SITE(A_s, K_s, s)
  if B_s is "no" **then return** "no"
return combination of matrices B_s and genotypes without 2-entries

Procedure SOLVE-CPPH-2-SITE(A, K, s_2).
Input: An $n \times m$ genotype matrix A with 2-site s_2 and a set of constraints K
Output: An explaining haplotype matrix B for A that satisfies the four gamete property and the constraints K, if it exists; or "no", otherwise
Recursion break:
if for every pool$(i, H_i) \in K$ we have $|H_i| = 2$ **then**
  replace all pool constraints by corresponding phase constraints
  return solution for the resulting $\mathrm{C}_{\mathrm{phase}}\mathrm{PPH}$ instance
Recursive branch-and-reduce:
else for each component G' of G_{cover} with corresponding instance A', K' **do**
  $s \leftarrow$ some site from G'
  $B'_e \leftarrow$ **call** TRY-PHASE-CPPH$(A', K' \cup \{\text{equal-phase}(s_2, s)\}, s_2)$
  $B'_u \leftarrow$ **call** TRY-PHASE-CPPH$(A', K' \cup \{\text{unequal-phase}(s_2, s)\}, s_2)$
  if $B'_e = B'_u =$ "no" **then return** "no" **else** add B'_e or B'_u to solution
return solution

Sub-Procedure TRY-PHASE-CPPH(A, K, s_2).
call DEDUCE-PHASE-CONSTRAINTS and SANITIZE-POOL-CONSTRAINTS for A, K
if pool$(i, \emptyset) \notin K$ for all i **then return** SOLVE-CPPH-2-SITE(A, K, s_2)
else return "no"

Fig. 1. The polynomial-time algorithm for $\mathrm{C}_{\mathrm{pools,phase}}\mathrm{PPH}$

Procedure SANITIZE-POOL-CONSTRAINTS. This procedure removes superfluous haplotypes from pool constraints. Let K be a set of constraints. First, for a constraint pool$(i, H_i) \in K$ and a genotype g_i, it removes all h from H_i for which there exists a site s such that $h[s] \neq g_i[s] \in \{0, 1\}$. Second, it deletes every

haplotype h from H_i for which there exists no other haplotype $h' \in H_i$ such that $\{h, h'\}$ explains g_i. Third, it deletes haplotypes contradicting phase constraints: For two sites s and t with $g_i[s] = g_i[t] = 2$, it deletes h from H_i whenever $h[s] = h[t] \wedge \text{unequal-phase}(s,t) \in K$ or $h[s] \neq h[t] \wedge \text{equal-phase}(s,t) \in K$. Finally, if a pool constraint becomes empty, it outputs "no." Clearly, this step has the correctness property.

Procedure DEDUCE-PHASE-CONSTRAINTS. Let A be a genotype matrix and K a set of constraints. The procedure repeats the following rule as long as possible: Let s, t and u be three sites such that there is a genotype g_i with $g_i[s] = g_i[t] = g_i[u] = 2$ and there is no phase constraint for the pair t and u, but phase constraints for both pairs s and t, and s and u. If these phase constraints have the same type, we insert equal-phase(t,u) into K and, if their type is different, we insert unequal-phase(t,u) into K. Using graph representations for phase constraints and their dependencies, the result of this procedure can be computed in time $O(nm^2)$ [2,10].

Lemma 3.2. DEDUCE-PHASE-CONSTRAINTS *has the correctness property.*

Procedure SOLVE-CPPH. The pseudo-code of this procedure is shown in Figure 1. We go over this method line by line.

The first five lines preprocess the input. Line 1 extends an idea from Eskin, Halperin and Karp [10] to constraints. For every site s we iterate downwards through the genotypes and if a 1-entry appears before a 0-entry, we substitute all 1-entries by 0-entries and vice versa and adjust the constraints accordingly. As shown in [10], this step ensures that any two sites with at least one different entry induce 00. In line 2 the procedure first calculates the *leaf count* [13] of each column, which is the number of 2-entries of a column plus twice the number of its 1-entries. Then it sorts the columns decreasingly from left to right by this value. After this sorting we have $10 \in \text{ind}(A,s,t)$ for every two sites s and t with different entries and $s < t$. This holds since otherwise there is no genotype with $g[s] = 1 \wedge g[t] \in \{0,2\}$ or $g[s] = 2 \wedge g[t] = 0$, but at least one genotype with $g[s] \in \{0,2\} \wedge g[t] = 1$ or $g[s] = 0 \wedge g[t] = 2$. This would imply that the leaf count of site t should be greater than the leaf count of site s, a contradiction. In line 3 the algorithm considers all pairs of sites s and t and updates their phase constraints as follows: If $\{00, 11\} \subseteq \text{ind}(A,s,t)$, it inserts equal-phase$(s,t)$ into K; and if $\{01, 01\} \subseteq \text{ind}(A,s,t)$, it inserts unequal-phase(s,t). This step has the correctness property since the new phase constraints reflect only induces that are already in the matrix. Finally, lines 4 and 5 deduce phase constraints and sanitize the pool constraints. In the following, we call a matrix that has undergone the preprocessing from lines 1 to 5 a *preprocessed genotype matrix.*

In lines 6 to 8 the genotype matrix A is partitioned genotype-wise into m submatrices $A_1, \ldots, A_m$, one matrix for each site. A genotype g belongs to the matrix A_s if $g[s] = 2$ and for every site $t < s$ we have $g[t] \neq 2$. Each A_s is passed along with the corresponding pool constraint and all phase constraints, stored in the set K_s, to a call of the procedure SOLVE-CPPH-2-SITE. The construction

of A_s ensures that site s has 2-entries in all genotypes from A_s. The effect of the partition is stated by the following lemma:

Lemma 3.3. *Let A be a preprocessed $n \times m$ genotype matrix with constraints K. Then there exists an explaining haplotype matrix B for A that satisfies the four gamete property and the constraints K if, and only if, for every site $s \in \{1, \ldots, m\}$ there exists an explaining haplotype matrix B_s for A_s that satisfies the four gamete property and the constraints K_s.*

Putting it altogether, SOLVE-CPPH correctly solves $\text{C}_{\text{pools,phase}}\text{PPH}$, provided that the procedure SOLVE-CPPH-2-SITE is correct, which we prove next.

Procedure SOLVE-CPPH-2-SITE. This procedure recursively solves the instances that are produced by SOLVE-CPPH, each consisting of a genotype matrix A with a 2-site s_2 and constraints K. The recursion stops when all pool constraints contain only two haplotypes (they must contain at least two haplotypes because a 2-entry is present in the genotype). In such a case the phasing of the genotype is completely known. We remove the pool constraints and, instead, add phase constraints that describe this particular phasing: For each constraint $\text{pool}(i, \{h, h'\})$ and sites s and t add the phase constraint $\text{equal-phase}(s, t)$ if $h[s] = h[t] \neq h'[s] = h'[t]$ and $\text{unequal-phase}(s, t)$ if $h[s] = h'[t] \neq h[t] = h'[s]$. The resulting instance of $\text{C}_{\text{phase}}\text{PPH}$ can be solved in polynomial time by Fact 2.1.

To describe the recursive step, we need some terminology. Let $\text{geno}_2(s)$ be the set of A's genotypes that have a 2-entry at site s. Let S_{free} be the set of sites s of A where $s \neq s_2$ and there is no phase constraint for s and s_2 in K. Let S_{cover} be the set of sites $s \in S_{\text{free}}$ for which there is no site $s' \in S_{\text{free}}$ with $\text{geno}_2(s) \subseteq \text{geno}_2(s')$; in the case that sites from S_{free} have the same set of 2-entries, we choose exactly one of them to be contained in S_{cover}. Note that when a genotype from A has a 2-entry in a site from S_{free}, then it also has a 2-entry a site from S_{cover}. Let G_{cover} be the graph that has S_{cover} as its vertex set and an edge between sites s and s' if $\text{geno}_2(s) \cap \text{geno}_2(s') \neq \emptyset$. Whenever there is an edge between sites in G_{cover}, then there exists a phase constraint for them.

In the recursive step the algorithm iterates over the components G' of G_{cover} and considers the submatrix A' of A made up by all genotypes with 2-entries in sites of G' along with a constraints set K', consisting of the pool constraints for the genotypes from A' and all phase constraints. It chooses a site s from G' and adds once the constraint $\text{equal-phase}(s_2, s)$ and once $\text{unequal-phase}(s_2, s)$ to the set of constraints. In each case, it checks which additional phase constraints are now triggered using the sub-procedure TRY-PHASE-CPPH. This sub-procedure calls DEDUCE-PHASE-CONSTRAINTS followed by SANITIZE-POOL-CONSTRAINTS and tries to solve the resulting instance recursively by calling SOLVE-CPPH-2-SITE. If for all components a recursive call returns a solution, the procedure combines them along with haplotypes for genotypes that are not in any matrix A' to a solution for the whole instance. The following lemma states the correctness of SOLVE-CPPH-2-SITE:

Lemma 3.4. *Let A be a preprocessed $n \times m$ genotype matrix with 2-site s_2 and constraints K. Then* SOLVE-CPPH-2-SITE *returns a haplotype matrix B that*

explains A and satisfies the four gamete property and the constraints K, if it exists, or "no", otherwise.

Runtime. The input to the algorithm consists of a genotype matrix of dimension $n \times m$, phase constraints and pool constraints. Let p equal the sum of the sizes of all pool constraints. We show that the runtime is $O(p(n+p)m^2)$, as claimed in Theorem 3.1. All individual operations of the algorithm take time at most $O((n+p)m^2)$. Thus, it suffice to show that the tree of recursive calls of procedure SOLVE-CPPH-2-SITE has at most p leafs: The procedure partitions its input matrix into submatrices with constraints. For every submatrix A' with constraints K' it may branch into two possible phasings for the sites s_2 and s. The call of DEDUCE-PHASE-CONSTRAINTS ensures that there are phase constraints between s_2 and all sites from G': when two sites are connected via an edge in G_{cover}, we know that there is a phase constraint for them and a genotype that contains 2-entries in these sites and s_2. Note that the phases between s_2 and the sites from G' for the case equal-phase(s_2, s) are exactly opposite to the phases for the case unequal-phase(s_2, s). This implies that, since all genotypes in A' have a 2-entry in s_2 and a site from G', every haplotype from the pool constraints *is passed to at most one recursive call.* This yields a partition of sets of haplotypes from the pool constraints among all recursive calls. Since the procedure stops when the sizes of the pools drop to two (or zero), the number of leafs of the recursive tree of procedure SOLVE-CPPH-2-SITE is bounded by $p + 1$.

We remark that we have implemented the algorithm in Java and applied it to laboratory data. Our prototypical implementation handles typical real-data inputs in a matter of seconds on a standard machine.

4 Constrained Maximum Parsimony Haplotyping

In this section, we present two results on the fixed-parameter tractability (see [6] for background in parametrized complexity theory) of the constrained maximum parsimony haplotyping problem. First, we prove that $\text{C}_{\text{pools for all}}\text{MH}$ is W[2]-complete when parametrized by the minimum number of distinct haplotypes in an explaining haplotype matrix. In sharp contrast, MH and $\text{C}_{\text{one pool for all}}\text{MH}$ are fixed-parameter tractable for this parameter, as shown in [20] and [12], respectively. This means that the possibility to specify pool constraints on a per-genotype basis vastly increases the complexity of the problem. Second, we show that a fixed-parameter algorithm is possible even for $\text{C}_{\text{pools}}\text{MH}$ when we extend the parameter to the number of distinct haplotypes plus the number of duplicated genotypes that have incomparable pools.

The algorithms for MH and $\text{C}_{\text{one pool for all}}\text{MH}$ from the literature use data structures that describe how haplotypes are shared among genotypes. Given an $n \times m$ genotype matrix A, we define a *haplotype sharing plan P for A of size k* as a multigraph $G = (V, E)$ (a graph with multiple edges between the same vertices) with $|V| = k$ and $|E| = n$ where (a) edges are labeled bijectively by genotypes from A, (b) some vertices are labeled by haplotypes, and (c) every genotype that

has two labeled incident vertices is explained by the haplotype labels. We call a plan *complete* if all vertices are labeled and *empty* if no vertex is labeled. A plan P *extends* a plan P' if P arises from P' by labeling previously unlabeled vertices. A haplotype sharing plan P *satisfies a pool constraint* $\text{pool}(i, H_i)$ if the incident haplotypes of g_i lie in H_i. With this definition, constructing haplotype matrices with at most k distinct haplotypes is equivalent to constructing plans of at most size k.

Given a budget k for the number of distinct haplotypes in the solution, the known fixed-parameter algorithms for MH and $\text{C}_{\text{one pool for all}}\text{MH}$ consider all possible empty haplotype sharing plans of size k and check whether they can be extended to complete ones in polynomial time, using GF[2] equations for MH [20] and dynamic-programming for $\text{C}_{\text{one pool for all}}\text{MH}$ [12]. To bound the number of edges of the plan they use a preprocessing step that deletes duplicated genotypes and retains only one of them. Since k haplotypes can explain at most $k(k-1)/2$ different genotypes, these algorithms consider at most $O(k^{2n}) \leq O(k^{2k^2})$ different empty plans.

These ideas cannot be extended to a fixed-parameter algorithm when genotype-specific pool constraints are given since we cannot delete duplicated genotypes in a preprocessing step. This is due to the fact that genotypes might have the same entries, but incomparable pools, which we cannot merge directly. Strong evidence that no slightly variation of the standard approaches will work is given by Theorem 4.1.

Theorem 4.1. $\text{C}_{\text{pools for all}}\text{MH}$*, parametrized by the number of distinct haplotypes in the solution, is* W[2]*-complete. Consequently,* $\text{C}_{\text{pools}}\text{MH}$ *is* W[2]*-hard for the same parametrization.*

The instances constructed in the W[2]-hardness proof of $\text{C}_{\text{pools for all}}\text{MH}$ contain only identical genotypes, namely completely heterozygous genotypes, while pools might be highly incomparable. Since such a worst case instance is unlikely to be present in practice, we propose to additionally parametrize the problem by the maximum number l of duplicated genotypes with pairwise incomparable pool constraints. When parametrized by the number k of distinct haplotypes and at the same time by l, $\text{C}_{\text{pools}}\text{MH}$ becomes fixed-parameter tractable.

Theorem 4.2. $\text{C}_{\text{pools}}\text{MH}$ *is fixed-parameter tractable with respect to the number of distinct haplotypes that are used in an explaining haplotype matrix plus the maximum number of duplicated genotypes with pairwise incomparable pool constraints.*

The algorithm (Figure 2 shows pseudo-code) first preprocesses the instance, such that at most l genotypes have the same entries. Then it iterates over at most $O(k^{2n}) \leq O(k^{lk^2})$ empty plans. After an initial check of whether a plan can be extended to a complete one without constraints, the algorithm considers every component of the plan independently. If a component contains genotypes having pool constraints, it picks one of these genotypes and an incident vertex and tries all assignments of permissible pool haplotypes to the vertex. A haplotype

Procedure SOLVE-CMH(A, K, k).
Input: An $n \times m$ genotype matrix A, pool constraints K and a budget k.
Output: An explaining haplotype matrix B for A with at most k distinct haplotypes that satisfies the constraints K, if it exists; or "no", otherwise.
Preprocessing:
1 **call** SANITIZE-POOL-CONSTRAINTS
2 **for each** g_i and g_j with $g_i = g_j$ **do**
3 **if** there is no pool constraint for g_j,
or pool$(i, H_i) \in K$, pool$(j, H_j) \in K$, $H_i \subseteq H_j$ **then**
4 delete g_j
5 **if** there are more than $lk(k-1)/2$ genotypes **then output** "no"
Try to extend empty haplotype sharing plans:
6 **for each** empty haplotype sharing plan P of size k **do**
7 **if** P cannot be extended to a complete plan **then** skip P
8 **for each** component P' of P **do**
9 **if** there is a genotype g_i in P' with pool$(i, H_i) \in K$ **then**
10 $v \leftarrow$ some vertex incident to g_i
11 **for each** haplotype $h \in H_i$ that is permissible for v in P **do**
12 $P'' \leftarrow P'$; label v with h in P'' and calculate haplotypes for all vertices
13 **if** P'' is a haplotype sharing plan satisfying its pool constraints **then**
14 store P'' as a solution for P' and **continue** with next P'
15 **skip** P
16 **else** choose a permissible haplotype for one vertex from P',
calculate haplotypes for all other vertices, and store the solution P''
17 combine all P'' to a plan for A and K and **return** combined plan
18 **output** "no"

Fig. 2. The fixed-parameter algorithm for $\text{C}_{\text{pools}}\text{MH}$

is permissible for a vertex in a plan if there exists an extending complete plan with the vertex labeled by this haplotype. Conversely, its assignment directly determines haplotypes for all other vertices in the component and it remains to check that these haplotypes satisfy the other pool constraints. For components without constraints, an assignment of haplotypes is always possible, due to line 7. Since the inner part of the main loop needs only polynomial time, this gives the desired fixed-parameter runtime.

5 Constrained Maximum Parsimony Perfect Phylogeny Haplotyping

We show that $\text{C}_{\text{pools,phase}}\text{MPPH}$ and, therefore, MPPH and $\text{C}_{\text{one pool for all}}\text{MPPH}$, are fixed-parameter tractable with respect to the number of distinct haplotypes in the solution.

Theorem 5.1. $\text{C}_{\text{pools,phase}}\text{MPPH}$ *is fixed-parameter tractable with respect to the number of distinct haplotypes in the solution.*

We prove the theorem by combining the recursive branch-and-reduce technique from Section 3 with haplotype sharing plans, which control the size of solutions.

The algorithm first ensures that the input contains no duplicate genotypes. Then it iterates over at most $O(k^{2n}) \leq O(k^{2k^2})$ empty plans. In every iteration it computes the partition from SOLVE-CPPH and solves the matrices independently. For this, it also decomposes the current plan such that a subplan is made up by the edges from the genotypes of its corresponding submatrix. To ensure that different instances are not related through vertices in the plan, the algorithm labels all vertices that are incident to genotypes from different submatrices with haplotypes. In the second main part, a recursive branch-and-reduce procedure, instances are partitioned such that the different parts are neither related through the matrix nor through the plan. For each part, similar to procedure SOLVE-CPPH-2-SITE, the algorithm branches into different phases between two columns. After labeling some vertices in the plan and sanitizing pool constraints, the algorithm solves completely independent matrices recursively. The iteration over at most $O(k^{2k^2})$ plans and the polynomial-time recursion give the desired fixed-parameter runtime.

6 Conclusion

We studied phylogeny- and parsimony-based haplotype inference in the presence of pool and phase constraints. Our main result is that $\text{C}_{\text{pools,phase}}\text{PPH}$ is polynomial-time solvable by a new recursive decomposition technique for genotypes and pools. This solves the question from [12] whether $\text{C}_{\text{one pool for all}}\text{PPH}$ is polynomial-time solvable. Our Java implementation of this algorithm shows that it works fast in practice. We showed that $\text{C}_{\text{pools}}\text{MH}$ is W[2]-hard by proving that $\text{C}_{\text{pools for all}}\text{MH}$ is W[2]-complete when parametrized by the number of distinct haplotypes in the solution. Both problems are fixed-parameter tractable when we also use the comparability of the pools as a parameter. For $\text{C}_{\text{pools,phase}}\text{MPPH}$ we presented an algorithm that extends the recursive decomposition of genotypes and pools by a decomposition of haplotype sharing plans, yielding a fixed-parameter algorithm for $\text{C}_{\text{pools,phase}}\text{MPPH}$ with respect to the number of distinct haplotypes in the solution.

For future work one research direction would be to incorporate more general constraints, like, for example, $*$-constraints where some entries in the haplotypes can be chosen freely. We may also try to allow a few additional rare haplotypes to be used that are not in any pool. A second direction would be to adjust the ideas to algorithms that work on incomplete data.

References

1. Bafna, V., Gusfield, D., Hannenhalli, S., Yooseph, S.: A note on efficient computation of haplotypes via perfect phylogeny. J. Comput. Biol. 11(5), 858–866 (2004)
2. Bafna, V., Gusfield, D., Lancia, G., Yooseph, S.: Haplotyping as perfect phylogeny: A direct approach. J. of Comput. Biol. 10(3–4), 323–340 (2003)
3. Bonizzoni, P.: A linear-time algorithm for the perfect phylogeny haplotype problem. Algorithmica 48(3), 267–285 (2007)

4. Brown, D.G., Harrower, I.M.: Integer programming approaches to haplotype inference by pure parsimony. IEEE/ACM T. on Comput. Biol. and Bioinf. 3(2), 141–154 (2006)
5. Ding, Z., Filkov, V., Gusfield, D.: A linear-time algorithm for the perfect phylogeny haplotyping (PPH) problem. J. Comput. Biol. 13(2), 522–553 (2006)
6. Downey, R.G., Fellows, M.R.: Parameterized Complexity. Springer, New York (1999)
7. Elberfeld, M.: Perfect phylogeny haplotyping is complete for logspace. Computing Research Repository (CoRR), abs/0905.0602 (2009)
8. Elberfeld, M., Tantau, T.: Computational complexity of perfect-phylogeny-related haplotyping problems. In: Ochmański, E., Tyszkiewicz, J. (eds.) MFCS 2008. LNCS, vol. 5162, pp. 299–310. Springer, Heidelberg (2008)
9. Elberfeld, M., Tantau, T.: Phylogeny- and parsimony-based haplotype inference with constraints. Technical Report SIIM-TR-A-10-01, Universität zu Lübeck (2010)
10. Eskin, E., Halperin, E., Karp, R.M.: Efficient reconstruction of haplotype structure via perfect phylogeny. J. Bioinf. and Comput. Biol. 1(1), 1–20 (2003)
11. Excoffier, L., Slatkin, M.: Maximum-likelihood estimation of molecular haplotype frequencies in a diploid population. Mol. Biol. and Evol. 12(5), 921–927 (1995)
12. Fellows, M.R., Hartman, T., Hermelin, D., Landau, G.M., Rosamond, F.A., Rozenberg, L.: Haplotype inference constrained by plausible haplotype data. In: Kucherov, G., Ukkonen, E. (eds.) CPM 2009. LNCS, vol. 5577, pp. 339–352. Springer, Heidelberg (2009)
13. Gusfield, D.: Haplotyping as perfect phylogeny: Conceptual framework and efficient solutions. In: Proc. of RECOMB 2002, pp. 166–175. ACM Press, New York (2002)
14. Gusfield, D.: Haplotype inference by pure parsimony. In: Baeza-Yates, R., Chávez, E., Crochemore, M. (eds.) CPM 2003. LNCS, vol. 2676, pp. 144–155. Springer, Heidelberg (2003)
15. Huang, Y.-T., Chao, K.-M., Chen, T.: An approximation algorithm for haplotype inference by maximum parsimony. J. Comput. Biol. 12(10), 1261–1274 (2005)
16. Jager, G., Climer, S., Zhang, W.: Complete parsimony haplotype inference problem and algorithms. In: Fiat, A., Sanders, P. (eds.) ESA 2009. LNCS, vol. 5757, pp. 337–348. Springer, Heidelberg (2009)
17. Lancia, G., Pinotti, M.C., Rizzi, R.: Haplotyping populations by pure parsimony: Complexity of exact and approximation algorithms. INFORMS J. on Comput. 16(4), 348–359 (2004)
18. Liu, Y., Zhang, C.-Q.: A linear solution for haplotype perfect phylogeny problem. In: Proc. Int. Conf. Adv. in Bioinf. and Appl., pp. 173–184. World Scientific, Singapore (2005)
19. Satya, R.V., Mukherjee, A.: An optimal algorithm for perfect phylogeny haplotyping. J. Comput. Biol. 13(4), 897–928 (2006)
20. Sharan, R., Halldórsson, B.V., Istrail, S.: Islands of tractability for parsimony haplotyping. IEEE/ACM T. Comput. Biol. and Bioinf. 3(3), 303–311 (2006)
21. van Iersel, L., Keijsper, J., Kelk, S., Stougie, L.: Shorelines of islands of tractability: Algorithms for parsimony and minimum perfect phylogeny haplotyping problems. IEEE/ACM T. Comput. Biol. and Bioinf. 5(2), 301–312 (2008)
22. Wang, L., Xu, Y.: Haplotype inference by maximum parsimony. Bioinformatics 19(14), 1773–1780 (2003)

Faster Computation of the Robinson-Foulds Distance between Phylogenetic Networks

Tetsuo Asano[1], Jesper Jansson[2], Kunihiko Sadakane[3], Ryuhei Uehara[1], and Gabriel Valiente[4]

[1] School of Information Science, Japan Advanced Institute of Science and Technology, Ishikawa 923-1292, Japan
{t-asano,uehara}@jaist.ac.jp
[2] Ochanomizu University, 2-1-1 Otsuka, Bunkyo-ku, Tokyo 112-8610, Japan
jesper.jansson@ocha.ac.jp
[3] National Institute of Informatics, Hitotsubashi 2-1-2, Chiyoda-ku, Tokyo 101-8430, Japan
sada@nii.ac.jp
[4] Algorithms, Bioinformatics, Complexity and Formal Methods Research Group, Technical University of Catalonia, E-08034 Barcelona, Spain
valiente@lsi.upc.edu

Abstract. The Robinson-Foulds distance, which is the most widely used metric for comparing phylogenetic trees, has recently been generalized to phylogenetic networks. Given two networks N_1, N_2 with n leaves, m nodes, and e edges, the Robinson-Foulds distance measures the number of clusters of descendant leaves that are not shared by N_1 and N_2. The fastest known algorithm for computing the Robinson-Foulds distance between those networks runs in $O(m(m+e))$ time. In this paper, we improve the time complexity to $O(n(m+e)/\log n)$ for general networks and $O(n\,m/\log n)$ for general networks with bounded degree, and to optimal $O(m+e)$ time for planar phylogenetic networks and bounded-level phylogenetic networks. We also introduce the natural concept of the minimum spread of a phylogenetic network and show how the running time of our new algorithm depends on this parameter. As an example, we prove that the minimum spread of a level-k phylogenetic network is at most $k+1$, which implies that for two level-k phylogenetic networks, our algorithm runs in $O((k+1)(m+e))$ time.

1 Introduction

The Robinson-Foulds distance, introduced in [17], has been the most widely used metric over almost three decades for comparing phylogenetic trees. However, it is now known that the evolutionary history of life cannot be properly represented as a phylogenetic tree [7], and phylogenetic networks have emerged as the representation of choice for incorporating reticulate evolutionary events, like recombination, hybridization, or lateral gene transfer, in an evolutionary history [16].

A. Amir and L. Parida (Eds.): CPM 2010, LNCS 6129, pp. 190–201, 2010.

Phylogenetic networks are directed acyclic graphs with *tree nodes* (those with at most one parent) corresponding to point mutation events and *hybrid nodes* (with more than one parent) corresponding to hybrid speciation events. As in the case of phylogenetic trees, the leaves are distinctly labeled by a set of extant species. Additional conditions are usually imposed on these directed acyclic graphs to narrow down the output space of reconstruction algorithms [13,14] or to provide a realistic model of recombination [19,20].

Two such additional conditions are especially relevant to the Robinson-Foulds distance. A phylogenetic network is *time consistent* when it has a temporal representation [1], that is, an assignment of discrete time stamps to the nodes that increases from parents to tree children and remains the same from parents to hybrid children, meaning that the parents of each hybrid node coexist in time and thus, the corresponding reticulate evolutionary event can take place. A phylogenetic network is *tree-child* when every internal node has at least one tree child [4], meaning that every non-extant species has some extant descendant through mutation alone.

The Robinson-Foulds distance between two phylogenetic networks is defined as the cardinality of the symmetric difference between their two sets of all induced clusters (where the cluster induced by a node v in a phylogenetic network is the set of all descendant leaves of v in the network) divided by two, and thus it measures the number of clusters not shared by the networks. It is a metric on the space of all tree-child time-consistent phylogenetic networks [4, Cor. 1], and it generalizes the Robinson-Foulds distance between rooted phylogenetic trees. Clearly, the Robinson-Foulds distance requires computing the cluster representation of the networks, that is, the set of descendant leaves for each node in the networks. While there are improved algorithms for computing the cluster representation of a phylogenetic tree [6,15,21,22], the only known algorithm for computing the cluster representation of a phylogenetic network [3] is based on breadth-first searching descendant leaves from each of the nodes in turn, and takes $O(m(m+e))$ time using $O(n\,m)$ space on phylogenetic networks with n leaves, m nodes, and e edges.

In this paper, we present a faster algorithm for computing the Robinson-Foulds distance between two input phylogenetic networks. For general phylogenetic networks, we first improve the time complexity by following an approach similar in spirit to the algorithm proposed in [4] for computing the path multiplicity representation of a phylogenetic network; by using a compressed representation of the characteristic vectors, we obtain a simple algorithm for computing the Robinson-Foulds distance between phylogenetic networks in $O(n(m+e)/\log n)$ time using $O(n\,m/\log n)$ space, assuming a word size of $\omega = \lceil \log n \rceil$ bits; see [12]. For phylogenetic networks of bounded degree, this becomes $O(n\,m/\log n)$ time and space.

In the case of level-k phylogenetic networks [5], we further improve the time complexity by using a more succinct representation of a cluster of descendant leaves as an interval of consecutive integers, which allows us to compute the Robinson-Foulds distance in $O((k+1)(m+e))$ time. For this purpose, we introduce what

we call the *minimum spread* of a phylogenetic network, and prove that every level-k network has minimum spread at most $k+1$. For special cases of bounded-level phylogenetic networks such as planar phylogenetic networks, in particular outer-labeled planar split networks [2,8] and galled-trees [10,11], we show that the minimum spread is 1, which means that our algorithm can be implemented to run in optimal $O(m+e)$ time.

The paper is organized as follows. Section 2 introduces some notation and explains the naive representation of clusters. Section 3 describes more efficient ways to represent the clusters both for general networks and for planar and level-k networks, and defines the *minimum spread* of a phylogenetic network. A bottom-up algorithm for computing the Robinson-Foulds distance is presented in Section 4 that takes advantage of the cluster representation. Finally, some conclusions are drawn in Section 5.

2 Preliminaries

Let $N = (V, E)$ be a given phylogenetic network with n leaves, m nodes, and e edges. For any nodes $u, v \in V$, we say that v is a *descendant* of u if v is reachable from u in N. (Here, any node is considered to be a descendant of itself.) For every $v \in V$, define $C[v]$ as the set of all leaves which are descendants of v. The set $C[v]$ is called the *cluster* of v, and the collection $\{C[v] \mid v \in V\}$ is called the *naive cluster representation* of N.

The naive cluster representation of N can be computed in $O(m(m+e))$ time and $O(n\,m)$ space by breadth-first searching descendant leaves from each of the nodes of N in turn [3]. A significant improvement in time complexity can be achieved by replacing the m top-down searches by n bottom-up searches, because m can be arbitrarily large for a phylogenetic network with n leaves and, even in the particular case of a tree-child time-consistent phylogenetic network, $m \leqslant (n+4)(n-1)/2$, and this bound is tight [3, Prop. 1]. The following lemma is the basis of such an improvement.

Lemma 1. *Let $v \in V$ be a node of a phylogenetic network $N = (V, E)$. Then, $C[v] = \{v\}$ if v is a leaf, and $C[v] = C[v_1] \cup \cdots \cup C[v_k]$ if v is an internal node with children $\{v_1, \ldots, v_k\}$.*

Proof. The only (trivial) descendant of a leaf in a phylogenetic network is the leaf itself. The paths from an internal node to the leaves of a phylogenetic network are the paths from the children of the internal node to the leaves. □

Lemma 1 suggests a simple bottom-up algorithm (Algorithm 1) for computing the naive cluster representation of N in polynomial time. In the following description, the cluster $C[v]$ of each node v in N is computed during a bottom-up traversal of N, with the help of an (initially empty) queue Q of nodes. The cluster $C[v]$ of each child v of an internal node u is joined in turn to the (initially empty) cluster $C[u]$ of the parent node u.

Algorithm 1. Compute the naive cluster representation C of a phylogenetic network N

```
procedure naive_cluster_representation(N, C)
    for each node v of N do
        if v is a leaf then
            C[v] ← {label(v)}
            enqueue(Q, v)
        else
            C[v] ← ∅
    while Q is not empty do
        v ← dequeue(Q)
        mark node v as visited
        for each parent u of node v do
            C[u] ← C[u] ∪ C[v]
            if all children of u are visited then
                enqueue(Q, u)
```

Lemma 2. *Let N be a phylogenetic network with n leaves, m nodes, and e edges. The naive cluster representation of N can be computed in $O(n(m+e))$ time using $O(n\,m)$ space.*

Proof. Each node is enqueued and dequeued only once, and each parent of each dequeued node v is visited only once from v. The union of two subsets of an n element set, which takes $O(n)$ time, is computed $O(m + e)$ times. □

3 More Efficient Cluster Representation

3.1 Characteristic Vector Representation

A *leaf numbering function* is a bijection from the set of leaves in N to the set $\{1, 2, \ldots, n\}$. For any leaf numbering function f and node $v \in V$, the *characteristic vector* for v under f, denoted by $C_f[v]$, is a bit vector of length n such that for any $i \in \{1, 2, \ldots, n\}$, the ith bit equals 1 if and only if $f^{-1}(i)$ is a descendant of v in N. Note that $C_f[r] = 111\ldots1$ for the root r of N, and that $C_f[\ell]$ contains exactly one 1 for any leaf ℓ of N.

Example 1. Consider the phylogenetic network in Figure 1. Number the leaves according to the circular ordering $v_2, v_{20}, v_{27}, v_{28}, v_{34}, v_{45}, v_{42}, v_{43}, v_{46}, v_{47}, v_{10}, v_{22}, v_{23}, v_{36}, v_{37}, v_{33}, v_{12}, v_{18}, v_{25}, v_{26}, v_{14}$ along the outer face. This corresponds to a depth-first search of the directed spanning tree obtained by removing one incoming edge (shown in red in Figure 1) for each node of in-degree 2 in the network, and it yields the characteristic vectors listed in Table 1. □

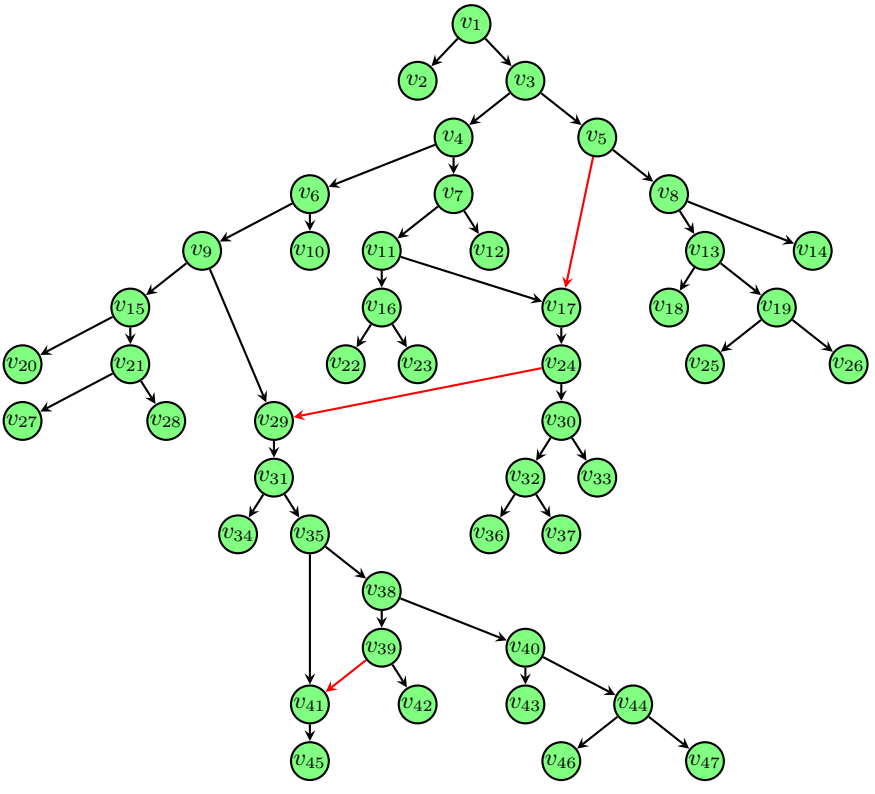


Fig. 1. An example of a phylogenetic network based on real data, adapted from [23]. This is the smallest level-2 phylogenetic network consistent with 1,330 rooted triplets of sequences from different isolates of the yeast *Cryptococcus gattii*.

Obviously, the characteristic vector representation of all clusters in N can be stored explicitly using a total of $m\,n$ bits and can be constructed in $O(n(m+e))$ time by an algorithm analogous to Algorithm 1. Our next goal is to find suitable leaf numbering functions for different types of phylogenetic networks which lead to more compact ways of storing the characteristic vectors as well as faster ways of computing them. We first consider arbitrary leaf numbering functions, and then study leaf numbering functions for some important special classes of phylogenetic networks.

3.2 Compressed Characteristic Vector Representation

Fix any arbitrary leaf numbering function f for the given phylogenetic network N. The time complexity of Algorithm 1 can be improved by employing a characteristic vector of size n to encode each cluster, packing the characteristic vector of a subset of the n leaves into $O(n/\log n)$ integers (assuming a word

Table 1. Characteristic vector representation of the clusters for the phylogenetic network in Figure 1

node	characteristic vector of the cluster																				
	v_2	v_{20}	v_{27}	v_{28}	v_{34}	v_{45}	v_{42}	v_{43}	v_{46}	v_{47}	v_{10}	v_{22}	v_{23}	v_{36}	v_{37}	v_{33}	v_{12}	v_{18}	v_{25}	v_{26}	v_{14}
v_{21}	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
v_{15}	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
v_{41}	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
v_{39}	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
v_{44}	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0
v_{40}	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0
v_{38}	0	0	0	0	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0
v_{35}	0	0	0	0	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0
v_{31}	0	0	0	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0
v_{29}	0	0	0	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0
v_9	0	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0
v_6	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
v_{16}	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0
v_{32}	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0
v_{30}	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0
v_{24}	0	0	0	0	1	1	1	1	1	1	0	0	0	1	1	1	0	0	0	0	0
v_{17}	0	0	0	0	1	1	1	1	1	1	0	0	0	1	1	1	0	0	0	0	0
v_{11}	0	0	0	0	1	1	1	1	1	1	0	1	1	1	1	1	0	0	0	0	0
v_7	0	0	0	0	1	1	1	1	1	1	0	1	1	1	1	1	1	0	0	0	0
v_4	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0
v_{19}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0
v_{13}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0
v_8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1
v_5	0	0	0	0	1	1	1	1	1	1	0	0	0	1	1	1	0	1	1	1	1
v_3	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
v_1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

size of $\omega = \lceil \log n \rceil$ bits), and computing the bitwise-OR of vectors instead of performing the set union operation. See [12] for further details about bit-level parallelism.

The pseudocode for the improved version of Algorithm 1 is given in Algorithm 2, where $x \ll t$ denotes the bitwise shift of an integer x to the left by t, and $x \mid y$ denotes the bitwise OR of two integers x and y.

The improvement in time complexity of Algorithm 2 comes from bit-level parallelism of the set union operations.

Lemma 3. *Let N be a phylogenetic network with n leaves, m nodes, and e edges. The cluster representation of N can be computed in $O(n(m+e)/\log n)$ time using $O(n^2 + n\,m/\log n)$ words.*

Proof. There are $2^{\log n} = n$ bit vectors, and the bitwise-OR of all these ω-bit vectors takes $O(n^2)$ time. After this prepreprocessing, each node is enqueued and dequeued only once, and each parent of each dequeued node v is visited only once from v. The union of two subsets of an n element set, which takes $O(n/\log n)$ time as the bitwise-OR of $\lceil n/\omega \rceil$ ω-bit vectors, is computed $O(m+e)$ times.

The bitwise-OR of all the ω-bit vectors is stored in $O(n^2)$ words, and the cluster representation is stored as a compact boolean table, with m rows and $n/\log n$ columns. □

Algorithm 2. Compute the compressed cluster representation C of a phylogenetic network N

```
procedure compressed_cluster_representation(N, C)
    n ← number of leaves of N
    k ← ⌈n/ω⌉
    for x ← 0, ..., n − 1 do
        for y ← 0, ..., n − 1 do
            OR[x, y] ← x | y
    for each node v of N do
        C_1[v], ..., C_k[v] ← 0
        if v is a leaf then
            i ← ⌊(f(v) − 1)/ω⌋ + 1
            C_i[v] ← 1 ≪ ω · i − f(v)
            enqueue(Q, v)
    while Q is not empty do
        v ← dequeue(Q)
        mark node v as visited
        for each parent u of node v do
            for i ← 1, ..., k do
                C_i[u] ← OR[C_i[u], C_i[v]]
            if all children of u are visited then
                enqueue(Q, u)
```

3.3 Interval List Representation

A maximal consecutive sequence of 1's in a bit vector is called an *interval*. For a given leaf numbering function f and node $v \in V$, let $I_f(v)$ denote the number of intervals in $C_f[v]$ and let the *spread of* f be $I_f = \max_{v \in V} I_f(v)$. The *minimum spread of* N is the minimum value of I_f, taken over all possible leaf numbering functions f.

Below, we first bound the minimum spread of certain types of phylogenetic networks, and then show more generally how the characteristic vectors of phylogenetic networks having small minimum spread can be stored compactly. From here on, we only consider phylogenetic networks in which each node has either at most one parent (tree node) or exactly two parents (hybrid node).

A phylogenetic network is *planar* if the underlying undirected graph is outer-labeled planar, that is, if it admits a non-crossing layout on the plane with all the leaves lying on the outer face. Planar phylogenetic networks arise for instance when representing conflicting phylogenetic signals, leading to the so-called outer-labeled planar split networks; see [2,9].

Lemma 4. *If N is a planar phylogenetic network then a leaf numbering function f with $I_f = 1$ can be computed in $O(m + e)$ time.*

Proof. Fix any planar embedding of N and let f be the leaf numbering function that assigns the numbers $1, 2, \ldots, n$ to the leaves in consecutive order along the outer face from the leftmost to the rightmost leaf. We claim that for every $v \in V$, $C_f[v]$ has a single interval. Since every leaf has a singleton cluster and the union of two overlapping or neighboring intervals is a single interval, we need to show that the children of any internal node have overlapping or neighboring clusters of descendant leaves.

Let $v \in V$ be an internal node with children $u, w \in V$ and assume $C[u] = \{h, \ldots, i\}$ and $C[w] = \{\ell, \ldots, m\}$ are intervals of descendant leaves with $h \leqslant i < j \leqslant k < \ell \leqslant m$ but $j, \ldots, k \notin C[v]$. Then, any path from the root of N to any of the leaves $j, \ldots, k$ will cross some edge along either a path from v to i or a path from v to ℓ, contradicting the assumption that N is planar. Therefore, $j, \ldots, k \in C[v]$ and the set $\{h, \ldots, i, j, \ldots, k, \ell, \ldots, m\}$ of descendant leaves forms one interval. □

Next, let $\mathcal{U}(N)$ denote the undirected graph obtained by replacing every directed edge in N by an undirected edge. A *biconnected component* of an undirected graph is a connected subgraph that remains connected after removing any node and all edges incident to it; see [18]. Recall the following definition from [5].

Definition 1. *A network N is called* level-k phylogenetic network *if, for every biconnected component B in $\mathcal{U}(N)$, the subgraph of N induced by the set of nodes in B contains at most k nodes with indegree 2.*

Corollary 1. *If N is a level-1 phylogenetic network (that is, a galled-tree [10,11]), then a leaf numbering function f with $I_f = 1$ can be computed in $O(m + e)$ time.*

Proof. Since each biconnected component of N forms a cycle and all the cycles in N are disjoint, the outside of an embedding of a cycle into a plane lies on the outer-plane. Then, it is obvious that $I_f = 1$. □

Lemma 5. *If N is a level-k phylogenetic network then a leaf numbering function f with $I_f = k + 1$ can be computed in $O(m + e)$ time.*

Proof. Fix any (directed) spanning tree T of N, and let f be the leaf numbering function obtained by doing a depth-first search of T starting at the root and assigning the numbers $1, 2, \ldots, n$ to the leaves in the order that they are first visited. Clearly, this takes $O(m + e)$ time.

We now prove that f has spread $k+1$. For any node v in V, define $L(T[v])$ as the set of all leaves in the subtree of T rooted at v. The key observation is that the leaves in $L(T[v])$ must be visited consecutively by any depth-first search of T, and thus form a single interval in $C_f[v]$. Next, let u be any node in V and let H be the set of hybrid nodes in N that belong to the same biconnected component as u and which are descendants of u (in case u is not on any merge path then H is the empty set). Then, the set of leaves that are descendants of u in N can

Table 2. Interval list representation of the clusters for the phylogenetic network in Figure 1

node	interval list
v_{21}	(v_{27}, v_{28})
v_{15}	(v_{20}, v_{28})
v_{41}	(v_{45}, v_{45})
v_{39}	(v_{45}, v_{42})
v_{44}	(v_{46}, v_{47})
v_{40}	(v_{43}, v_{47})
v_{38}	(v_{45}, v_{47})
v_{35}	(v_{45}, v_{47})
v_{31}	(v_{34}, v_{47})
v_{29}	(v_{34}, v_{47})

node	interval list
v_9	(v_{20}, v_{47})
v_6	(v_{20}, v_{10})
v_{16}	(v_{22}, v_{23})
v_{32}	(v_{36}, v_{37})
v_{16}	(v_{22}, v_{23})
v_{32}	(v_{36}, v_{37})
v_{30}	(v_{36}, v_{33})
v_{24}	$(v_{34}, v_{47}), (v_{36}, v_{33})$
v_{17}	$(v_{34}, v_{47}), (v_{36}, v_{33})$
v_{11}	$(v_{34}, v_{47}), (v_{22}, v_{33})$

node	interval list of the cluster
v_7	$(v_{34}, v_{47}), (v_{22}, v_{12})$
v_4	(v_{20}, v_{12})
v_{19}	(v_{25}, v_{26})
v_{13}	(v_{18}, v_{26})
v_8	(v_{18}, v_{14})
v_5	$(v_{34}, v_{47}), (v_{36}, v_{33}), (v_{18}, v_{14})$
v_3	(v_{20}, v_{14})
v_1	(v_2, v_{14})

be written as $L(T[u]) \cup \bigcup_{h \in H} L(T[h])$. N is a level-k phylogenetic network, so $|H| \leqslant k$, which together with the key observation above implies that $C_f[u]$ is the union of at most $k+1$ intervals. It follows that $I_f(u) \leqslant k+1$ for every $u \in V$. □

Example 2. Consider again the phylogenetic network in Figure 1. The leaf numbering in Example 1 yields the interval lists listed in Table 2. The network is level-2 and its spread corresponds to the 3 disjoint intervals $(v_{34}, v_{47}), (v_{36}, v_{33})$, (v_{18}, v_{14}) of node v_5. □

Now, we consider how to store characteristic vectors under leaf numbering functions having small spread. An efficient approach is to store the starting and ending positions of all intervals in sorted order. We call this representation the *interval list representation* of the clusters. We immediately obtain the following result.

Lemma 6. *Given any leaf numbering function f, the total space needed to store all characteristic vectors under f using the interval list representation is $O(I_f\, m \log n)$ bits.*

Proof. For each of the m nodes in N, the starting and ending positions of each of its at most I_f intervals are stored in $\lceil 2 \log n \rceil$ bits. □

Lemma 7. *Given any leaf numbering function f, all descendant leaf bit vectors under f using the interval list representation can be computed in $O(I_f(m+e))$ time.*

Proof. Use Algorithm 1 but replace the union operation as follows. Let v be an internal node with children u, w. Assuming that $C_f[u]$ and $C_f[w]$ are known, $C_f[v]$ can be computed in $O(I_f)$ time by a straightforward algorithm which scans the two sorted position lists for $C_f[u]$ and $C_f[w]$ and merges any intervals which overlap or are neighbors. □

4 An Algorithm for Computing the Robinson-Foulds Distance

We now present an algorithm for computing the Robinson-Foulds distance between two input phylogenetic networks N_1, N_2 (Algorithm 3).

The algorithm first computes the clusters of N_1 and N_2 using any of the cluster representations described in the previous sections of this paper. Then, the cardinality of the symmetric difference of the two cluster representations is obtained by radix sorting and simultaneous traversal techniques. Finally, the algorithm outputs the Robinson-Foulds distance between N_1 and N_2.

Algorithm 3. Compute the Robinson-Foulds distance between two phylogenetic networks N_1, N_2

```
function robinson_foulds_distance(N1, N2)
    cluster_representation(N1, C1); radix sort C1
    cluster_representation(N2, C2); radix sort C2
    m1, m2 ← number of nodes of N1, N2
    i1 ← 1
    i2 ← 1
    c ← 0
    while i1 ⩽ m1 and i2 ⩽ m2 do
        if C1[i1] < C2[i2] then
            i1 ← i1 + 1
        else if C1[i1] > C2[i2] then
            i2 ← i2 + 1
        else
            i1 ← i1 + 1
            i2 ← i2 + 1
            c ← c + 1
    return m1 + m2 − 2 · c
```

Theorem 1. *Let N_1, N_2 be two phylogenetic networks with n leaves, m nodes, and e edges. The Robinson-Foulds distance between N_1, N_2 can be computed in:*

- $O(n(m+e)/\log n)$ *time and* $O(n^2 + n\,m/\log n)$ *words for general networks,*
- $O(n\,m/\log n)$ *time and* $O(n^2 + n\,m/\log n)$ *words for general networks with bounded degree,*
- $O(m+e)$ *time and* $O(m \log n)$ *bits for planar phylogenetic networks,*
- $O((k+1)(m+e))$ *time and* $O(k\,m \log n)$ *bits for level-k phylogenetic networks.*

Proof. Implement Algorithm 3 by applying Lemmas 3–7 to obtain the respective cluster representations. The radix sort step and remaining operations can be performed in $O(m\,x)$ time, where x denotes the amount of space needed to represent one cluster. □

5 Conclusion

We have presented a new and simple algorithm for computing the Robinson-Foulds distance between two phylogenetic networks. While the fastest known algorithm for computing the Robinson-Foulds distance between two phylogenetic networks with n leaves, m nodes, and e edges runs in $O(m(m+e))$ time, the new algorithm takes advantage of bit-level parallelism and runs in $O(n(m+e)/\log n)$ time on general networks, assuming a word size of $\omega = \lceil \log n \rceil$ bits. In the case of level-k phylogenetic networks, we take advantage of the succinct representation of clusters as intervals of consecutive integers, and the new algorithm runs in $O((k+1)(m+e))$ time.

We have also introduced a new parameter, the *minimum spread* of a phylogenetic network, and proved that every level-k network has minimum spread at most $k+1$. For the particular case of bounded-level phylogenetic networks such as planar phylogenetic networks, which include outer-labeled planar split networks and galled-trees, we have shown that the minimum spread is 1, meaning that the new algorithm can be implemented to run in optimal $O(m+e)$ time.

Acknowledgment

JJ was supported by the Special Coordination Funds for Promoting Science and Technology. TA, RU, GV were supported by the Spanish government and the EU FEDER program under project PCI2006-A7-0603.

References

1. Baroni, M., Semple, C., Steel, M.: Hybrids in real time. Syst. Biol. 55(1), 46–56 (2006)
2. Bryant, D., Moulton, V.: Neighbor-Net: An agglomerative method for the construction of phylogenetic networks. Mol. Biol. Evol. 21(2), 255–265 (2004)
3. Cardona, G., Llabrés, M., Rosselló, F., Valiente, G.: Metrics for phylogenetic networks I: Generalizations of the Robinson-Foulds metric. IEEE ACM T. Comput. Biol. 6(1), 1–16 (2009)
4. Cardona, G., Rosselló, F., Valiente, G.: Comparison of tree-child phylogenetic networks. IEEE ACM T. Comput. Biol. (2009)
5. Choy, C., Jansson, J., Sadakane, K., Sung, W.K.: Computing the maximum agreement of phylogenetic networks. Theor. Comput. Sci. 335(1), 93–107 (2005)
6. Day, W.H.E.: Optimal algorithms for comparing trees with labeled leaves. J. Classif. 2(1), 7–28 (1985)
7. Doolittle, W.F.: Phylogenetic classification and the universal tree. Science 284(5423), 2124–2128 (1999)
8. Grünewald, S., Forslund, K., Dress, A., Moulton, V.: QNet: An agglomerative method for the construction of phylogenetic networks from weighted quartets. Mol. Biol. Evol. 24(2), 532–538 (2007)
9. Grünewald, S., Moulton, V., Spillner, A.: Consistency of the QNet algorithm for generating planar split networks from weighted quartets. Discr. Appl. Math. 157(10), 2325–2334 (2009)

10. Gusfield, D., Eddhu, S., Langley, C.: Efficient reconstruction of phylogenetic networks with constrained recombination. In: Proc. 2nd IEEE Computer Society Bioinformatics Conf., pp. 363–374 (2003)
11. Gusfield, D., Eddhu, S., Langley, C.H.: The fine structure of galls in phylogenetic networks. INFORMS J. Comput. 16(4), 459–469 (2004)
12. Hagerup, T.: Sorting and searching on the word RAM. In: Meinel, C., Morvan, M. (eds.) STACS 1998. LNCS, vol. 1373, pp. 366–398. Springer, Heidelberg (1998)
13. Jin, G., Nakhleh, L., Snir, S., Tuller, T.: Maximum likelihood of phylogenetic networks. Bioinformatics 22(21), 2604–2611 (2006)
14. Jin, G., Nakhleh, L., Snir, S., Tuller, T.: Efficient parsimony-based methods for phylogenetic network reconstruction. Bioinformatics 23(2), 123–128 (2007)
15. Pattengale, N.D., Gottlieb, E.J., Moret, B.M.: Efficiently computing the Robinson-Foulds metric. J. Comput. Biol. 14(6), 724–735 (2007)
16. Posada, D., Crandall, K.A.: Intraspecific gene genealogies: Trees grafting into networks. Trends Ecol. Evol. 16(1), 37–45 (2001)
17. Robinson, D.F., Foulds, L.R.: Comparison of phylogenetic trees. Math. Biosci. 53(1/2), 131–147 (1981)
18. Rosselló, F., Valiente, G.: All that glisters is not galled. Math. Biosci. 221(1), 54–59 (2009)
19. Strimmer, K., Moulton, V.: Likelihood analysis of phylogenetic networks using directed graphical models. Mol. Biol. Evol. 17(6), 875–881 (2000)
20. Strimmer, K., Wiuf, C., Moulton, V.: Recombination analysis using directed graphical models. Mol. Biol. Evol. 18(1), 97–99 (2001)
21. Sul, S.-J., Brammer, G., Williams, T.L.: Efficiently computing arbitrarily-sized Robinson-Foulds distance matrices. In: Crandall, K.A., Lagergren, J. (eds.) WABI 2008. LNCS (LNBI), vol. 5251, pp. 123–134. Springer, Heidelberg (2008)
22. Sul, S.-J., Williams, T.L.: An experimental analysis of Robinson-Foulds distance matrix algorithms. In: Halperin, D., Mehlhorn, K. (eds.) Esa 2008. LNCS, vol. 5193, pp. 793–804. Springer, Heidelberg (2008)
23. van Iersel, L., Keijsper, J., Kelk, S., Stougie, L., Hagen, F., Boekhout, T.: Constructing level-2 phylogenetic networks from triplets. IEEE ACM T. Comput. Biol. 6(4), 667–681 (2009)

Mod/Resc Parsimony Inference

Igor Nor[1,2,⋆], Danny Hermelin[3], Sylvain Charlat[1], Jan Engelstadter[4], Max Reuter[5], Olivier Duron[6], and Marie-France Sagot[1,2,⋆]

[1] Université de Lyon, F-69000, Lyon, Université Lyon 1, CNRS, UMR5558
[2] Bamboo Team, INRIA Grenoble Rhône-Alpes, France
[3] Max Planck Institute for Informatics, Saarbrücken - Germany
[4] Institute of Integrative Biology, ETH Zurich, Switzerland
[5] University College London, UK
[6] Institute of Evolutionary Sciences, CNRS - University of Montpellier II, France
norigor@gmail.com, Marie-France.Sagot@inria.fr

Abstract. We address in this paper a new computational biology problem that aims at understanding a mechanism that could potentially be used to genetically manipulate natural insect populations infected by inherited, intra-cellular parasitic bacteria. In this problem, that we denote by Mod/Resc Parsimony Inference, we are given a boolean matrix and the goal is to find two other boolean matrices with a minimum number of columns such that an appropriately defined operation on these matrices gives back the input. We show that this is formally equivalent to the Bipartite Biclique Edge Cover problem and derive some complexity results for our problem using this equivalence. We provide a new, fixed-parameter tractability approach for solving both that slightly improves upon a previously published algorithm for the Bipartite Biclique Edge Cover. Finally, we present experimental results where we applied some of our techniques to a real-life data set.

Keywords: Computational biology, biclique edge covering, bipartite graph, boolean matrix, NP-completeness, graph theory, fixed-parameter tractability, kernelization.

1 Introduction

Wolbachia is a genus of inherited, intra-cellular bacteria that infect many arthropod species, including a significant proportion of insects. The bacterium was first identified in 1924 by M. Hertig and S. B. Wolbach in Culex pipiens, a species of mosquito. *Wolbachia* spreads by altering the reproductive capabilities of its hosts [6]. One of these alterations consists in inducing so-called *cytoplasmic incompatibility* [7]. This phenomenon, in its simplest expression, results in the death of embryos produced in crosses between males carrying the infection and uninfected females. A more complex pattern is the death of embryos seen in crosses between males and females carrying different *Wolbachia* strains. The study of *Wolbachia*

⋆ Corresponding authors.

A. Amir and L. Parida (Eds.): CPM 2010, LNCS 6129, pp. 202–213, 2010.

and cytoplasmic incompatibility is of interest due to the high incidence of such infections, amongst others in human disease vectors such as mosquitoes, where cytoplasmic incompatibility could potentially be used as a driver mechanism for the genetic manipulation of natural populations.

The molecular mechanisms underlying cytoplasmic incompatibility are currently unknown, but the observations are consistent with a "toxin / antitoxin" model [16]. According to this model, the bacteria present in males modify the sperm (the so-called modification, or mod factor) by depositing a "toxin" during its maturation. Bacteria present in females, on the other hand, deposit an antitoxin (rescue, or resc factor) in the eggs, so that offsprings of infected females can develop normally. The simple compatibility patterns seen in several insect hosts species [1,2,3] has led to the general view that cytoplasmic incompatibility relies on a single pair of mod / resc genes. However, more complex patterns, such as those seen in Figure 1 of the mosquito *Culex pipiens* [5], suggest that this conclusion cannot be generalized. The aim of this paper is to provide a first model and algorithm to determine the minimum number of mod and resc genes required to explain a compatibility dataset for a given insect host. Such an algorithm will have an important impact on the understanding of the genetic architecture of cytoplasmic incompatibility. Beyond *Wolbachia*, the method proposed here can be applied to any parasitic bacteria inducing cytoplasmic incompatibility.

C	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
1	0	0	1	0	0	0	1	1	0	0	0	0	0	1	1	0	1	1	1
2	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
3	0	1	0	0	0	1	0	1	0	0	0	0	0	1	0	0	0	0	0
4	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0
7	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
8	1	1	1	1	1	0	1	0	0	1	0	1	1	1	1	1	1	1	0
9	0	0	1	0	0	0	1	1	0	0	0	0	0	1	1	1	1	1	0
10	1	0	1	0	0	0	1	0	1	0	1	0	0	1	0	0	1	1	0
11	0	1	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0
12	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0
13	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
15	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
16	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
17	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
18	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
19	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fig. 1. The Culex pipiens dataset. Rows represent females and columns males.

Let us now propose a formal description of this problem. Let the *compatibility matrix* C be an n-by-n matrix describing the observed cytoplasmic compatibility relationships among n strains, with females in rows and males in columns. For the *Culex pipiens* dataset, the content of the C matrix is directly given by Figure 1. For each entry $C_{i,j}$ of this matrix, a value of 1 indicates that the cross between the i'th female and j'th male is incompatible, while a value of 0 indicates it

is compatible. No intermediate levels of incompatibility are observed in *Culex pipiens*, so that such a discrete code (0 or 1) is sufficient to describe the data. Let the *mod matrix* M be an n-by-k matrix, with n strains and k mod genes. For each $M_{i,j}$ entry, a 0 indicates that strain i does not carry gene j, and a 1 indicates that it does carry this gene. Similarly, the *rescue matrix* R is an n-by-k matrix, with n strains and k resc genes, where each $R_{i,j}$ entry indicates whether strain i carries gene j. A cross between female i and male j is compatible only if strain i carries at least all the rescue genes matching the mod genes present in strain j. Using this rule, one can assess whether an (M, R) pair is a solution to the C matrix, that is, to the observed data.

We can easily find non-parsimonious solutions to this problem, that is, large M and R matrices that are solutions to C, as will be proven in the next section. However, solutions may also exist with fewer mod and resc genes. We are interested in the minimum number of genes for which solutions to C exist, and the set of solutions for this minimum number. This problem can be summarized as follows: Let C (compatibility) be a boolean n-by-n matrix. A pair of n-by-k boolean matrices M (mod) and R (resc) is called a solution to C if, for any row j in R and row i in M, $C_{i,j} = 0$ if and only if $R_{j,\ell} \geq M_{i,\ell}$ holds for all ℓ, $1 \leq \ell \leq k$. This appropriately models the fact stated above that, for any cross to be compatible, the female must carry at least all the rescue genes matching the mod genes present in the male. For a given matrix C, we are interested in the minimum value of k for which solutions to C exist, and the set of solutions for this minimum k. We refer to this problem as the Mod/Resc Parsimony Inference problem (see also Section 2). Since in some cases, data (on females or males) may be missing, the compatibility matrix C has dimension n-by-m for n not necessarily equal to m. We will consider this more general situation in what follows.

In this paper, we present the Mod/Resc Parsimony Inference problem and prove it is equivalent to a well-studied graph-theoretical problem known in the literature by the name of Bipartite Biclique Edge Cover. In this problem, we are given a bipartite graph, and we want to cover its edges with a minimum number of complete bipartite subgraphs (bicliques). This problem is known to be NP-complete, and thus Mod/Resc Parsimony Inference turns out to be NP-complete as well. In Section 4, we investigate a previous fixed-parameter tractability approach [8] for solving the Bipartite Biclique Edge Cover problem and improve its algorithm. In addition, we show a reduction between this problem and the Clique Edge Cover problem. Finally, in Section 5, we present experimental results where we applied some of these techniques to the *Culex pipiens* data set presented in Figure 1. This provided a surprising finding from a biological point of view.

2 Problem Definition and Notation

In this section, we briefly review some notation and terminology that will be used throughout the paper. We also give a precise mathematical definition of

the Mod/Resc Parsimony Inference problem we study. For this, we first need to define a basic operation between two boolean vectors:

Definition 1. *The $\otimes$ vectors multiplication is an operation between two boolean vectors $U, V \in \{0,1\}^k$ such that :*

$$U \otimes V := \begin{cases} 1 & : \quad U[i] > V[i] \text{ for some } i \in \{1, \ldots, k\} \\ 0 & : \quad \text{otherwise} \end{cases}$$

In other words, the result of the $\otimes$ multiplication is 0 if, for all corresponding locations, the value in the second vector is not less than in the first.

The reader should note that this operation is not symmetric. For example, if $U := (0,1,1,0)$ and $V := (1,1,1,0)$, then $U \otimes V = 0$, while $V \otimes U = 1$. We next generalize the $\otimes$ multiplication to boolean matrices. This follows easily from the observation that the boolean vectors $U, V \in \{0,1\}^k$ may be seen as matrices of dimension 1-by-k. We thus use the same symbol $\otimes$ to denote the operation applied to matrices.

Definition 2. *The $\otimes$ row-by-row matrix multiplication is a function $\{0,1\}^{n \times k} \times \{0,1\}^{m \times k} \rightarrow \{0,1\}^{n \times m}$ such that $C = M \otimes R$ iff $C_{i,j} = M_i \otimes R_j$ for all $i \in \{1, \ldots, n\}$ and $j \in \{1, \ldots m\}$. (Here M_i and R_j respectively denote the i'th and j'th row of M and R.)*

Definition 3. *In the Mod/Resc Parsimony Inference problem, the input is a boolean matrix $C \in \{0,1\}^{n \times m}$, and the goal is to find two boolean matrices $M \in \{0,1\}^{n \times k}$ and $R \in \{0,1\}^{m \times k}$ such that $C = M \otimes R$ and with k minimal.*

We first need to prove there is always a correct solution to the Mod/Resc Inference Problem. Here we show that there is always a solution for as many mod and resc genes as the minimum between the number of male and female strains in the dataset.

Lemma 1. *The Mod/Resc Parsimony Inference problem always has a solution.*

Proof. A satisfying output for the Mod/Resc Parsimony Inference problem always exists for any possible C of size n-by-m. For instance, let M be of size n-by-n and equal to the identity matrix, and let R be of size m-by-n and such that $R = \overline{C}^T$. This solution is correct since the only 1-value in an arbitrary row r_i of the matrix M is at location M_{ii}. Thus, the only situation where $C_{ij} = 1$ is when $R_{ji} = 0$, which is the case by construction. □

We will be using some standard graph-theoretic terminology and notation. We use G, G', and so forth to denote graphs in general, where $V(G)$ denotes the vertex set of a graph G, and $E(G)$ its edge-set. By a *subgraph* of G, we mean a graph G' with $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$. For a bipartite graph G, *i.e.* a graph whose vertex-set can be partitioned into two classes with no edges occurring between vertices of the same class, we use $V_1(G)$ and $V_2(G)$ to denote the two vertex classes of G. A *complete bipartite graph* (*biclique*) is a bipartite graph G with $E(G) := \{\{u, v\} : u \in V_1(G), v \in V_2(G)\}$. We will sometimes use B, B_1, and so forth to denote bicliques.

3 Equivalence to Bipartite Biclique Edge Cover

In this section, we show that the Mod/Resc Parsimony Inference problem is equivalent to a classical and well-studied graph theoretical problem known in the literature as the Bipartite Graph Biclique Edge Cover problem. Using this equivalence, we first derive the complexity status of Mod/Resc Parsimony Inference, and later devise FPT algorithms for this problem. We begin with a formal definition of the Bipartite Graph Biclique Edge Cover problem.

Definition 4. *In the* Bipartite Biclique Edge Cover Problem *problem, the input is a bipartite graph G, and the goal is to find the minimum number of biclique subgraphs $B_1, \ldots, B_k$ of G such that $E(G) := \bigcup_\ell E(B_\ell)$.*

Given a bipartite graph G with $V_1(G) := \{u_1, \ldots, u_n\}$ and $V_2(G) := \{u_1, \ldots, u_m\}$, the *bi-adjacency* matrix of G is a boolean matrix $A(G) \in \{0,1\}^{n \times m}$ defined by $A(G)_{i,j} := 1 \iff \{u_i, v_j\} \in E(G)$. In this way, every boolean matrix C corresponds to a bipartite graph, and vice versa.

Theorem 1. *Let C be a boolean matrix of size $n \times m$. Then there are two matrices $M \in \{0,1\}^{n \times k}$ and $R \in \{0,1\}^{m \times k}$ with $C = M \otimes R$ iff the bipartite graph G with $A(G) := C$ has a biclique edge cover with k bicliques.*

Proof. ($\Longleftarrow$) Let G be the bipartite graph with the bi-adjacency matrix C, and suppose G has biclique edge cover $B_1, B_2, \ldots, B_k$. We construct two boolean matrices M and R as follows. Let $V_1(G) := \{u_1, \ldots, u_n\}$ and $V_2(G) := \{v_1, \ldots, v_m\}$. We define:

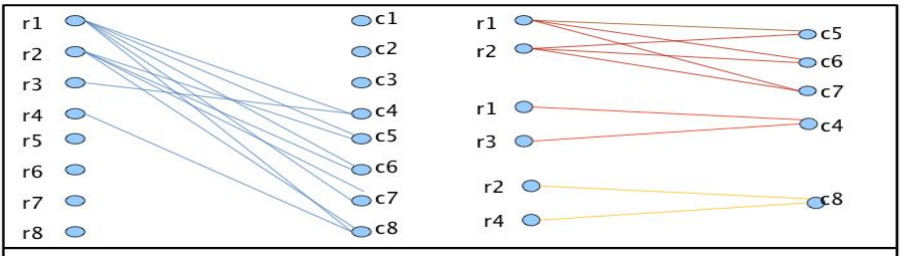
1. $M_{i,\ell} = 1 \iff u_i \in V_1(B_\ell)$.
2. $R_{j,\ell} = 0 \iff v_j \in V_2(B_\ell)$.

An illustration of this construction is given in Figure 2.

We argue that $C = M \otimes R$. Consider an arbitrary location $C_{i,j} = 1$. By definition we have $\{u_i, v_j\} \in E(G)$. Since the bicliques $B_1, \ldots, B_k$ cover all edges of G, we know that there is some ℓ, $\ell \in \{1, \ldots, k\}$, with $u_i \in V_1(B_\ell)$ and $v_j \in V_2(B_\ell)$. By construction we know that $M_{i,\ell} = 1$ and $R_{j,\ell} = 0$, and so $M_i \otimes R_j = 1$, which means that the entry at row i and column j in $M \otimes C$ is equal to 1. On the other hand, if $C_{ij} = 0$, then $\{u_i, v_j\} \notin E(G)$, and thus there is no biclique B_ℓ with $u_i \in V_1(B_\ell)$ and $v_j \in V_2(B_\ell)$. As a result, for all $\ell \in \{1, \ldots, k\}$, if $M_{i,\ell} = 1$ then $R_{i,\ell} = 1$ as well, which means that the result of the $\otimes$ multiplication between the i'th row in M and the j'th row in R will be equal to 0.

($\Longrightarrow$) Assume there are two matrices $M \in \{0,1\}^{n \times k}$ and $R \in \{0,1\}^{m \times k}$ with $C = M \otimes R$. Construct k subgraphs $B_1, \ldots, B_k$ of G, where the ℓ'th subgraph is defined as follows:

1. $u_i \in V_1(B_\ell) \iff M_{i,\ell} = 1$.
2. $v_j \in V_2(B_\ell) \iff R_{j,\ell} = 0$.
3. $\{u_i, v_j\} \in E(B_\ell) \iff \{u_i, v_j\} \in E(G)$.



G	1	2	3	4	5	6	7	8
1	0	0	0	1	1	1	1	0
2	0	0	0	0	1	1	1	1
3	0	0	0	1	0	0	0	0
4	0	0	0	0	0	0	0	1
5	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0

M	1	2	3
1	1	1	0
2	1	0	1
3	0	1	0
4	0	0	1
5	0	0	0
6	0	0	0
7	0	0	0
8	0	0	0

R	1	2	3
1	1	1	1
2	1	1	1
3	1	1	1
4	1	0	1
5	0	1	1
6	0	1	1
7	0	1	1
8	1	1	0

Fig. 2. Reduction illustrated

We first argue that each of the subgraphs $B_1, \ldots, B_k$ is a biclique. Consider an arbitrary subgraph B_ℓ, and an arbitrary pair of vertices $u_i \in V_1(B_\ell)$ and $v_j \in V_2(B_\ell)$. By construction, it follows that $M_{i,\ell} = 1$ and $R_{i,\ell} = 0$. As a result, it must be that $C_{i,j} = 1$, which means that $\{u_i, v_j\} \in E(G)$. Next, we argue that $\bigcup_\ell E(B_\ell) = E(G)$. Consider an arbitrary edge $\{u_i, v_j\} \in E(G)$. Since $C = A(G)$, we have $C_{i,j} = 1$. Furthermore, since $M \otimes R = C$, there must be some $\ell \in \{1, \ldots, k\}$ with $M_{i,\ell} > R_{j,\ell}$. However, this is exactly the condition for having u_i and v_j in the biclique subgraph B_ℓ. It follows that indeed $\bigcup_\ell E(B_\ell) = E(G)$, and thus the theorem is proved. □

Due to the equivalence between Mod/Resc Parsimony Inference and Bipartite Biclique Edge Cover, we can infer from known complexity results regarding Bipartite Biclique Edge Cover the complexity of our problem. First, since Bipartite Biclique Edge Cover is well-known to be NP-complete [15], it follows that Mod/Resc Parsimony Inference is NP-complete as well. Furthermore, Gruber and Holzer [11] recently showed that the Bipartite Biclique Edge Cover problem cannot be approximated within a factor of $n^{1/3-\varepsilon}$ unless P = NP where n is the total number of vertices. Since the reduction given in Theorem 1 is clearly an approximate preserving reduction, we can deduce the following:

Theorem 2. Mod/Resc Parsimony Inference *is* NP-complete*, and furthermore, for all* $\varepsilon > 0$*, the problem cannot be approximated within a factor of* $(n+m)^{1/3-\varepsilon}$ *unless* P = NP.

4 Fixed-Parameter Tractability

In this section, we explore a parameterized complexity approach [4,9,14] for the Mod/Resc Parsimony Inference problem. Due to the equivalence shown in the previous section, we focus for convenience reasons on Bipartite Biclique Edge Cover. In parameterized complexity, problem instances are appended with an additional parameter, usually denoted by k, and the goal is to find an algorithm for the given problem which runs in time $f(k) \cdot n^{O(1)}$, where f is an arbitrary computable function. In our context, our goal is to determine whether a given input bipartite graph G with n vertices has a biclique edge cover of size k in time $f(k) \cdot n^{O(1)}$.

4.1 The Kernelization

Fleischner *et al.* [8] studied the Bipartite Biclique Edge Cover problem in the context of parameterized complexity. The main result in their paper is to provide a kernel for the problem based on the techniques given by Gramm *et al.* [10] for the similar Clique Edge Cover problem. Kernelization is a central technique in parameterized complexity which is best described as a polynomial-time transformation that converts instances of arbitrary size to instances of a size bounded by the problem parameter (usually of the same problem), while mapping "yes"-instances to "yes"-instances, and "no"-instances to "no"-instances. More precisely, a *kernelization algorithm* $\mathcal{A}$ for a parameterized problem (language) Π is a polynomial-time algorithm such that there exists some computable function f that, given an instance (I, k) of Π, $\mathcal{A}$ produces an instance (I', k') of Π with:

- $|I'| + k' \leq f(k)$, and
- $(I, k) \in \Pi \iff (I', k') \in \Pi$.

We refer the reader to *e.g.* [12,14] for more information on kernelization.

A typical kernelization algorithm works with reduction rules, which transform a given instance to a slightly smaller equivalent instance in polynomial time. The typical argument used when working with reduction rules is that once none of these can be applied, the resultant instance has size bounded by a function of the parameter. For the Bipartite Biclique Edge Cover, two kernelization rules have been applied by Fleischner *et al.* [8]:

RULE 1: If G has a vertex with no neighbors, remove this vertex without changing the parameter.

RULE 2: If G has two vertices with identical neighbors, remove one of these vertices without changing the parameter.

Lemma 2 ([8]). *Applying rules 1 and 2 above exhaustively gives a kernelization algorithm for* Bipartite Biclique Edge Cover *that runs in* $O(max(n, m)^3)$ *time, and transforms an instance* (G, k) *to an equivalent instance* (G', k) *with* $|V(G')| \leq 2^k$ *and* $|E(G')| \leq 2^{2k}$.

We add two additional rules, which will be necessary for further interesting properties.

RULE 3: If there is a vertex v with exactly one neighbor u in G, then remove both v and u, and decrease the parameter by one.

Lemma 3. *Rule 3 is correct.*

Proof. Assume a biclique cover of size k of the graph, and assume that vertex v is a member of some of the bicliques in this cover. By definition, at least one of the bicliques covers the edge $\{u, v\}$. Since this is the only edge adjacent to v, the bicliques that cover $\{u, v\}$ include only vertex u among the vertices in its bipartite vertex class. Thus, a biclique that covers $\{u, v\}$ can be extended to cover all other edges of u while keeping the property of being a biclique. □

RULE 4: Assume Rule 3 does not apply. If there is a vertex v in G which is adjacent to all vertices in the opposite bipartition class of G, then remove v without decreasing the parameter.

Lemma 4. *Rule 4 is correct.*

Proof. After applying Rule 3 above, each remaining vertex in the graph has at least two neighbors. Assume a biclique cover of size k of all the edges except those adjacent to vertex v. Assume w.l.o.g. that $v \in V_1(G)$. Since each vertex $u \in V_2(G)$ has degree at least 2, it is adjacent to an edge which is covered by the biclique cover. It therefore belongs to some biclique in this cover. For each biclique in the cover, add now vertex v to its set of vertices. Since v is adjacent to all the vertices of $V_2(G)$, each changed component is a correct biclique and the new solution covers all the edges, including those of vertex v, and is of same size. □

Let us now consider the time complexity for checking the new rules introduced. Let us assume we have a counter for each vertex, which has the size of its set of neighbors. Once a vertex has been found to which the rule should be applied, applying each rule takes $O(max(n, m))$ time, including updating the counters of the neighbors of the deleted vertex. Linearly running through the vertices and checking each rule condition also requires $O(max(n, m))$ time using the counters. Since one can apply the reduction rules at most $O(max(n, m))$ times, the total time required for the extended kernelization remains $O(max(n, m)^3)$. We observe that although the new rules do not change the kernelization size, which remains 2^k vertices in a solution of size k, they can be useful in the following section.

4.2 Bipartite Biclique Edge Cover and Clique Edge Cover

In this section, we show the connection between the Bipartite Biclique Edge Cover and the Clique Edge Cover problems. We show that in the context of fixed-parameter tractability, we can easily translate our problem to the classical

clique covering problem and then use it for a solution to our problem. For instance, it gives another way for the kernelization of the problem and can provide interesting heuristics, mentioned in [10].

Given a kernelized bipartite graph G' as an instance to the Bipartite Biclique Edge Cover problem, we transform G' into a (non-bipartite) graph G'' defined by $V(G'') := V(G') \cup \{v'\} \cup \{u'\}$ and $E(G'') := E(G') \cup \{\{u, v\} : u, v \in V_1(G') \cup \{v'\}$ and $u, v \in V_2(G') \cup \{u'\}\}$ where v' and u' are two new nodes not in $V(G')$.

Theorem 3. *The edges of G' can be covered with k cliques iff the edges of G'' can be covered with $k + 2$ cliques.*

Proof. Suppose $B_1, \ldots, B_k$ is a biclique edge cover of G'. Then each $V(B_i)$, $i \in \{1, \ldots, k\}$, induces a clique in G''. Furthermore, the only remaining edges which are not covered in G'' are the ones between vertices in $V_1(G') \cup \{v'\}$ and vertices in $V_2(G') \cup \{u'\}$, which can be covered by the two cliques induced by these vertex sets in G''. Altogether this gives us $k+2$ cliques that cover all edges in G''. Conversely, take a clique edge cover $K_1, \ldots, K_c$ of G''. By construction, v' cannot share a same clique with any node in $V_2(G') \cup \{u'\}$ and likewise u' cannot share a same clique with any node in $V_1(G') \cup \{v'\}$. It follows that there must be at least two cliques in $\{K_1, \ldots, K_c\}$, say K_1 and K_2, with $V(K_1) \subseteq V_1(G') \cup \{v'\}$ and $V(K_2) \subseteq V_2(G') \cup \{u'\}$. Thus, there is a subset of the cliques in $\{K_3, \ldots, K_c\}$ which have vertices in both partition classes of G', and which cover all the edges in G'. Taking the corresponding bicliques in G', and adding duplicated bicliques if necessary, gives us k bicliques that cover all edges in G'. □

4.3 Algorithms

After the kernelization algorithm is applied, the next step is usually to solve the problem using brute-force. This is what is done in [8]. However, the time complexity given there is inaccurate, and the parametric-dependent time bound of their algorithm is $O(k^{4^k} 2^{3k}) = O(2^{2^{2k} \lg k + 3k})$ instead of the $O(2^{2k^2+3k})$ bound stated in their paper. Furthermore, the algorithm they describe is initially given for the related Bipartite Biclique Edge Partition problem (where each edge is allowed to appear exactly once in a biclique), and the adaptation of such algorithm to the Bipartite Biclique Edge Cover problem is left vague and imprecise. Here, we suggest two possible brute-force procedures for the Bipartite Biclique Edge Cover problem, each of which outperforms the algorithm of [8] in the worst-case. We assume throughout that we are working with a kernelized instance obtained by applying the algorithm described in Section 4.1, *i.e.* a pair (G', k) where G' is a bipartite graph with at most 2^k vertices (and consequently at most 4^k edges).

The first brute-force algorithm: For each $k' \leq k$, try all possible partitions of the edge-set $E(G')$ of G' into k' subsets. For each such partition $\Pi = \{E_1, \ldots, E_{k'}\}$, check whether each of the subgraphs $G'[E_1], \ldots, G'[E_{k'}]$ is a biclique, where

$G'[E_i]$ is the subgraph of G induced by E_i. If yes, report $G'[E_1], \ldots, G'[E_{k'}]$ as a solution. If some $G'[E_i]$ is not a biclique, check whether edges in $E(G') \setminus E(G'_i)$ can be added to $E[G'_i]$ in order to make the graph a biclique. Continue with the next partition if some graph in $G'[E_1], \ldots, G'[E_{k'}]$ cannot be appended in this way in order to get a biclique, and otherwise report the solution found. Finally, if the above procedure fails for all partitions of $E(G')$ into $k' \leq k$ subsets, report that G' does not have a biclique edge cover of size k.

Lemma 5. *The above algorithm correctly determines whether G' has a bipartite biclique edge cover of size k in time $\frac{2^{2^{2k} \lg k + 2k + \lg k}}{k!}$.*

Proof. Correctness of the above algorithm is immediate in case a solution is found. To see that the algorithm is also correct when it reports that no solution can be found, observe that for any biclique edge cover $B_1, \ldots, B_k$ of G, the set $\{E_1, \ldots, E_k\}$ with $E_i := E(G'_i) \setminus \bigcup_{j<i} E(G'_j)$ defines a partition of $E(G')$ (with some of the E_i's possibly empty), and given this partition, the algorithm above would find the biclique edge cover of G'. Correctness of the algorithm thus follows.

Regarding the time complexity, the time needed for appending edges to each subgraph is at most $O(|(V(G'))^2|) = O(2^{2k})$, and thus a total of $O(2^{2k}k) = O(2^{2k+\lg k})$ time is required for the entire partition. The number of possible partitions of $E(G')$ into k disjoint set is the *Stirling number of the second kind* $S(2^{2k}, k)$, which has been shown in [13] to be asymptotically equal to $O(\frac{k^{4^k}}{k!} = \frac{2^{2^{2k} \lg k}}{k!})$. Thus, the total complexity of the algorithm is $O(\frac{2^{2^{2k} \lg k + 2k + \lg k}}{k!})$. □

The second brute-force algorithm: We generate the set $\mathcal{K}(G')$ of all possible inclusion-wise maximal bicliques in G', and try all possible k-subsets of $\mathcal{K}(G')$ to see whether one covers all edges in G'. Correctness of the algorithm is immediate since one can always restrict oneself to using only inclusion-wise maximal bicliques in a biclique edge cover. To generate all maximal bicliques, we first transform G' into the graph G'' given in Theorem 3. Thus, every inclusion-wise maximal biclique in G' is an inclusion-wise maximal clique in G''. We then use the algorithm of [18] on the complement graph $\overline{G''}$ of G'', *i.e.* the graph defined by $V(\overline{G''}) := V(G'')$ and $E(\overline{G''}) := \{\{u, v\} : u, v \in V(\overline{G''}), u \neq v, \text{ and } \{u, v\} \notin E(G'')\}$.

Theorem 4. *The* Bipartite Biclique Edge Cover *problem can be solved in $O(f(k) + max(n, m)^3)$ time, where $f(k) := 2^{k2^{k-1}+3k}$.*

Proof. Given a bipartite graph G as an instance to Bipartite Biclique Edge Cover, we first apply the kernelization algorithm to obtain an equivalent graph G' with 2^k vertices, and then apply the brute-force algorithm described above to determine whether G' has a biclique edge cover of size k. Correctness of this algorithm follows directly from Section 4.1 and the correctness of the brute-force procedure. To analyze the time complexity of this algorithm, we first note that Prisner showed that any bipartite graph on n vertices has at most $2^{n/2}$ inclusion-wise maximal bicliques [18]. This implies that

$|\mathcal{K}(G')| \leq 2^{2^{k-1}}$. The algorithm of [17] runs in $O(|V(G')||E(G')||\mathcal{K}(G')|)$ time, which is $O(2^k 2^{2k} 2^{2^{k-1}}) = O(2^{2^{k-1}+3k})$. Finally, the total number of k-subsets of $\mathcal{K}(G')$ is $O(2^{k2^{k-1}})$, and checking whether each of these subsets covers the edges of G' requires $O(|V(G')||E(G')|) = O(2^{3k})$ time. Thus, the total time complexity of the entire algorithm is $O(2^{2^{k-1}+3k} + 2^{k2^{k-1}+3k} + max(n,m)^3) = O(2^{k2^{k-1}+3k} + max(n,m)^3)$. □

It is worthwhile mentioning that some particular bipartite graphs have a number of inclusion-wise maximal bicliques which is polynomial in the number of their vertices. For these types of bipartite graphs, we could improve on the worst-case analysis given in the theorem above. For instance, a bipartite chordal graph G has at most $|E(G)|$ inclusion-wise maximal bicliques [18]. A bipartite graph with $n+m$ vertices and no induced cocktail-party graph of order ℓ has at most $max(n,m)^{2(\ell-1)}$ inclusion-wise maximal bicliques [17]. The cocktail party graph of order ℓ is the graph with nodes consisting of two rows of paired nodes in which all nodes but the paired ones are connected with a graph edge (for a full definition, see [17]). Observing that the algorithm in Section 4.1 preserves chordality and does not introduce any new cocktail-party induced subgraphs, we obtain the following corollary:

Corollary 1. *The* Bipartite Biclique Edge Cover *problem can be solved in* $O(2^{2k^2+3k} + max(n,m)^3)$ *time when restricted to chordal bipartite graphs, and in* $O(2^{2k^2(\ell-1)+3k} + max(n,m)^3)$ *time when restricted to bipartite graphs with no induced cocktail-party graphs of order* ℓ.

5 Experimental Results

We performed experiments of the parameterized algorithms on the *Culex pipiens* dataset, given in Figure 1. We implemented the algorithms in the C++ programming language, with source code of approximately 2500 lines.

The main difficulty in practice is to find the minimal size k. Different approaches could be used. One would proceed by first checking if there is no solution of small sizes since this is easy to check using the FPT approach, and then increasing the size until reaching a smallest size k for which one solution exists. Another would proceed by using different fast and efficient heuristics to discover a solution of a given size k' that in general will be greater than the optimal size k sought. Then applying dichotomy (the optimal solution is between 1 and $k'-1$), the minimal size could be found using the FPT approach for the middle value between 1 and $k'-1$, and so on. The source code and the results can be viewed on the webpage *http://lbbe.univ-lyon1.fr/-Nor-Igor-.html*.

The result obtained on the *Culex pipiens* dataset indicates that 8 pairs of mod/resc genes are required to explain the dataset. This appears to be in sharp contrast to simpler patterns seen in other host species [2,3,1] that had led to the general belief that cytoplasmic incompatibility can be explained with a single pair of mod / resc genes. In biological terms, this result means that contrary to

earlier beliefs, the number of genetic determinants of cytoplasmic incompatibility present in a single *Wolbachia* strain can be large, consistent with the view that it might involve repeated genetic elements such as transposable elements or phages.

References

1. Bordenstein, S.R., Werren, J.H.: Bidirectional incompatibility among divergent *wolbachia* and incompatibility level differences among closely related *wolbachia* in *nasonia*. Heredity 99(3), 278–287 (2007)
2. Merçot, H., Charlat, S.: *Wolbachia* infections in *drosophila melanogaster* and *d. simulans*: polymorphism and levels of cytoplasmic incompatibility. Genetica 120(1-3), 51–59 (2004)
3. Dobson, S.L., Marsland, E.J., Rattanadechakul, W.: *Wolbachia*-induced cytoplasmic incompatibility in single- and superinfected *aedes albopictus* (diptera: *Culicidae*). J Med Entomol. 38(3), 382–387 (2001)
4. Downey, R.G., Fellows, M.R.: Parameterized Complexity. Springer, Heidelberg (1999)
5. Duron, O., Bernard, C., Unal, S., Berthomieu, A., Berticat, C., Weill, M.: Tracking factors modulating cytoplasmic incompatibilities in the mosquito *culex pipiens*. Mol Ecol. 15(10), 3061–3071 (2006)
6. Engelstadter, J., Hurst, G.D.D.: The ecology and evolution of microbes that manipulate host reproduction. Annual Review of Ecology, Evolution and Systematics (40), 127–149 (2009)
7. Engelstadter, J., Telschow, A.: Cytoplasmic incompatibility and host population structure. Heredity (103), 196–207 (2009)
8. Fleischner, H., Mujuni, E., Paulusma, D., Szeider, S.: Covering graphs with few complete bipartite subgraphs. Theoretical Computer Science 410(21-23), 2045–2053 (2009)
9. Flum, J., Grohe, M.: Parameterized Complexity Theory. Springer, Heidelberg (2006)
10. Gramm, J., Guo, J., Huffner, F., Niedermeier, R.: Data reduction, exact, and heuristic algorithms for clique cover. In: Proceedings of the 8th ACM/SIAM workshop on ALgorithm ENgineering and EXperiments (ALENEX), pp. 86–94 (2006)
11. Gruber, H., Holzer, M.: Inapproximability of nondeterministic state and transition complexity assuming P≠NP. In: Harju, T., Karhumäki, J., Lepistö, A. (eds.) DLT 2007. LNCS, vol. 4588, pp. 205–216. Springer, Heidelberg (2007)
12. Guo, J., Niedermeier, R.: Invitation to data reduction and problem kernelization. SIGACT News 38(1), 31–45 (2007)
13. Korshunov, A.D.: Asymptotic behaviour of stirling numbers of the second kind. Diskret. Anal. 39(1), 24–41 (1983)
14. Niedermeier, R.: Invitation to Fixed-Parameter Algorithms. Oxford University Press, Oxford (2006)
15. Orlin, J.: Contentment in graph theory: covering graphs with cliques. Indagationes Mathematicae 80(5), 406–424 (1977)
16. Poinsot, D., Charlat, S., Merçot, H.: On the mechanism of *wolbachia*-induced cytoplasmic incompatibility: confronting the models with the facts. Bioessays 25(1), 259–265 (2003)
17. Prisner, E.: Bicliques in graphs I: Bounds on their number. Combinatorica 20(1), 109–117 (2000)
18. Tsukiyama, S., Ide, M., Ariyoshi, H., Shirakawa, I.: A new algorithm for generating all the maximal independent sets. SIAM Journal on Computing 6(3), 505–517 (1977)

Extended Islands of Tractability for Parsimony Haplotyping*

Rudolf Fleischer[1], Jiong Guo[2], Rolf Niedermeier[3], Johannes Uhlmann[3], Yihui Wang[1], Mathias Weller[3], and Xi Wu[1]

[1] School of Computer Science, IIPL, Fudan University, Shanghai, China
{rudolf,yihuiwang,wuxi}@fudan.edu.cn
[2] Universität des Saarlandes,
Campus E 1.4, D-66123 Saarbrücken, Germany
jguo@mmci.uni-saarland.de
[3] Institut für Informatik, Friedrich-Schiller-Universität Jena,
Ernst-Abbe-Platz 2, D-07743 Jena, Germany
{rolf.niedermeier,johannes.uhlmann,mathias.weller}@uni-jena.de

Abstract. Parsimony haplotyping is the problem of finding a smallest size set of haplotypes that can explain a given set of genotypes. The problem is NP-hard, and many heuristic and approximation algorithms as well as polynomial-time solvable special cases have been discovered. We propose improved fixed-parameter tractability results with respect to the parameter "size of the target haplotype set" k by presenting an $O^*(k^{4k})$-time algorithm. This also applies to the practically important constrained case, where we can only use haplotypes from a given set. Furthermore, we show that the problem becomes polynomial-time solvable if the given set of genotypes is complete, i.e., contains all possible genotypes that can be explained by the set of haplotypes.

1 Introduction

Over the last few years, haplotype inference has become one of the central problems in algorithmic bioinformatics [10,2]. Its applications include drug design, pharmacogenetics, mapping of disease genes, and inference of population histories. One of the major approaches to haplotype inference is *parsimony haplotyping*: Given a set of genotypes, the task is to find a minimum-cardinality set of haplotypes that explains the input set of genotypes. The task to select as few haplotypes as possible (parsimony criterion) is motivated by the observation that in natural populations the number of haplotypes is much smaller than the number of genotypes [2]. Referring for the background in molecular biology to the rich literature (see, e.g., the surveys by Catanzaro and Labbé [2] and Gusfield and Orzack [10]), we focus on the underlying combinatorial problem. In an abstract way, a genotype can be seen as a length-m string over the alphabet $\{0, 1, 2\}$,

* Supported by the DFG, research projects PABI, NI 369/7, and DARE, GU 1023/1, NI 369/11, NSF China (No. 60973026), Shanghai Leading Academic Discipline Project (project number B114), Shanghai Committee of Science and Technology of China (nos. 08DZ2271800 and 09DZ2272800), the Excellence Cluster on Multimodal Computing and Interaction (MMCI), and Robert Bosch Foundation (Science Bridge China 32.5.8003.0040.0).

A. Amir and L. Parida (Eds.): CPM 2010, LNCS 6129, pp. 214–226, 2010.

while a haplotype can be seen as a length-m string over the alphabet $\{0, 1\}$. A set H of haplotypes *explains*, or *resolves*, a set G of genotypes if for every $g \in G$ there is either an $h \in H$ with $g = h$ (trivial case), or there are two haplotypes h_1 and h_2 in H such that, for all $i \in \{1, \ldots, m\}$,

- if g has letter 0 or 1 at position i, then both h_1 and h_2 have this letter at position i, and
- if g has letter 2 at position i, then one of h_1 or h_2 has letter 0 at position i while the other one has letter 1.

For example, $H = \{00100, 01110, 10110\}$ resolves $G = \{02120, 20120, 22110\}$. Parsimony haplotyping is NP-hard, and numerous algorithmic approaches based on heuristics and integer linear programming methods are applied in practice [2]. There is also a growing list of combinatorial approaches (with provable performance guarantees) including the identification of polynomial-time solvable special cases, approximation algorithms, and fixed-parameter algorithms [5,13,14,16,11].

In this work, we contribute new combinatorial algorithms for parsimony haplotyping, based on new insights into the combinatorial structure of a haplotype solution. Lancia and Rizzi [14] showed that parsimony haplotyping can be solved in polynomial time if every genotype string contains at most two letters 2, while the problem becomes NP-hard if genotypes may contain three letters 2 [13]. Sharan et al. [16] proved that parsimony haplotyping is APX-hard in even very restricted cases and identified instances with a specific structure that allow for polynomial-time exact solutions or constant-factor approximations. Moreover, they showed that the problem is fixed-parameter tractable with respect to the parameter k ="number of haplotypes in the solution set". The corresponding exact algorithm has running time $O(k^{k^2+k}m)$. These results were further extended by van Iersel et al. [11] to cases where the *genotype matrix* (the rows are the genotypes and the columns are the m positions in the genotype strings) has restrictions on the number of 2's in the rows and/or columns. They identified various special cases of haplotyping with polynomial-time exact or approximation algorithms with approximation factors depending on the numbers of 2's per column and/or row, leaving open the complexity of the case with at most two 2's per column (and an unbounded number of 2's per row). Further results in this direction have been recently provided by Cicalese and Milanič [3]. Finally, Fellows et al. [5] introduced the *constrained parsimony haplotyping problem* where the set of haplotypes may not be chosen arbitrarily from $\{0, 1\}^m$ but only from a pool $\tilde{H}$ of plausible haplotypes. Using an intricate dynamic programming algorithm, they extended the fixed-parameter tractability result of Sharan et al. [16] to the constrained case, proving a running time of $k^{O(k^2)} \cdot \text{poly}(m, |\tilde{H}|)$. Jäger et al. [12] recently presented an experimental study of algorithms for computing *all* possible haplotype solutions for a given set of genotypes, where the integer linear programming and branch-and-bound algorithms were sped up using some insights into the combinatorial structure of the haplotype solution, as for example eliminating equal columns from the genotype matrix and recursively decomposing a large problem into smaller ones.

Our contributions are as follows. We simplify and improve the fixed-parameter tractability results of Sharan et al. [16] and Fellows et al. [5] by proposing

fixed-parameter algorithms for the constrained and unconstrained versions of parsimony haplotyping that run in $k^{4k} \cdot \text{poly}(m, |\tilde{H}|)$ time, which is a significant exponential speed-up over previous algorithms. Moreover, we develop polynomial-time data reduction rules that yield a problem kernel of size at most $2^k k^2$ for the unconstrained case. A combinatorially demanding part is to show that the problems become polynomial-time solvable if we require that the given set of genotypes is complete in the sense that it contains all genotypes that can be resolved by some pair of haplotypes in the solution set H. We call this special case *induced parsimony haplotyping*, and we distinguish between the case that the genotypes are given as a multiset (note that different pairs of haplotypes may resolve the same genotype), or just as a set without multiplicities. We show that, while there may be an exponential number of optimal solutions in the general case, there can be at most two optimal solutions in the induced case. For both induced cases, unconstrained and constrained, we propose algorithms running in $O(k \cdot m \cdot |G|)$ and $O(k \cdot m \cdot (|G| + |\tilde{H}|))$ time, respectively. Note that these polynomial-time solvable cases stand in sharp contrast to previous polynomial-time solvable cases [3,14,16,11], all of which require a bound on the number of 2's in the genotype matrix.

2 Preliminaries and Definitions

Throughout this paper, we consider *genotypes* as strings of length m over the alphabet $\{0, 1, 2\}$, while *haplotypes* are considered as strings of length m over the alphabet $\{0, 1\}$. If s is a string, then $s[i]$ denotes the letter of s at position i. This applies to both haplotypes and genotypes. Two haplotypes h_1 and h_2 *resolve* a genotype g, denoted by $\text{res}(h_1, h_2) = g$, if, for all positions i, either $h_1[i] = h_2[i] = g[i]$, or $g[i] = 2$ and $h_1[i] \neq h_2[i]$.

For a given set H of haplotypes, let $\text{res}(H) := \{\text{res}(h_1, h_2) \mid h_1, h_2 \in H\}$ denote the set of genotypes resolved by H and $\text{mres}(H)$ the multiset of genotypes resolved by H (the multiplicity of a genotype g in $\text{mres}(H)$ corresponds to the number of pairs of haplotypes in H resolving g). We also write $\text{res}(h, H)$ ($\text{mres}(h, H)$) for the (multi)set of genotypes resolved by h with all haplotypes in H. We say a set H of haplotypes *resolves* a given set G of genotypes if $G \subseteq \text{res}(H)$, and H *induces* G if $\text{res}(H) = G$. If G is a multiset, we similarly require $G \subseteq \text{mres}(H)$ and $\text{mres}(H) = G$, respectively. A haplotype h is *consistent* with a genotype g if $h[i] = g[i]$ for all positions i with $g[i] \neq 2$.

We refer to the monographs [4,6,15] for any details concerning parameterized algorithmics and the survey [9] for an overview on problem kernelization.

We consider the following haplotype inference problems parameterized with the size of the haplotype set H to be computed:

HAPLOTYPE INFERENCE BY PARSIMONY (HIP):

Input: A set G of length-m genotypes and an integer $k \geq 0$.
Question: Is there a set H of length-m haplotypes such that $|H| \leq k$ and $G \subseteq \text{res}(H)$?

In CONSTRAINED HAPLOTYPE INFERENCE BY PARSIMONY (CHIP) the input additionally contains a set $\tilde{H}$ of length-m haplotypes and the task is to find a set of at

most k haplotypes from $\tilde{H}$ resolving G. Note that with k haplotypes one can resolve at most $\binom{k}{2} + k$ genotypes. Hence, throughout this paper, we assume that $|G|$ is bounded by $\binom{k}{2} + k$.

In this paper, we introduce the "induced case" of constrained and unconstrained parsimony haplotyping. To simplify the presentation of the results for the induced case, in Section 3 we assume that each genotype contains at least one letter 2. Then, we need two different haplotypes to resolve a genotype. Hence, in the induced case, we assume that $\mathrm{res}\,(H)$ does not contain an element of H. We claim without proof that our algorithms in Section 3 can be adapted to instances without these restrictions.

Formally, INDUCED (CONSTRAINED) HAPLOTYPE INFERENCE BY PARSIMONY, (C)IHIP for short, is defined as follows. Given a set G of length-m genotypes (and a set $\tilde{H}$ of length-m haplotypes), the task is to find a set $H(\subseteq \tilde{H})$ of length-m haplotypes such that $G = \mathrm{res}\,(H)$?

Due to the lack of space, some proofs are deferred to a full version of this paper.

3 Induced Haplotype Inference by Parsimony

The main result of this section is that one can solve INDUCED HAPLOTYPE INFERENCE BY PARSIMONY (IHIP) and INDUCED CONSTRAINED HAPLOTYPE INFERENCE BY PARSIMONY (ICHIP) in $O(k \cdot m \cdot |G|)$ and $O(k \cdot m \cdot |G| \cdot |\tilde{H}|)$ time, respectively. In the first paragraph, we consider the following special case of IHIP: given a multiset of $\binom{k}{2}$ length-m genotypes (which are not necessarily distinct), is there a multiset of k length-m haplotypes inducing them? By allowing genotype multisets, we enforce that the input contains information about how often each genotype is resolved by the haplotypes. This allows us to observe a special structure in the input, which makes it easier to present our results. In the second paragraph, we extend our findings to the case that the input genotypes are given as a set, that is, without multiplicities. In this case, we might have some genotypes that are resolved multiple times. However, we do not know in advance which of the input genotypes would be resolved more than once. This makes the set case more delicate than the multiset case. In fact, the set case can be interpreted as a generalization of the multiset case. However, being easier to present, we focus on the multiset case first. Recall that, for the ease of presentation, throughout this section we assume that every genotype contains at least one letter 2 and that $\mathrm{res}\,(H)$ and $\mathrm{mres}\,(H)$ do not intersect H.

The Multiset Case. In this paragraph, we show that one can solve INDUCED HAPLOTYPE INFERENCE BY PARSIMONY (IHIP) in $O(k \cdot m \cdot |G|)$ time in the multiset case. This easily generalizes to the constrained case.

We need the following notation. Let $\#_x(i)$ denote the number of genotypes in G which have letter x at position i, for $x \in \{0, 1, 2\}$. We start with a simple structural observation that must be fulfilled by yes-instances. If G is a yes-instance for IHIP, then the set of genotypes restricted to their first positions (i.e., single-letter genotypes) is also a yes-instance. By a simple column-exchange argument, this extends to all positions, implying the following observation (see Fig. 1 for an example).

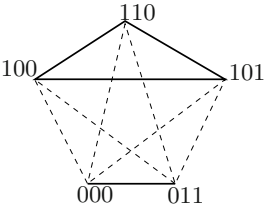


Fig. 1. An example illustrating the Number Condition with $k_0 = 2$ and $k_1 = 3$. Vertices are labeled with haplotypes. Solid edges are genotypes starting with 0 or 1 while dashed edges are genotypes starting with 2.

Observation 1 ("Number Condition") *If a multiset of genotypes is a yes-instance for IHIP, then, for each position $i \in \{1, \ldots, m\}$, there exist two integers $k_0 \geq 0$ and $k_1 \geq 0$ such that $k = k_0 + k_1$, $\#_0(i) = \binom{k_0}{2}$, $\#_1(i) = \binom{k_1}{2}$, and $\#_2(i) = k_0 \cdot k_1$.*

The next lemma is the basis for recursively solving IHIP. For the ease of presentation, we define the operation $\oplus$. It can be applied to a haplotype h and a genotype g if, for all $i \in \{1, \ldots, m\}$, either $h[i] = g[i]$ or $g[i] = 2$. It produces the unique length-m haplotype $h' := h \oplus g$ such that $\operatorname{res}(h, h') = g$. We further define $i^\star$ as the first position for which there are genotypes $g, g' \in G$ with $g[i^\star] \neq g'[i^\star]$. Furthermore, for all $x \in \{0, 1, 2\}$, we denote the set of all genotypes $g \in G$ with $g[i^\star] = x$ as G_x. Clearly, any solution for G can be partitioned into a solution for G_0 and a solution for G_1, as formalized by Lemma 1.

Lemma 1. *Let G be a multiset of genotypes such that not all genotypes in G are identical. Let H be a set of haplotypes inducing G. For $x \in \{0, 1\}$, let H_x denote the haplotypes in H with x at position $i^\star$. Then, H_0 induces G_0, H_1 induces G_1, and G_2 is exactly the multiset of genotypes resolved by taking each time one haplotype from H_0 and one haplotype from H_1. Moreover, $H_0 \cap H_1 = \emptyset$.*

The function `solve(`G`)` (see Alg. 1) recursively computes a solution for G, with the base cases provided by the next two lemmas. Lemma 2 identifies two cases for which there exists a unique solution for G, which in each case can be computed in polynomial time.

Lemma 2. *Assume that $|G| \geq 2$. If all genotypes in G are identical or if $G_x = \emptyset$ for some $x \in \{0, 1\}$, then there exists at most one solution for G. Moreover, in $O(|G| \cdot m)$ time, one can compute a solution or report that G is a no-instance.*

Proof. First we consider the case that all genotypes are identical. Since every genotype has letter 2, Lemma 1 implies that G is a no-instance.

Now, assume that not all genotypes are identical and $G_x = \emptyset$ for some $x \in \{0, 1\}$. Without loss of generality, $G_0 = \emptyset$ and $G_1 \neq \emptyset$. By definition of $i^\star$, $G_2 \neq \emptyset$. Note that in a solution for G there can be at most one haplotype having letter 0 at position $i^\star$ (otherwise, we have a contradiction to the fact that $G_0 = \emptyset$). Moreover, there must exist at least one haplotype with 0 at position $i^\star$ (otherwise one cannot resolve the haplotypes

Function `solve(`G`)`
Input: A multiset of genotypes $G \subseteq \{0,1,2\}^m$.
Output: A set $\mathcal{H}$ containing at most two multisets of haplotypes each of which induces G, if G is a yes-instance; otherwise "no".

```
1  begin
2      if all genotypes in G are identical or G_x = ∅ for some x ∈ {0,1} then
3          return the unique solution {H} (see Lemma 2);
4      else if |G_0| = 1 and |G_1| = 1 then
5          return the at most two solutions {H, H'} (see Lemma 3);
6      else
7          Choose x ∈ {0,1} such that |G_x| > 1 and |G_x| is minimal;
8          H ← solve(G_x);
9          forall H ∈ H do replace H with MultisetExtend(H, G, G_2) in H;
10         if H contains only the empty set then return "no";
11         return H;
12     end
13 end
```

Algorithm 1. solve(G) recursively computes all (at most two) solutions for G

in G_2). Thus, in any solution H for G, there must exist a unique haplotype $h \in H$ with $h[i^\star] = 0$; further, $G_2 = \operatorname{mres}(h, H \setminus \{h\})$. One can now infer all haplotypes as follows. Clearly, one can answer "no" if there is an i, $1 \leq i \leq m$, such that both letters 0 and 1 appear at position i of the genotypes in G_2. If there is a position i and a $g \in G_2$ with $g[i] \neq 2$, then one can set $h[i] := g[i]$; otherwise, to have a solution for G, all genotypes in G_1 must have the same letter $y \in \{0,1\}$ at this position, so one can set $h[i] := 1 - y$. With h settled, one can easily determine the haplotypes h' with $h'[i^\star] = 1$ (these are the haplotypes $g \oplus h$ for $g \in G_2$). Finally, one has to make sure that all these haplotypes induce G. If not, then the input instance is a no-instance. The running time of this procedure is $O(|G| \cdot m)$. □

Next, we show that there are at most two solutions for G if each of G_0 and G_1 contains only a single genotype.

Lemma 3. *If $|G_0| = 1$ and $|G_1| = 1$, then there are at most two solutions for G. Moreover, in $O(m)$ time, one can compute these solutions or report that G is a no-instance.*

Proof. Let g_0 and g_1 be the genotypes in G_0 and G_1, respectively. By Lemma 1, two pairs of haplotypes are required to resolve them, denoted by h_0^0 and h_0^1 (resolving g_0), and h_1^0 and h_1^1 (resolving g_1). If $|G_2| \neq 4$, then return "no" (see Observation 1); otherwise, let $G_2 = \{g_2, g_3, g_4, g_5\}$. If none of g_0 and g_1 contains letter 2, then the haplotypes are easily constructed (they are equal to the respective genotype). Otherwise, let i be the first position where g_0 or g_1 has letter 2, say $g_0[i] = 2$. Without loss of generality, let $h_0^0[i] := 0$ and $h_0^1[i] := 1$. We consider the following two cases:

Case 1: $g_1[i] \neq 2$.
Without loss of generality, let $g_1[i] = 0$. Then, two of the genotypes in G_2 must have 0 at position i and the other two must have 2 at position i; otherwise, return "no". Without loss of generality, let $g_2[i] = g_3[i] = 0$ and $g_4[i] = g_5[i] = 2$. Since g_2 and g_3 must be resolved by h_0^0, one can uniquely determine h_0^0 as follows. Consider any position j. If $g_2[j] \neq 2$ and $g_3[j] \neq 2$, then they must both be equal (if not, then return "no"). In this case, let $h_0^0[j] = g_2[j]$. If exactly one of $g_2[j]$ and $g_3[j]$ is equal to 2, say $g_3[j] = 2$, then let $h_0^0[j] = g_2[j]$. If $g_2[j] = g_3[j] = 2$, then we know that $g_1[j] \neq 2$ (otherwise, return "no") and thus $h_0^0[j] := 1 - g_1[j]$. Finally, let $h_0^1 := h_0^0 \oplus g_0$, $h_1^0 := h_0^0 \oplus g_2$, and $h_1^1 := h_0^0 \oplus g_3$. If these haplotypes also correctly resolve g_1, g_4, and g_5, then we have a unique solution for G, otherwise return "no".

Case 2: $g_0[i] = g_1[i] = 2$.
There is a genotype in G_2 having 0 at position i and another having 1 at position i (otherwise, return "no"). Without loss of generality, let $g_2[i] = 0$, $g_3[i] = 1$, $h_1^0[i] := 0$, and $h_1^1[i] := 1$. Then, $g_4[i] = g_5[i] = 2$ and $g_2 = \operatorname{res}(h_0^0, h_1^0)$ and $g_3 = \operatorname{res}(h_0^1, h_1^1)$. Now there are two possibilities to resolve g_4 and g_5. Either $g_4 = \operatorname{res}(h_0^1, h_1^0)$ and $g_5 = \operatorname{res}(h_0^0, h_1^1)$, or $g_4 = \operatorname{res}(h_0^0, h_1^1)$ and $g_5 = \operatorname{res}(h_0^1, h_1^0)$. By choosing one of these two possibilities, all four haplotypes are fixed. Thus, there are at most two solutions for G.

Note that there are only six genotypes. Thus, for every position the computations are clearly doable in constant time. Hence, the whole procedure runs in $O(m)$ time. □

The next two lemmas show that one can solve an IHIP instance recursively if neither Lemma 2 nor Lemma 3 applies. That is, we now assume that not all genotypes are identical and we have $|G_x| > 1$ for some $x \in \{0, 1\}$. We show that, given a solution for G_x, one can uniquely extend this solution to a solution for G, or decide that G is a no-instance, leading to function `MultisetExtend` (see Alg. 2)

Lemma 4. *Let $|G_x| > 1$ for some $x \in \{0, 1\}$, let H_x be a multiset of haplotypes inducing G_x, and let g be a genotype in G_2 with the smallest number of 2's. If G is induced by H with $H_x \subseteq H$, then all haplotypes in H_x consistent with g must be identical.*

Proof. Without loss of generality, we assume that $|G_0| > 1$. Suppose that there is an H with $H_x \subseteq H$ inducing G. Since $g[i^\star] = 2$, there must be a haplotype $h_1 \in H_x$ and a haplotype $h_2 \in H \setminus H_x$ resolving g. Clearly, h_1 and h_2 are consistent with g. We show that there is no other haplotype $h \in H_x$ such that $h \neq h_1$ and h is consistent with g. For the sake of contradiction, assume that there is such a haplotype h. First, note that h, h_1, and h_2 are consistent with g and hence identical at positions where g does not have letter 2. Since $h \neq h_1$, h differs from h_1 in at least one of the positions where g has letter 2. Thus, h_2 (which together with h_1 resolves g and hence is the complement of h_1 at the positions where g has letter 2) must have the same letter as h at some position where h_1 and h_2 differ. This implies that $\operatorname{res}(h, h_2) \in G_2$ has fewer 2's than g, contradicting the choice of g. □

Lemma 5. *Let $|G_x| > 1$ for some $x \in \{0, 1\}$, and let H_x be a multiset of haplotypes inducing G_x. If G is induced by H with $H_x \subseteq H$, then H is uniquely determined and function* `MultisetExtend` *(see Alg. 2) computes H in $O(|H_x| \cdot |G_2| \cdot m)$ time.*

Function `MultisetExtend(`H_x, G, G_2`)`
Input: A haplotype multiset H_x inducing G_x for some $x \in \{0, 1\}$, and a multiset G_2 of genotypes.
Output: A haplotype multiset H inducing G with $H_x \subseteq H$, if one exists; otherwise an empty set.

```
1  begin
2      H := H_x;
3      while G_2 ≠ ∅ do
4          Choose a g ∈ G_2 with smallest number of 2's;
5          Choose an h ∈ H_x consistent with g;
6          h' := h ⊕ g;
7          H := H ∪ {h'};
8          G' := {g' | ∃h'' ∈ H_x : g' = res(h', h'')};
9          if G' ⊈ G_2 then return ∅;
10         G_2 := G_2 \ G';
11     end
12     if mres(H) = G then return H;
13     else return ∅;
14 end
```

Algorithm 2. An algorithm to extend a solution for G_x to G in the multiset case

Proof. The correctness of lines 4–7 of `MultisetExtend` (see Alg. 2) follows from Lemma 4. Since including $h' := h \oplus g$ in H is the only choice, the genotypes resolved by h' and other haplotypes in H_x should also be in G_2; otherwise, no solution exists. Thus, lines 8 and 9 of `MultisetExtend` are correct. Line 10 of `MultisetExtend` safely removes the genotypes resolved by h' from G_2. The next while-iteration proceeds to find the next pair consisting of a haplotype h and a genotype $g \in G_2$ satisfying Lemma 4. If there is a solution for G comprising H_x, then we must end up with an empty G_2. Moreover, $H \setminus H_x$ should resolve all genotypes in G_{1-x} and, together with H_x, the genotypes in G_2; this is examined in line 12 of `MultisetExtend`. Thus, the function `MultisetExtend` is correct. By Lemma 4, the solution H with $H_x \subseteq H$ is unique.

Concerning the running time, note that the most time-consuming part of the function is to find the consistent haplotypes in H_x for a given genotype in G_2. This can be done in $O(|H_x| \cdot |G_2| \cdot m)$ time by iterating over all haplotypes in H_x and for each haplotype over all genotypes in G_2. □

Putting all together, we obtain the main theorem of this paragraph.

Theorem 1. *In case of a* multiset G *of length-*m *genotypes,* INDUCED HAPLOTYPE INFERENCE BY PARSIMONY *and* CONSTRAINED INDUCED HAPLOTYPE INFERENCE BY PARSIMONY *can be solved in* $O(k \cdot m \cdot |G|)$ *and* $O(k \cdot m \cdot (|G| + |\tilde{H}|))$ *time, respectively.*

Proof. (Sketch) We show that the algorithm `solve(`G`)` (see Alg. 1) is correct. If all genotypes are identical or $G_x = \emptyset$, for some $x \in \{0, 1\}$, then the correctness follows

from Lemma 2. Hence, in the following, assume that not all genotypes are identical, $G_0 \neq \emptyset$, and $G_1 \neq \emptyset$. Distinguish the cases that $|G_0| = |G_1| = 1$ and $|G_x| > 1$, for some $x \in \{0, 1\}$. In the case that $|G_0| = |G_1| = 1$, one can compute the solutions (at most two) for G using Lemma 3. In the other case, for some $x \in \{0, 1\}$, it holds that $|G_x| > 1$ and $|G_{1-x}| > 0$. Without loss of generality, assume $|G_0| > 1$. By Lemma 1, a solution for G consists of a solution H_0 for G_0 and a solution H_1 for G_1, and $H_0 \cap H_1 = \emptyset$. Since one tries to extend every solution for G_0 and these extensions are unique by Lemma 5, one will find every possible solution for G. Since the base cases have at most two solutions and extensions are uniquely determined by Lemma 5, there exist at most two solutions for G. In the constrained case, one only needs to check whether one of the computed solutions is in the given set of haplotypes. The claimed running time follows from Lemmas 2, 3, and 5. □

The Set Case. If the input is not a multiset, but a set G of genotypes, that is, all genotypes in G are pairwise distinct, then the Number Condition (Observation 1) does not necessarily hold. Consider the haplotype set $H = \{000, 001, 110, 111\}$ which induces the set $\mathrm{res}\,(H) = \{002, 112, 221, 220, 222\}$, but also induces the multiset $\mathrm{mres}\,(H) = \{002, 112, 221, 220, 222, 222\}$ (observe that $\mathrm{res}\,(000, 111) = \mathrm{res}\,(001, 110) = 222$). The problem is that we cannot directly infer from G which genotypes should be resolved more than once. However, many properties of the multiset case (as for example Lemmas 1,2, and 3) carry over to the set case, so we only need a moderate modification of the multiset algorithm to solve the set case. More specifically, the key to solve the set case is to adapt function `MultisetExtend` (all details are deferred to the long version of this paper).

Theorem 2. *In case of a* set G *of length-*m *genotypes,* INDUCED HAPLOTYPE INFERENCE BY PARSIMONY *and* CONSTRAINED INDUCED HAPLOTYPE INFERENCE BY PARSIMONY *can be solved in* $O(k \cdot m \cdot |G|)$ *and* $O(k \cdot m \cdot (|G| + |\tilde{H}|))$ *time, respectively.*

4 General Haplotype Inference by Parsimony

This section contains an algorithm to solve the general parsimony haplotyping problem for the unconstrained and the constrained versions in $O(k^{4k+1} \cdot m)$ and $O(k^{4k+1} \cdot m \cdot |\tilde{H}|)$ time, respectively, improving and partially simplifying previous fixed-parameter tractability results [16,5]. In addition, we provide a simple kernelization.

We start with some preliminary considerations. Given a set of haplotypes resolving a given set of genotypes, the relation between the haplotypes and the genotypes can be depicted by an undirected graph, the *solution graph*, in which the edges are labeled by the genotypes and every vertex v is labeled by a haplotype h_v. If an edge $\{u, v\}$ is labeled by genotype g, we require that $g = \mathrm{res}\,(h_u, h_v)$. We call such a vertex/edge labeling *consistent*. If only the edges are labeled, the graph is an *inference graph* (because it allows us to infer all the haplotypes). Solution graphs and inference graphs may contain loops.

In what follows, assume that the input is a yes-instance, i.e., a solution graph exists. Intuitively, our algorithm "guesses" an inference graph for G (by enumerating all possible such graphs) and then infers the haplotypes from the genotype labels on the edges.

Input: A set of genotypes $G \subseteq \{0,1,2\}^m$ and an integer $k \geq 0$.
Output: Either a set of haplotypes H with $|H| \leq k$ and $G \subseteq \operatorname{res}(H)$, or "no" if there is no solution of size at most k.

```
1  forall size-k subsets G' ⊆ G do
2      forall inference graphs Γ for G' on k vertices and k edges do
3          forall non-bipartite connected components of Γ do
4              if possible, compute the labels of all vertices of the component (Lemma 7),
               otherwise try the next inference graph (goto line 2);
5          end
6          forall bipartite connected components of Γ do
7              if possible, compute a consistent vertex labeling for the component
               (Lemma 8), otherwise try the next inference graph (goto line 2);
8          end
9          Let H denote the inferred haplotypes (vertex labels);
10         if G ⊆ res(H) then return H;
11     end
12 end
13 return "no";
```

Algorithm 3. An algorithm solving HIP in $O(k^{4k+1} \cdot m)$ time

To this end, it guesses for every connected component of the solution graph a spanning subgraph with edges labeled by some of the genotypes in G in such a way that we have enough information at hand to infer the haplotypes. Then, one has to solve the following subproblem: Given an inference graph for a subset of genotypes of G, does there exist a consistent vertex labeling? The next three lemmas show how to solve this subproblem by separately considering the connected components of the inference graphs.

Lemma 6. *Let G be a set of genotypes and let $\Gamma = (V, E)$ be a connected inference graph for G. For each position i, $1 \leq i \leq m$, if there is a genotype $g \in G$ with $g[i] \neq 2$, then one can, in $O(|V| + |E|)$ time, uniquely infer the letters of all haplotypes at position i or report that there is no consistent vertex labeling.*

Lemma 7. *Let $\Gamma = (V, E)$ be a connected inference graph for a set G of genotypes that contains an odd-length cycle. Then, there exists at most one consistent vertex labeling. Furthermore, one can compute in $O(m \cdot (|V| + |E|))$ time a consistent vertex labeling or report that no consistent vertex labeling exists.*

Lemma 8. *Let $\Gamma = (V_a, V_b, E)$ be a connected bipartite inference graph for a set G of length-m genotypes. Let $u \in V_a$ and $w \in V_b$ be arbitrarily chosen. Then,*

1. *one can compute in $O(m \cdot (|V_a| + |V_b| + |E|))$ time a consistent vertex labeling or report that no consistent vertex labeling exists, and*
2. *the genotypes resolved by h_u and h_w are identical for every consistent vertex labeling.*

Next, we describe the algorithm for the unconstrained version (HIP), see Alg. 3. To solve HIP, we could enumerate all inference graphs for G and then find the vertex

labeling using Lemmas 7 and 8. However, to be more efficient, we first select a size-k subset of genotypes (line 1 of Alg. 3), and then we enumerate all inference graphs on k vertices containing exactly k edges labeled by the k chosen genotypes (line 2 of Alg. 3). Assume that there exists a solution graph for G. Of all inference graphs on k vertices and k edges consider one with the following properties:

- it contains a spanning subgraph of every connected component of the solution graph, and
- the spanning subgraph of any non-bipartite connected component contains an odd cycle (thus, the bipartite components of the inference graph are exactly the bipartite components of the solution graph).

Obviously, this inference graph exists and is considered by Alg. 3. By Lemma 7, we can uniquely infer the vertex labels for all connected components of the inference graph containing an odd cycle. For every bipartite component, we can get a consistent vertex labeling from Lemma 8. In such a bipartite component, for any two vertices $u \in V_a$ and $v \in V_b$, the genotypes resolved by h_u and h_v are identical for every consistent vertex labeling. Thus, the haplotypes resolve all genotypes contained in the respective (bipartite) component of the solution graph. In summary, if the given instance is a yes-instance, then our algorithm will find a set of at most k haplotypes resolving the given genotypes.

Theorem 3. HAPLOTYPE INFERENCE BY PARSIMONY *and* CONSTRAINED HAPLOTYPE INFERENCE BY PARSIMONY *can be solved in* $O(k^{4k+1} \cdot m)$ *and* $O(k^{4k+1} \cdot m \cdot |\tilde{H}|)$ *time, respectively.*

Proof. (Sketch) We first consider the unconstrained case. By the discussion above, Alg. 3 correctly solves HIP. It remains to analyze its running time. First, there are $O(\binom{|G|}{k})$ size-k subsets G' of G. Second, there are $O(k^{2k})$ inference graphs on k vertices containing exactly k edges labeled by the genotypes in G' because for every genotype $g \in G'$ we have k^2 choices for the endpoints (loops are allowed) of the edge labeled by g. For each of those inference graphs, applying Lemma 7 and Lemma 8 to its connected components takes $O(k \cdot m)$ time. Hence, the overall running time of Alg. 3 sums up to $O(\binom{|G|}{k} \cdot k^{2k} \cdot m \cdot k)$. Since $|G| \leq k^2$, the running time can be bounded by $O(k^{4k+1} \cdot m)$.

One can easily adapt Alg. 3 to solve CHIP as follows. As before, one enumerates all size-k subsets $G' \subseteq G$ and all inference graphs for G'. Since, by Lemma 7, the vertex labels for the connected components containing an odd cycle are uniquely determined, one only has to check whether the inferred haplotypes are contained in the given haplotype pool $\tilde{H}$ (otherwise, try the next inference graph). Basically, the only difference is how to proceed with the bipartite components of the inference graph. Let (W, F) be a connected bipartite component of the current inference graph. Instead of choosing an arbitrary consistent vertex labeling as done in Lemma 8, proceed as follows. Choose an arbitrary vertex $v \in W$ and check for every haplotype $h \in \tilde{H}$ whether there exists a consistent vertex labeling for this component where v is labeled by h. Note that fixing the vertex label for v implies the existence of at most one consistent vertex labeling of (W, F). If it exists, this labeling can be computed by a depth-first traversal starting at v. If for a haplotype h there exists a consistent vertex labeling of (W, F) such that

all labels are contained in $\tilde{H}$, then proceed with the next bipartite component. Otherwise, one can conclude that for the current inference graph there is no consistent vertex labeling using only the given haplotypes, and, hence, one can proceed with the next inference graph. The correctness and the claimed running time follow by almost the same arguments as in the unconstrained case. □

Problem Kernelization. In this paragraph, we show that HIP admits an exponential-size problem kernel. To this end, we assume the input G to be in the matrix representation that is mentioned in the introduction; that is, each row represents a genotype while each column represents a position. Since it is obvious that we can upper-bound the number n of genotypes in the input by k^2, it remains to bound the number m of columns (positions) in the input. The idea behind the following data reduction rule is that we can safely delete a column if there is another column that is identical. By applying this rule exhaustively, we can bound the number of columns by 2^k.

Reduction Rule. *Let (G, k) be an instance of* HIP. *If two columns of G are equal, then delete one of them.*

The correctness of the reduction rule follows by the observation that, given at most k haplotypes resolving the genotypes in the reduced instance, we can easily find a solution for the original instance by copying the respective haplotype positions. Next, we bound the number of columns.

Lemma 9. *Let (G, k) be a yes-instance of* HIP *that is reduced with respect to the reduction rule. Then, G has at most 2^k columns.*

Proof. Let H denote a matrix of k haplotypes resolving G. It is obvious that if two columns i and j of H are equal, then columns i and j of G are equal. Now, since G does not contain a pair of equal columns, neither does H. Since there are only 2^k different strings in $\{0, 1\}^k$, it is clear that H cannot contain more than 2^k columns and thus, neither can G. □

Since the number n of genotypes can be bounded by k^2 and the number m of columns can be bounded by 2^k (Lemma 9), one directly obtains Proposition 1.

Proposition 1. HAPLOTYPE INFERENCE BY PARSIMONY *admits a problem kernel of size at most $2^k \cdot k^2$ that can be constructed in $O(n \cdot m \cdot \log m)$ time.*

Plugging Proposition 1 into Theorem 3, we achieve the following.

Corollary 1. *HIP can be solved in $O(k^{4k+1} \cdot 2^k + n \cdot m \cdot \log m)$ time.*

5 Conclusion

We contributed new combinatorial algorithms for parsimony haplotyping with the potential to make the problem more feasible in practice without giving up the demand for optimal solutions. Our results also lead to several new questions for future research. For instance, our kernelization result yields a problem kernel of exponential size. It would

be interesting to know whether a polynomial-size problem kernel exists, which may also be seen in the light of recent breakthrough results on methods to prove the non-existence of polynomial-size kernels [1,7]. A second line of research is to make use of the polynomial-time solvable induced cases to pursue a "distance from triviality" approach [8]. The idea here is to identify and exploit parameters that measure the distance of general instances of parsimony haplotyping to the "trivial" (that is, polynomial-time solvable) induced cases. Research in this direction is underway. A more speculative research direction could be to investigate whether our results on the induced case (with at most two optimal solutions) may be useful in the context of recent research [12] on finding all optimal solutions in the general case. Clearly, it remains an interesting open problem to find a fixed-parameter algorithm for parsimony haplotyping with an exponential factor of the form c^k for some constant c.

References

1. Bodlaender, H.L., Downey, R.G., Fellows, M.R., Hermelin, D.: On problems without polynomial kernels. J. Comput. System Sci. 75(8), 423–434 (2009)
2. Catanzaro, D., Labbé, M.: The pure parsimony haplotyping problem: Overview and computational advances. International Transactions in Operational Research 16(5), 561–584 (2009)
3. Cicalese, F., Milanič, M.: On parsimony haplotyping. Technical Report 2008-04, Universität Bielefeld (2008)
4. Downey, R.G., Fellows, M.R.: Parameterized Complexity. Springer, Heidelberg (1999)
5. Fellows, M.R., Hartman, T., Hermelin, D., Landau, G.M., Rosamond, F.A., Rozenberg, L.: Haplotype inference constrained by plausible haplotype data. In: Kucherov, G., Ukkonen, E. (eds.) CPM 2009. LNCS, vol. 5577, pp. 339–352. Springer, Heidelberg (2009)
6. Flum, J., Grohe, M.: Parameterized Complexity Theory. Springer, Heidelberg (2006)
7. Fortnow, L., Santhanam, R.: Infeasibility of instance compression and succinct PCPs for NP. In: Proc. 40th STOC, pp. 133–142. ACM Press, New York (2008)
8. Guo, J., Hüffner, F., Niedermeier, R.: A structural view on parameterizing problems: Distance from triviality. In: Downey, R.G., Fellows, M.R., Dehne, F. (eds.) IWPEC 2004. LNCS, vol. 3162, pp. 162–173. Springer, Heidelberg (2004)
9. Guo, J., Niedermeier, R.: Invitation to data reduction and problem kernelization. ACM SIGACT News 38(1), 31–45 (2007)
10. Gusfield, D., Orzack, S.H.: Haplotype inference. CRC Handbook on Bioinformatics, ch. 1, pp. 1–25. CRC Press, Boca Raton (2005)
11. van Iersel, L., Keijsper, J., Kelk, S., Stougie, L.: Shorelines of islands of tractability: Algorithms for parsimony and minimum perfect phylogeny haplotyping problems. IEEE/ACM Trans. Comput. Biology Bioinform. 5(2), 301–312 (2008)
12. Jäger, G., Climer, S., Zhang, W.: Complete parsimony haplotype inference problem and algorithms. In: Fiat, A., Sanders, P. (eds.) ESA 2009. LNCS, vol. 5757, pp. 337–348. Springer, Heidelberg (2009)
13. Lancia, G., Pinotti, M.C., Rizzi, R.: Haplotyping populations by pure parsimony: Complexity of exact and approximation algorithms. INFORMS Journal on Computing 16(4), 348–359 (2004)
14. Lancia, G., Rizzi, R.: A polynomial case of the parsimony haplotyping problem. Operations Research Letters 34, 289–295 (2006)
15. Niedermeier, R.: Invitation to Fixed-Parameter Algorithms. Oxford University Press, Oxford (2006)
16. Sharan, R., Halldórsson, B.V., Istrail, S.: Islands of tractability for parsimony haplotyping. IEEE/ACM Trans. Comput. Biology Bioinform. 3(3), 303–311 (2006)

Sampled Longest Common Prefix Array

Jouni Sirén*

Department of Computer Science, University of Helsinki, Finland
jltsiren@cs.helsinki.fi

Abstract. When augmented with the longest common prefix (LCP) array and some other structures, the suffix array can solve many string processing problems in optimal time and space. A compressed representation of the LCP array is also one of the main building blocks in many compressed suffix tree proposals. In this paper, we describe a new compressed LCP representation: the *sampled LCP array*. We show that when used with a compressed suffix array (CSA), the sampled LCP array often offers better time/space trade-offs than the existing alternatives. We also show how to construct the compressed representations of the LCP array directly from a CSA.

1 Introduction

The suffix tree is one of the most important data structures in string processing and bioinformatics. While it solves many problems efficiently, its usefulness is limited by its size: typically 10–20 times the size of the text [17]. Much work has been put on reducing the size, resulting in data structures such as the enhanced suffix array [1] and several variants of the compressed suffix tree [22,21,11,18].

Most of the proposed solutions are based on three structures: 1) the suffix array, listing the suffixes of the text in lexicographic order; 2) the longest common prefix (LCP) array, listing the lengths of the longest common prefixes of lexicographically adjacent suffixes; and 3) a representation of suffix tree topology. While there exists an extensive literature on compressed suffix arrays (CSA)[1] [19], less has been done on compressing the other structures.

Existing proposals to compress the LCP information are based on the permuted LCP (PLCP) array that arranges the entries in text order. While the PLCP array can be compressed, one requires expensive CSA operations to access LCP values through it. In this paper, we describe the sampled LCP array as an alternative to the PLCP-based approaches. Similar to the suffix array samples used in CSAs, the sampled LCP array often offers better time/space trade-offs than the PLCP-based alternatives.

We also modify a recent PLCP construction algorithm [14] to work directly with a compressed suffix array. Using it, we can construct any PLCP representation with negligible working space in addition to the CSA and the PLCP.

* Funded by the Academy of Finland under grant 119815.

[1] In this paper, we use the term *compressed suffix array* to refer to any compressed self-index based on the Burrows-Wheeler transform.

A. Amir and L. Parida (Eds.): CPM 2010, LNCS 6129, pp. 227–237, 2010.

A variant of the algorithm can also be used to construct the sampled LCP array, but requires more working space. While our algorithm is much slower than the alternatives, it is the first LCP construction algorithm that does not require access to the text and the suffix array. This is especially important for large texts, as the suffix array may not be available or the text might not fit into memory.

We begin with basic definitions and background information in Sect. 2. Section 3 is a summary of previous compressed LCP representations. In Sect. 4, we show how to build the PLCP array directly from a CSA. We describe our sampled LCP array in Sect. 5. Section 6 contains experimental evaluation of our proposals. We finish with conclusions and discussion on future work in Sect. 7.

2 Background

A *string* $S = S[1, n]$ is a *sequence* of *characters* from *alphabet* $\Sigma = \{1, 2, \ldots, \sigma\}$. A *substring* of S is written as $S[i, j]$. A substring of type $S[1, j]$ is called a *prefix*, while a substring of type $S[i, n]$ is called a *suffix*. A *text* string $T = T[1, n]$ is a string terminated by $T[n] = \$ \notin \Sigma$ with lexicographic value 0. The *lexicographic order* "$<$" among strings is defined in the usual way.

The *suffix array (SA)* of text $T[1, n]$ is an array of pointers $\mathsf{SA}[1, n]$ to the suffixes of T in lexicographic order. As an abstract data type, a suffix array is any data structure with similar functionality as the concrete suffix array. This can be defined by an efficient support for the following operations: (a) *count* the number of occurrences of a *pattern* in the text; (b) *locate* these occurrences (or more generally, retrieve a suffix array value); and (c) *display* any substring of T.

Compressed suffix arrays (CSA) [12,8] support these operations. Their compression is based on the *Burrows-Wheeler transform (BWT)* [3], a permutation of the text related to the SA. The BWT of text T is a sequence $L[1, n]$ such that $L[i] = T[\mathsf{SA}[i] - 1]$, if $\mathsf{SA}[i] > 1$, and $L[i] = T[n] = \$$ otherwise.

The Burrows-Wheeler transform is reversible. The reverse transform is based on a permutation called *LF-mapping* [3,8]. Let $C[1, \sigma]$ be an array such that $C[c]$ is the number of characters in $\{\$, 1, 2, \ldots, c - 1\}$ occurring in the text. For convenience, we also define $C[0] = 0$ and $C[\sigma + 1] = n$. By using this array and the sequence L, we define LF-mapping as $LF(i) = C[L[i]] + rank_{L[i]}(L, i)$, where $rank_c(L, i)$ is the number of occurrences of character c in prefix $L[1, i]$.

The inverse of LF-mapping is $\Psi(i) = select_c(L, i - C[c])$, where c is the highest value with $C[c] < i$, and $select_c(L, j)$ is the position of the jth occurrence of character c in L [12]. By its definition, function Ψ is strictly increasing in the range $\Psi_c = [C[c] + 1, C[c + 1]]$ for every $c \in \Sigma$. Additionally, $T[\mathsf{SA}[i]] = c$ and $L[\Psi(i)] = c$ for every $i \in \Psi_c$.

These functions form the backbone of CSAs. As $\mathsf{SA}[LF(i)] = \mathsf{SA}[i] - 1$ [8] and hence $\mathsf{SA}[\Psi(i)] = \mathsf{SA}[i] + 1$, we can use these functions to move the suffix array position backward and forward in the sequence. Both of the functions can be efficiently implemented by adding some extra information to a compressed representation of the BWT. Standard techniques [19] to support suffix array operations include *backward searching* [8] for *count*, and adding a sample of suffix array values for *locate* and *display*.

Let $lcp(A, B)$ be the length of the longest common prefix of sequences A and B. The *longest common prefix (LCP) array* of text $T[1, n]$ is the array $\mathsf{LCP}[1, n]$ such that $\mathsf{LCP}[1] = 0$ and $\mathsf{LCP}[i] = lcp(T[\mathsf{SA}[i-1], n], T[\mathsf{SA}[i], n])$ for $i > 1$. The array requires $n \log n$ bits of space, and can be constructed in $O(n)$ time [15,14].

3 Previous Compressed LCP Representations

We can exploit the redundancy in LCP values by reordering them in text order. This results in the *permuted LCP (PLCP) array*, where $\mathsf{PLCP}[\mathsf{SA}[i]] = \mathsf{LCP}[i]$. The following lemma describes a key property of the PLCP array.

Lemma 1 ([15,14]). *For every* $i \in \{2, \ldots, n\}$, $\mathsf{PLCP}[i] \geq \mathsf{PLCP}[i-1] - 1$.

As the values $\mathsf{PLCP}[i] + 2i$ form a strictly increasing sequence, we can store the array in a bit vector of length $2n$ [22]. Various schemes exist to represent this bit vector in a succinct or compressed form [22,11,18].

Space-efficiency can also be achieved by sampling every qth PLCP value, and deriving the missing values when needed [16]. Assume we have sampled $\mathsf{PLCP}[aq]$ and $\mathsf{PLCP}[(a+1)q]$, and we want to determine $\mathsf{PLCP}[aq+b]$ for some $b < q$. Lemma 1 states that $\mathsf{PLCP}[aq] - b \leq \mathsf{PLCP}[aq+b] \leq \mathsf{PLCP}[(a+1)q] + q - b$, so at most $q + \mathsf{PLCP}[(a+1)q] - \mathsf{PLCP}[aq]$ character comparisons are required to determine the missing value. The average number of comparisons over all entries is $O(q)$ [14]. By carefully selecting the sampled positions, we can store the samples in $o(n)$ bits, while requiring only $O(\log^{\delta} n)$ comparisons in the worst case for any $0 < \delta \leq 1$ [10].

Unfortunately these compressed representations are not very suitable for use with CSAs. The reason is that the LCP values are accessed through suffix array values, and *locate* is an expensive operation in CSAs. In addition to that, sampled PLCP arrays require access to the text, using the similarly expensive *display*.

Assume that a CSA has SA sample rate d, and that it computes $\Psi(\cdot)$ in time t_Ψ. To retrieve $\mathsf{SA}[i]$, we compute $i, \Psi(i), \Psi^2(i), \ldots$, until we find a sampled suffix array value. If the sampled value was $\mathsf{SA}[\Psi^k(i)] = j$, then $\mathsf{SA}[i] = j - k$. We find a sample in at most d steps, so the time complexity for *locate* is $O(d \cdot t_\Psi)$. Similarly, to retrieve a substring $T[i, i+l]$, we use the samples to get $\mathsf{SA}^{-1}[d \cdot \lfloor \frac{i}{d} \rfloor]$. Then we iterate the function Ψ until we reach text position $i + l$. This takes at most $d + l$ iterations, making the time complexity for *display* $O((d + l) \cdot t_\Psi)$. From these bounds, we get the PLCP access times shown in Table 1.[2]

Depending on the type of index used, t_ψ varies from $O(1)$ to $O(\log n)$ in the worst case [19], and is close to 1 microsecond for the fastest indexes in practice [7,18]. This is significant enough that it makes sense to keep t_Ψ in Table 1.

The only (P)LCP representation so far that is especially designed for use with CSAs is Fischer's Wee LCP [10] that is basically the *select* structure from Sadakane's bit vector representation [22]. When the bit vector itself would be required to answer a query, some characters of two lexicographically adjacent

[2] Some CSAs use LF-mapping instead of Ψ, but similar results apply to them as well.

Table 1. Time/space trade-offs for (P)LCP representations. R is the number of equal letter runs in BWT, q is the PLCP sample rate, and $0 < \delta \leq 1$ is a parameter. The numbers for CSA assume Ψ access time t_Ψ and SA sample rate d.

Representation	Space (bits)	Access times	
		Using SA	Using CSA
LCP	$n \log n$	$O(1)$	$O(1)$
PLCP [22]	$2n + o(n)$	$O(1)$	$O(d \cdot t_\Psi)$
PLCP [11]	$2R \log \frac{n}{R} + O(R) + o(n)$	$O(1)$	$O(d \cdot t_\Psi)$
PLCP [18]	$2R \log \frac{n}{R} + O(R \log \log \frac{n}{R})$	$O(\log \log n)$	$O(d \cdot t_\Psi + \log \log n)$
Sampled PLCP [10]	$o(n)$	$O(\log^\delta n)$	$O((d + \log^\delta n) \cdot t_\Psi)$
Sampled PLCP [16]	$\frac{n}{q} \log n$	$O(q)$	$O((d + q) \cdot t_\Psi)$

suffixes are compared to determine the LCP value. This increases the time complexity, while reducing the size significantly. In this paper, we take the other direction by reducing the access time, while achieving similar compression as in the run-length encoded PLCP variants [11,18].

4 Building the PLCP Array from a CSA

In this section, we adapt the *irreducible LCP algorithm* [14] to compute the PLCP array directly from a CSA.

Definition 1. *For $i > 1$, the* left match *of suffix $T[\mathsf{SA}[i], n]$ is $T[\mathsf{SA}[i-1], n]$.*

Definition 2. *Let $T[j, n]$ be the left match of $T[i, n]$. $\mathsf{PLCP}[i]$ is* reducible*, if $i, j > 1$ and $T[i-1] = T[j-1]$. If $\mathsf{PLCP}[i]$ is not reducible, then it is* irreducible*.*

The following lemma shows why reducible LCP values are called reducible.

Lemma 2 ([14]). *If $\mathsf{PLCP}[i]$ is reducible, then $\mathsf{PLCP}[i] = \mathsf{PLCP}[i-1] - 1$.*

The irreducible LCP algorithm works as follows: 1) find the irreducible PLCP values; 2) compute them naively; and 3) fill in the reducible values by using Lemma 2. As the sum of the irreducible values is at most $2n \log n$, the algorithm works in $O(n \log n)$ time [14].

The original algorithm uses the text and its suffix array that are expensive to access in a CSA. In the following lemma, we show how to find the irreducible values by using the function Ψ instead.

Lemma 3. *Let $T[j, n]$ be the left match of $T[i, n]$. The value $\mathsf{PLCP}[i+1]$ is reducible if and only if $T[i] = T[j]$ and $\Psi(\mathsf{SA}^{-1}[j]) = \Psi(\mathsf{SA}^{-1}[i]) - 1$.*

Proof. Let $x = \mathsf{SA}^{-1}[i]$. Then $x - 1 = \mathsf{SA}^{-1}[j]$.

"If." Assume that $T[i] = T[j]$ and $\Psi(x-1) = \Psi(x) - 1$. Then the left match of $T[\mathsf{SA}[\Psi(x)], n] = T[i+1, n]$ is $T[\mathsf{SA}[\Psi(x-1)], n] = T[j+1, n]$. As $i + 1 > 1$ and $j + 1 > 1$, it follows that $\mathsf{PLCP}[i+1]$ is reducible.

```
— Compute the PLCP array
1  PLCP[1] ← 0
2  (i, x) ← (1, SA^{-1}[1])
3  while i < n
4      Ψ_c ← rangeContaining(x)
5      if x − 1 ∉ Ψ_c or Ψ(x − 1) ≠ Ψ(x) − 1
6          PLCP[i + 1] ← lcp(Ψ(x))
7      else PLCP[i + 1] ← PLCP[i] − 1
8      (i, x) ← (i + 1, Ψ(x))

— Compute an LCP value
9  def lcp(b)
10     (a, k) ← (b − 1, 0)
11     Ψ_c ← rangeContaining(b)
12     while a ∈ Ψ_c
13         (a, b, k) ← (Ψ(a), Ψ(b), k + 1)
14         Ψ_c ← rangeContaining(b)
15     return k
```

Fig. 1. The irreducible LCP algorithm for using a CSA to compute the PLCP array. Function rangeContaining(x) returns $\Psi_c = [C[c] + 1, C[c + 1]]$ where $x \in \Psi_c$.

"Only if." Assume that $\mathsf{PLCP}[i + 1]$ is reducible, and let $T[k, n]$ be the left match of $T[i + 1, n]$. Then $k > 1$ and $T[k - 1] = T[i]$. As $T[k - 1, n]$ and $T[i, n]$ begin with the same character, and $T[k, n]$ is the left match of $T[i + 1, n]$, there cannot be any suffix S such that $T[k - 1, n] < S < T[i, n]$. But now $j = k - 1$, and hence $T[i] = T[j]$. Additionally,

$$\Psi(\mathsf{SA}^{-1}[j]) = \Psi(\mathsf{SA}^{-1}[k - 1]) = \mathsf{SA}^{-1}[k] = \mathsf{SA}^{-1}[i + 1] - 1 = \Psi(\mathsf{SA}^{-1}[i]) - 1.$$

The lemma follows. □

The algorithm is given in Fig. 1. We maintain invariant $x = \mathsf{SA}^{-1}[i]$, and scan through the CSA in text order. If the conditions of Lemma 3 do not hold for $T[i, n]$, then $\mathsf{PLCP}[i + 1]$ is irreducible, and we have to compute it. Otherwise we reduce $\mathsf{PLCP}[i + 1]$ to $\mathsf{PLCP}[i]$. To compute an irreducible value, we iterate $(\Psi^k(b - 1), \Psi^k(b))$ for $k = 0, 1, 2, \ldots$, until $T[\Psi^k(b - 1)] \neq T[\Psi^k(b)]$. When this happens, we return k as the requested LCP value. As we compute $\Psi(\cdot)$ for a total of $O(n \log n)$ times, we get the following theorem.

Theorem 1. *Given a compressed suffix array for a text of length n, the irreducible LCP algorithm computes the PLCP array in $O(n \log n \cdot t_\Psi)$ time, where t_Ψ is the time required for accessing Ψ. The algorithm requires $O(\log n)$ bits of working space in addition to the CSA and the PLCP array.*

We can use the algorithm to build any PLCP representation from Table 1 directly. The time bound is asymptotically tight, as shown in the following lemma.

Lemma 4 (Direct extension of Lemma 5 in [14]). *For an order-k* de Bruijn sequence *on an alphabet of size σ, the sum of all irreducible PLCP values is $n(1 - 1/\sigma) \log_\sigma n - O(n)$.*

The sum of irreducible PLCP values of a random sequence should also be close to $n(1 - 1/\sigma) \log_\sigma n$. The probability that the characters preceding a suffix and its left match differ, making the PLCP value irreducible, is $(1 - 1/\sigma)$. On the other hand, the average irreducible value should be close to $\log_\sigma n$ [6]. For a text

generated by an order-k Markov source with H bits of entropy, the estimate becomes $n(1 - 1/\sigma')(\log n)/H$. Here σ' is the effective alphabet size, defined by the probability $1/\sigma'$ that two characters sharing an order-k context are identical.

The following proposition shows that large-scale repetitiveness reduces the sum of the irreducible values, and hence improves the algorithm performance.

Proposition 1. *For a concatenation of r copies of text $T[1, n]$, the sum of irreducible PLCP values is $s + (r - 1)n$, where s is the sum of the irreducible PLCP values of T.*

Proof. Let $\mathcal{T} = T_1 T_2 \cdots T_r$ be the concatenation, $\mathcal{T}_{a,i}$ the suffix starting at $T_a[i]$, and $\mathsf{PLCP}_a[i]$ the corresponding PLCP value. Assume that $T_r[n]$ is lexicographically greater than the other end markers, but otherwise identical to them.

For every i, the suffix array of $\mathcal{T}$ contains a range with values $\mathcal{T}_{1,i}, \mathcal{T}_{2,i}, \ldots, \mathcal{T}_{r,i}$ [18]. Hence for any $a > 1$ and any i, the left match of $\mathcal{T}_{a,i}$ is $\mathcal{T}_{a-1,i}$, making the PLCP values reducible for almost all of the suffixes of T_2 to T_r. The exception is that $\mathcal{T}_{2,1}$ is irreducible, as its left match is $\mathcal{T}_{1,1}$, and hence $\mathsf{PLCP}_2[1] = (r - 1)n$.

Let $T[j, n]$ be the left match of $T[i, n]$ in the suffix array of T. Then the left match of $\mathcal{T}_{1,i}$ is $\mathcal{T}_{r,j}$, and $\mathsf{PLCP}_1[i] = \mathsf{PLCP}[i]$. Hence the sum of the irreducible values corresponding to the suffixes of T_1 is s. □

5 Sampled LCP Array

By Lemmas 1 and 2, the local maxima in the PLCP array are among the irreducible values, and the local minima are immediately before them.

Definition 3. *The value* $\mathsf{PLCP}[i]$ *is* maximal*, if it is irreducible, and* minimal*, if either $i = n$ or $\mathsf{PLCP}[i + 1]$ is maximal.*

Lemma 5. *If* $\mathsf{PLCP}[i]$ *is non-minimal, then* $\mathsf{PLCP}[i] = \mathsf{PLCP}[i + 1] + 1$.

Proof. If $\mathsf{PLCP}[i]$ is non-minimal, then $\mathsf{PLCP}[i+1]$ is reducible. The result follows from Lemma 2. □

In the following, R is the number of equal letter runs in BWT.

Lemma 6. *The number of minimal PLCP values is R.*

Proof. Lemma 3 essentially states that $\mathsf{PLCP}[i + 1]$ is reducible, if and only if $L[\Psi(\mathsf{SA}^{-1}[i])] = T[i] = T[j] = L[\Psi(\mathsf{SA}^{-1}[j])] = L[\Psi(\mathsf{SA}^{-1}[i]) - 1]$, where $T[j, n]$ is the left match of $T[i, n]$. As this is true for $n - R$ positions i, there are exactly R irreducible values. As every maximal PLCP value can be reduced to the next minimal value, and vice versa, the lemma follows. □

Lemma 7. *The sum of minimal PLCP values is $S - (n - R)$, where S is the sum of maximal values.*

Proof. From Lemmas 5 and 6. □

If we store the minimal PLCP values in SA order, and mark their positions in a bit vector, we can use them in a similar way as the SA samples. If we need $\mathsf{LCP}[i]$, and $\mathsf{LCP}[\Psi^k(i)]$ is a sampled position for the smallest $k \geq 0$, then $\mathsf{LCP}[i] = \mathsf{LCP}[\Psi^k(i)] + k$. As k can be $\Theta(n)$ in the worst case, the time bound is $O(n \cdot t_\Psi)$.

To improve the performance, we sample one out of $d' = n/R^{1-\varepsilon}$ consecutive non-minimal values for some $\varepsilon > 0$. Then there are R minimal samples and at most $R^{1-\varepsilon}$ extra samples. We mark the sampled positions in a bit vector of Raman et al. [20], taking at most $(1+o(1)) \cdot R \log \frac{n}{R} + O(R) + o(n)$ bits of space. Checking whether an LCP entry has been sampled takes $O(1)$ time.

We use δ codes [5] to encode the actual samples. As the sum of the minimal values is at most $2n \log n$, these samples take at most

$$R \log \frac{2n \log n}{R} + O\left(R \log \log \frac{n}{R}\right) \leq R \log \frac{n}{R} + O(R \log \log n)$$

bits of space. The extra samples require at most $\log n + O(\log \log n)$ bits each. To provide fast access to the samples, we can use dense sampling [9] or directly addressable codes [2]. This increases the size by a factor of $1 + o(1)$, making the total for samples $(1 + o(1)) \cdot R \log \frac{n}{R} + O(R \log \log n) + o(R \log n)$ bits of space.

We find a sampled position in at most $n/R^{1-\varepsilon}$ steps. By combining the size bounds, we get the following theorem.

Theorem 2. *Given a text of length n and a parameter $0 < \varepsilon < 1$, the sampled LCP array requires at most $(2+o(1)) \cdot R \log \frac{n}{R} + O(R \log \log n) + o(R \log n) + o(n)$ bits of space, where R is the number of equal letter runs in the BWT of the text. When used with a compressed suffix array, retrieving an LCP value takes at most $O((n/R^{1-\varepsilon}) \cdot t_\Psi)$ time, where t_Ψ is the time required for accessing Ψ.*

By using the BSD representation [13] for the bit vector, we can remove the $o(n)$ term from the size bound with a slight loss of performance.

When the space is limited, we can afford to sample the LCP array denser than the SA, as SA samples are larger than LCP samples. In addition to the mark in the bit vector, an SA sample requires $2 \log \frac{n}{d}$ bits of space, while an LCP sample takes just $\log v + O(\log \log v)$ bits, where v is the sampled value.

The LCP array can be sampled by a two-pass version of the irreducible LCP algorithm. On the first pass, we scan the CSA in suffix array order to find the minimal samples. Position x is minimal, if x is the smallest value in the corresponding Ψ_c, or if $\Psi(x-1) \neq \Psi(x)-1$. As we compress the samples immediately, we only need $O(\log n)$ bits of working space. On the second pass, we scan the CSA in text order, and store the extra samples in an array. Then we sort the array to SA order, and merge it with the minimal samples. As the number of extra samples is $o(R)$, we need $o(R \log n)$ bits of working space.

Theorem 3. *Given a compressed suffix array for a text of length n, the modified irreducible LCP algorithm computes the sampled LCP array in $O(n \log n \cdot t_\Psi)$ time, where t_Ψ is the time required for accessing Ψ. The algorithm requires $o(R \log n)$ bits of working space in addition to the CSA and the samples, where R is the number of equal letter runs in the BWT of the text.*

6 Implementation and Experiments

We have implemented the sampled LCP array, a run-length encoded PLCP array, and their construction algorithms as a part of the RLCSA [23].[3] For PLCP, we used the same run-length encoded bit vector as in the RLCSA. For the sampled LCP, we used a gap encoded bit vector to mark the sampled positions, and a stripped-down version of the same vector for storing the samples.

To avoid redundant work, we compute minimal instead of maximal PLCP values, and interleave their computation with the main loop. To save space, we only use *strictly minimal* PLCP values with $\mathsf{PLCP}[i] < \mathsf{PLCP}[i+1] + 1$ as the minimal samples. When sampling the LCP array, we make both of the passes in text order, and store all the samples in an array before compressing them.

For testing, we used a 2.66 GHz Intel Core 2 Duo E6750 system with 4 GB of memory (3.2 GB visible to OS) running a Fedora-based Linux with kernel 2.6.27. The implementation was written in C++, and compiled on g++ version 4.1.2. We used four data sets: human DNA sequences (*dna*) and English language texts (*english*) from the Pizza & Chili Corpus [7], the Finnish language Wikipedia with version history (*fiwiki*) [23], and the genomes of 36 strains of Saccharomyces paradoxus (*yeast*) [18].[4] When the data set was much larger than 400 megabytes, a 400 MB prefix was used instead. Further information on the data sets can be found in Table 2.

Only on the *dna* data set, the sum of the minimal values was close to the entropy-based estimate. On the highly repetitive *fiwiki* and *yeast* data sets, the difference between the estimate and the measurement was very large, as predicted by Proposition 1. Even regular English language texts contained enough large-scale repetitiveness that the sum of the minimal values could not be adequately explained by the entropy of the texts. This suggests that, for many real-world texts, the number of runs in BWT is a better compressibility measure than the empirical entropy.

The sum of minimal PLCP values was a good estimate for PLCP construction time. LCP sampling was somewhat slower because of the second pass. Both algorithms performed reasonably well on the highly repetitive data sets, but were much slower on the regular ones. The overall performance was roughly an order of magnitude worse than for the algorithms using plain text and SA [14].

We measured the performance of the sampled LCP array and the run-length encoded PLCP array on each of the data sets. We also measured the *locate* performance of the RLCSA to get a lower bound for the time and space of any PLCP-based approach. The results can be seen in Fig. 2.

The sampled LCP array outperformed PLCP on *english* and *dna*, where most of the queries were resolved through minimal samples. On *fiwiki* and *yeast*, the situation was reversed. As many extra samples were required to get reasonable

[3] The implementation is available at `http://www.cs.helsinki.fi/group/suds/rlcsa/`

[4] The yeast genomes were obtained from the Durbin Research Group at the Sanger Institute, `http://www.sanger.ac.uk/Teams/Team71/durbin/sgrp/`

Table 2. Properties of the data sets. H_5 is the order-5 empirical entropy, σ' the corresponding effective alphabet size, # the number of (strictly) minimal values, and S the sum of those values. $S' = n(1 - 1/\sigma')(\log n)/H_5 - n/\sigma'$ is an entropy-based estimate for the sum of the minimal values. The construction times are in seconds.

		Estimates			Minimal values			Strictly minimal		
Name	**MB**	$\mathbf{H_5}$	σ'	$\mathbf{S'/10^6}$	$\mathbf{\#/10^6}$	$\mathbf{S/10^6}$	**S/n**	$\mathbf{\#/10^6}$	$\mathbf{S/10^6}$	**S/n**
english	400	1.86	2.09	3167	156.35	1736	4.14	99.26	1052	2.51
fiwiki	400	1.09	1.52	3490	1.79	273	0.65	1.17	117	0.28
dna	385	1.90	3.55	4252	243.49	3469	8.59	158.55	2215	5.48
yeast	409	1.87	3.34	4493	15.64	520	1.21	10.05	299	0.70

	Sample rates		PLCP		Sampled LCP	
Name	**SA**	**LCP**	**Time**	**MB/s**	**Time**	**MB/s**
english	8, 16, 32, 64	8, 16	1688	0.24	2104	0.19
fiwiki	64, 128, 256, 512	32, 64, 128	327	1.22	533	0.75
dna	8, 16, 32, 64	8, 16	3475	0.11	3947	0.10
yeast	32, 64, 128, 256	16, 32, 64	576	0.71	890	0.46

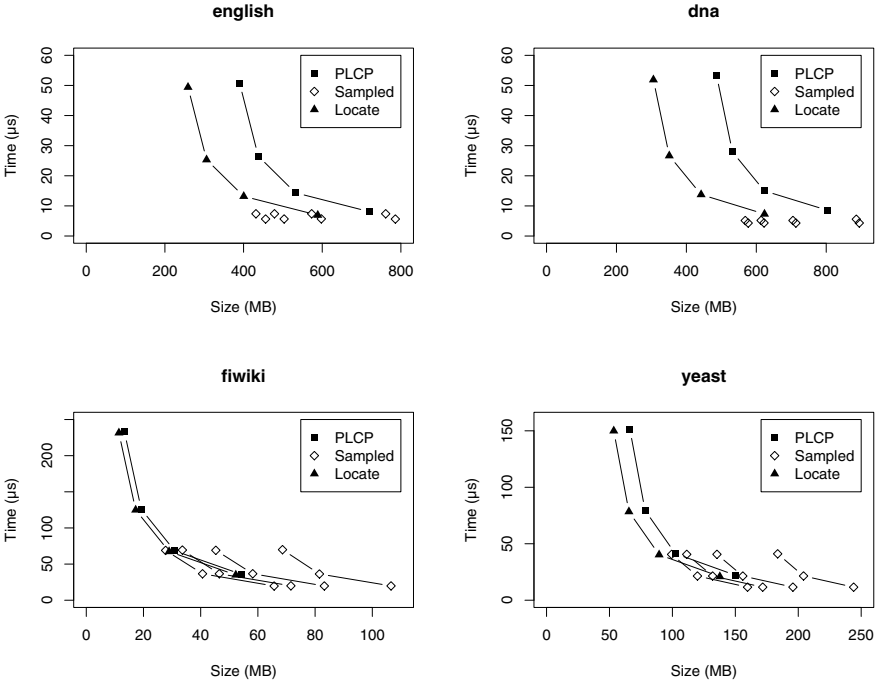


Fig. 2. Time/space trade-offs for retrieving an LCP or SA value. The times are averages over 10^6 random queries. Sampled LCP results are grouped by SA sample rate.

performance, increasing the size significantly, the sampled LCP array had worse time/space trade-offs than the PLCP array.

While we used RLCSA in the experiments, the results generalize to other types of CSA as well. The reason for this is that, in both PLCP and sampled LCP, the time required for retrieving an LCP value depends mostly on the number of iterations of Ψ required to find a sampled position.

7 Discussion

We have described the sampled LCP array, and shown that it offers better time/space trade-offs than the PLCP-based alternatives, when the number of extra samples required for dense sampling is small. Based on the experiments, it seems that one should use the sampled LCP array for regular texts, and a PLCP-based representation for highly repetitive texts.

In a recent proposal [4], the entire LCP array was compressed by using directly addressable codes (DAC) [2]. The resulting structure is much faster but usually also much larger than the other compressed LCP representations. See the full paper [24] for a comparison between the sampled LCP array and the DAC-based approach.

We have also shown that it is feasible to construct the (P)LCP array directly from a CSA. While the earlier algorithms are much faster, it is now possible to construct the (P)LCP array for larger texts than before, and the performance is still comparable to that of direct CSA construction [23]. On a multi-core system, it is also easy to get extra speed by parallelizing the construction.

It is possible to maintain the (P)LCP array when merging two CSAs. The important observation is that an LCP value can only change, if the left match changes in the merge. An open question is, how much faster the merging is, both in the worst case and in practice, than rebuilding the (P)LCP array.

While the suffix array and the LCP array can be compressed to a space relative to the number of of equal letter runs in BWT, no such representation is known for suffix tree topology. This is the main remaining obstacle in the way to compressed suffix trees optimized for highly repetitive texts.

References

1. Abouelhoda, M.I., Kurtz, S., Ohlebusch, E.: Replacing suffix trees with enhanced suffix arrays. Journal on Discrete Algorithms 2(1), 53–86 (2004)
2. Brisaboa, N.R., Ladra, S., Navarro, G.: Directly addressable variable-length codes. In: Hyyro, H. (ed.) SPIRE 2009. LNCS, vol. 5721, pp. 122–130. Springer, Heidelberg (2009)
3. Burrows, M., Wheeler, D.: A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation (1994)
4. Cánovas, R., Navarro, G.: Practical compressed suffix trees. In: Festa, P. (ed.) SEA 2010. LNCS, vol. 6049, pp. 94–105. Springer, Heidelberg (2010)
5. Elias, P.: Universal codeword sets and representations of the integers. IEEE Transactions on Information Theory 21(2), 194–203 (1975)

6. Fayolle, J., Ward, M.D.: Analysis of the average depth in a suffix tree under a Markov model. In: Proc. 2005 International Conference on Analysis of Algorithms, DMTCS Proceedings, vol. AD, pp. 95–104. DMTCS (2005)
7. Ferragina, P., González, R., Navarro, G., Venturini, R.: Compressed text indexes: From theory to practice. Journal of Experimental Algorithms 13, 1.12 (2009)
8. Ferragina, P., Manzini, G.: Indexing compressed text. Journal of the ACM 52(4), 552–581 (2005)
9. Ferragina, P., Venturini, R.: A simple storage scheme for strings achieving entropy bounds. In: SODA 2007, pp. 690–696. SIAM, Philadelphia (2007)
10. Fischer, J.: Wee LCP. arXiv:0910.3123v1 [cs.DS] (2009)
11. Fischer, J., Mäkinen, V., Navarro, G.: Faster entropy-bounded compressed suffix trees. Theoretical Computer Science 410(51), 5354–5364 (2009)
12. 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)
13. Gupta, A., Hon, W.-K., Shah, R., Vitter, J.S.: Compressed data structures: dictionaries and data-aware measures. Theoretical Computer Science 387(3), 313–331 (2007)
14. Kärkkäinen, J., Manzini, G., Puglisi, S.: Permuted longest-common-prefix array. In: Kucherov, G., Ukkonen, E. (eds.) CPM 2009. LNCS, vol. 5577, pp. 181–192. Springer, Heidelberg (2009)
15. 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)
16. Khmelev, D.: Program lcp version 0.1.9 (2004), `http://www.math.toronto.edu/dkhmelev/PROGS/misc/lcp-eng.html`
17. Kurtz, S.: Reducing the space requirement of suffix trees. Software: Practice and Experience 29(13), 1149–1171 (1999)
18. Mäkinen, V., Navarro, G., Sirén, J., Välimäki, N.: Storage and retrieval of individual genomes. In: Batzoglou, S. (ed.) RECOMB 2009. LNCS, vol. 5541, pp. 121–137. Springer, Heidelberg (2009)
19. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Computing Surveys 39(1), 2 (2007)
20. Raman, R., Raman, V., Rao, S.S.: Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In: SODA 2002, pp. 233–242. SIAM, Philadelphia (2002)
21. Russo, L., Navarro, G., Oliveira, A.: Fully-compressed suffix trees. In: Laber, E.S., Bornstein, C., Nogueira, L.T., Faria, L. (eds.) LATIN 2008. LNCS, vol. 4957, pp. 362–373. Springer, Heidelberg (2008)
22. Sadakane, K.: Compressed suffix trees with full functionality. Theory of Computing Systems 41(4), 589–607 (2007)
23. Sirén, J.: Compressed suffix arrays for massive data. In: Hyyro, H. (ed.) SPIRE 2009. LNCS, vol. 5721, pp. 63–74. Springer, Heidelberg (2009)
24. Sirén, J.: Sampled longest common prefix array. arXiv:1001.2101v2 [cs.DS] (2010)

Verifying a Parameterized Border Array in $O(n^{1.5})$ Time

Tomohiro I[1], Shunsuke Inenaga[2], Hideo Bannai[1], and Masayuki Takeda[1]

[1] Department of Informatics, Kyushu University
[2] Graduate School of Information Science and Electrical Engineering, Kyushu University
744 Motooka, Nishiku, Fukuoka, 819–0395 Japan
tomohiro.i@i.kyushu-u.ac.jp,
inenaga@c.csce.kyushu-u.ac.jp,
{bannai,takeda}@inf.kyushu-u.ac.jp

Abstract. The parameterized pattern matching problem is to check if there exists a renaming bijection on the alphabet with which a given pattern can be transformed into a substring of a given text. A *parameterized border array* (*p-border array*) is a parameterized version of a standard border array, and we can efficiently solve the parameterized pattern matching problem using p-border arrays. In this paper we present an $O(n^{1.5})$-time $O(n)$-space algorithm to verify if a given integer array of length n is a valid p-border array for an unbounded alphabet. The best previously known solution takes time proportional to the n-th Bell number $\frac{1}{e}\sum_{k=0}^{\infty}\frac{k^n}{k!}$, and hence our algorithm is quite efficient.

1 Introduction

The *parameterized matching* (*p-matching*) problem [1] is a kind of string matching problem, where a pattern is considered to occur in a text when there exists a renaming bijection on the alphabet with which the pattern can be transformed into a substring of the text. Parameterized matching has applications in e.g. software maintenance, plagiarism detection, and RNA structural matching, thus it has extensively been studied (e.g., see [2,3,4,5,6]).

In this paper we focus on *parameterized border arrays* (*p-border arrays*) [7], which are a parameterized version of border arrays [8]. Let Π be the alphabet. The p-border array of a given pattern p of length m can be computed in $O(m \log |\Pi|)$ time, and the p-matching problem can be solved in $O(n \log |\Pi|)$ time for any text p-string of length n, using the p-border array [7].

This paper deals with the *reverse engineering problem on p-border arrays*, namely, the problem of verifying if a given integer array of length n is a p-border array of some string. We propose an $O(n^{1.5})$-time $O(n)$-space algorithm to solve this problem for an unbounded alphabet. We emphasize that the best previously known solution to this problem takes time proportional to the n-th Bell number $\frac{1}{e}\sum_{k=0}^{\infty}\frac{k^n}{k!}$, and hence our algorithm is quite efficient.

A. Amir and L. Parida (Eds.): CPM 2010, LNCS 6129, pp. 238–250, 2010.

Related Work. There exists a linear time algorithm to solve the reverse problem on p-border arrays for a binary alphabet [9]. An $O(p_n)$-time algorithm to enumerate all p-border arrays of length up to n on a binary alphabet was also presented in [9], where p_n denotes the number of p-border arrays of length at most n for a binary alphabet.

In [10], a linear time algorithm to verify if a given integer array is the (standard) border array [8] of some string was presented. Their algorithm works for both bounded and unbounded alphabets. A simpler linear-time solution for the same problem for a bounded alphabet was shown in [11]. An algorithm to enumerate all border arrays of length at most n in $O(b_n)$-time was given in [10], where b_n is the number of border arrays of length at most n.

The reverse engineering problems, as well as the enumeration problems for other string data structures (suffix arrays, DAWG, etc.) have been extensively studied [12,13,14,15,16,17,18], whose solutions give us further insight concerning the data structures.

2 Preliminaries

Let Σ and Π be two disjoint finite alphabets. An element of $(\Sigma \cup \Pi)^*$ is called a *p-string*. The length of any p-string s is the total number of constant and parameter symbols in s and is denoted by $|s|$. The string of length 0 is called the empty string and is denoted by ε. For any p-string s of length n, the i-th symbol is denoted by $s[i]$ for each $1 \leq i \leq n$, and the *substring* starting at position i and ending at position j is denoted by $s[i:j]$ for $1 \leq i \leq j \leq n$.

Any two p-strings $s, t \in (\Sigma \cup \Pi)^*$ of length m are said to *parameterized match* (*p-match*) if s can be transformed into t by a renaming function f from the symbols of s to the symbols of t, where f is the identify on Σ. The p-matching problem on $\Sigma \cup \Pi$ is reducible in linear time to the p-matching problem on Π [2]. Thus we will only consider p-strings over Π.

Let $\mathcal{N}$ be the set of non-negative integers. Let $pv : \Pi^* \rightarrow \mathcal{N}^*$ be the function s.t. for any p-string s of length $n > 0$, $pv(s) = u$ where, for $1 \leq i \leq n$, $u[i] = 0$ if $s[i] \neq s[j]$ for any $1 \leq j < i$, and $u[i] = i - k$ if $k = \max\{j \mid s[i] = s[j], 1 \leq j < i\}$. Let $pv(\varepsilon) = \varepsilon$. Two p-strings s and t of the same length m p-match iff $pv(s) = pv(t)$. For any $p \in \mathcal{N}^*$, let $zeros(p)$ denotes the number of 0's in p, that is, $zeros(p) = |\{i \mid p[i] = 0, 1 \leq i \leq |p|\}|$. For any $s \in \Pi$, $zeros(pv(s))$ equals the number of different characters in s. For example, `aabb` and `bbaa` p-match since $pv(\texttt{aabb}) = pv(\texttt{bbaa}) = 0\ 1\ 0\ 1$. Note $zeros(pv(\texttt{aabb})) = zeros(pv(\texttt{bbaa})) = 2$.

A *parameterized border* (*p-border*) of a p-string s of length n is any integer j s.t. $0 \leq j < n$ and $pv(s[1:j]) = pv(s[n-j+1:n])$. For example, the set of p-borders of p-string `aabb` is $\{2, 1, 0\}$ since $pv(\texttt{aa}) = pv(\texttt{bb}) = 0\ 1$, $pv(\texttt{a}) = pv(\texttt{b}) = 0$, and $pv(\varepsilon) = pv(\varepsilon) = \varepsilon$. We also say that b is a p-border of $p \in \mathcal{N}^*$ if b is a p-border of some p-string $s \in \Pi^*$ and $p = pv(s)$. The *parameterized border array* (*p-border array*) β_s of a p-string s of length n is an array of length n such that $\beta_s[i] = j$, where j is the longest p-border of $s[1:i]$. For example, for p-string $s = \texttt{aabbaa}$, $\beta_s = [0, 1, 1, 2, 3, 4]$. When it is

clear from the context, we abbreviate β_s as β. Let $\mathrm{P} = \{pv(s) \mid s \in \Pi^*\}$ and $\mathrm{P}_\beta = \{p \in \mathrm{P} \mid \beta[i]$ is the longest p-border of $p[1:i], 1 \le i \le |\beta|\}$.

For any $i, j \in \mathcal{N}$, let $cut(i,j) = 0$ if $i \ge j$, and $cut(i,j) = i$ otherwise. For any $p \in \mathrm{P}$ and $1 \le j \le |p|$, let $suf(p,j) = cut(p[|p|-j+1],1)cut(p[|p|-j+2],2)\cdots cut(p[|p|],j)$. Let $suf(p,0) = \varepsilon$. For example, if $p[1:10] =$ 0 0 2 0 3 1 3 2 6 3,

$$\begin{aligned} suf(p,5) &= cut(p[6],1)cut(p[7],2)cut(p[8],3)cut(p[9],4)cut(p[10],5) \\ &= cut(1,1)cut(3,2)cut(2,3)cut(6,4)cut(3,5) = 0\ 0\ 2\ 0\ 3. \end{aligned}$$

Then, for any p-string $s \in \Pi^*$ and $1 \le j \le |s|$, $suf(pv(s),j) = pv(s[|s|-j+1:|s|])$. Hence, j is a p-border of $pv(s)$ iff $suf(pv(s),j) = pv(s)[1:j]$ for some $1 \le j < |s|$.

This paper deals with the following problem.

Problem 1 (Verifying a valid p-border array). Given an integer array y of length n, determine if there exists a p-string s such that $\beta_s = y$.

To solve Problem 1, we can use the algorithm of Moore et al. [19] to generate all strings in $\mathrm{P}^n = \{p \mid p \in \mathrm{P}, |p| = n\}$ in $O(|\mathrm{P}^n|)$ time, and then we check if $p \in \mathrm{P}_y$ for each generated $p \in \mathrm{P}^n$. Still, it is known that $|\mathrm{P}^n|$ is equal to the n-th Bell number $\frac{1}{e}\sum_{k=0}^{\infty}\frac{k^n}{k!}$.

As a much more efficient solution, we present our $O(n^{1.5})$-time algorithm in the sequel.

3 Properties on Parameterized Border Arrays

Here we introduce important properties of p-border arrays that are useful to solve Problem 1.

For any integer array ℓ, let $|\ell|$ denote the length of the integer array ℓ. Let $\ell[i:j]$ denote a subarray of ℓ for any $1 \le i \le j \le |\ell|$. Let $\Gamma = \{\gamma \mid \gamma[1] = 0, 1 \le \gamma[i] \le \gamma[i-1]+1, 1 < i \le |\gamma|\}$. For any $\gamma \in \Gamma$ and any $i \ge 1$, let $\gamma^k[i] = \gamma[i]$ if $k = 1$, and $\gamma[\gamma^{k-1}[i]]$ if $k > 1$ and $\gamma^{k-1}[i] \ge 1$. By the definition of Γ, the sequence $i, \gamma^1[i], \gamma^2[i], \ldots$ is monotonically decreasing and terminates with $1, 0$. Let $\mathrm{A} = \{\alpha \mid \alpha \in \Gamma, \alpha[i] \in \{\alpha^1[i-1]+1, \alpha^2[i-1]+1, \ldots, 1\}, 1 < i \le |\alpha|\}$. It is clear that $\mathrm{A} \subset \Gamma$. Let B denote the set of all p-border arrays.

Lemma 1. $\mathrm{B} \subseteq \Gamma$.

Proof. By definition, it is clear that $\beta[1] = 0$ and $1 \le \beta[i]$ for any $1 < i \le |\beta|$. For any $p \in \mathrm{P}_\beta$ and i, since $suf(p[1:i], \beta[i]) = p[1:\beta[i]]$, $suf(p[1:i-1], \beta[i]-1) = p[1:\beta[i]-1]$. Thus $\beta[i-1] \ge \beta[i]-1$, and therefore $\beta[i] \le \beta[i-1]+1$. □

Lemma 2. *For any $\beta \in \mathrm{B}$, $p \in \mathrm{P}_\beta$, and $1 \le i \le |p|$, $\{\beta^1[i], \beta^2[i], \ldots, 0\}$ is the set of p-borders of $p[1:i]$.*

Lemma 3. *For any $\beta \in \mathrm{B}$, $p \in \mathrm{P}_\beta$, and $1 \le i \le |p|$, if $p[i] = 0$, then $p[b] = 0$ for any $b \in \{\beta^1[i], \beta^2[i], \ldots, 1\}$.*

Lemma 4. $\mathrm{B} \subseteq \mathrm{A}$.

Proof. For any $\beta \in \mathrm{B}, p \in \mathrm{P}_\beta$ and $1 < i \leq |p|$, since $suf(p[1:i], \beta[i]) = p[1:\beta[i]]$, $suf(p[1:i-1], \beta[i]-1) = p[1:\beta[i]-1]$. Since $\beta[i]-1$ is a p-border of $p[1:i-1]$, $\beta[i]-1 \in \{\beta^1[i-1], \beta^2[i-1], \ldots, 0\}$ by Lemma 2. Hence, $\beta[i] \in \{\beta^1[i-1]+1, \beta^2[i-1]+1, \ldots, 1\}$. □

Definition 1 (Conflict Points). *Let $\alpha \in \mathrm{A}$. For any c', c $(1 < c' < c \leq |\alpha|)$, if $\alpha[c'] = \alpha[c]$ and $c' - 1 = \alpha^k[c-1]$ with some k, then c' and c are said to be* in conflict *with each other. Such points are called* conflict points.

Let C_α be the set of conflict points in α and $C_\alpha(c)$ be the set of points that conflict with c $(1 \leq c \leq |\alpha|)$. For any $i \leq j \in \mathcal{N}$, let $[i,j] = \{i, i+1, \ldots, j\} \subset \mathcal{N}$. We denote $C_\alpha^{[i,j]} = C_\alpha \cap [i,j]$ and $C_\alpha^{[i,j]}(c) = C_\alpha(c) \cap [i,j]$ to restrict the elements of the sets within the range $[i,j]$.

By Definition 1, $C_\alpha^{[1,c]}(c) = \{c'\} \cup C_\alpha^{[1,c']}(c')$ where $c' = \max C_\alpha^{[1,c]}(c)$. Consider a tree such that $C_\alpha \cup \{\perp\}$ is the set of nodes where $\perp$ is the root, and $\{(c', c) \mid c \in C_\alpha, c' = \max C_\alpha^{[1,c]}(c)\} \cup \{(\perp, c) \mid c \in C_\alpha, C_\alpha^{[1,c]}(c) = \emptyset\}$ the set of edges. This tree is called the *conflict tree* of α and it represents the relations of conflict points of α. Let $CT_\alpha(c)$ denote the set of children of node c and $CT_\alpha^{[i,j]}(c) = CT_\alpha(c) \cap [i,j]$. We define $order_\alpha(c)$ to be the depth of node c and $maxc_\alpha(c) = \max\{order_\alpha(c') \mid c' \in \{c\} \cup C_\alpha(c)\}$.

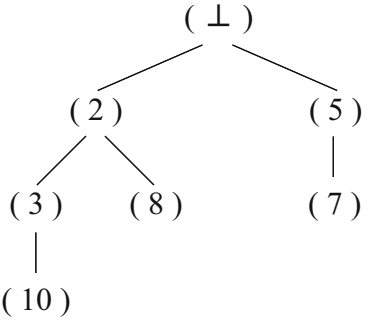


Fig. 1. The conflict tree of $\alpha = [0,1,1,2,3,4,3,1,2,1]$

Fig. 1 illustrates the conflict tree for $\alpha = [0,1,1,2,3,4,3,1,2,1]$. Here $C_\alpha = \{2,3,5,7,8,10\}$, $C_\alpha(3) = \{2,10\}$, $CT_\alpha(2) = \{3,8\}$, $order_\alpha(2) = order_\alpha(5) = 1$, $order_\alpha(3) = order_\alpha(7) = order_\alpha(8) = 2$, $order_\alpha(10) = 3$, $maxc_\alpha(5) = maxc_\alpha(7) = maxc_\alpha(8) = 2$, $maxc_\alpha(2) = maxc_\alpha(3) = maxc_\alpha(10) = 3$, and so on.

Lemma 5 will be used to show the $O(n^{1.5})$ time complexity of our algorithm of Section 4.

Lemma 5. *For any $\alpha[1:n] \in \mathrm{A}$, $n \geq 1 + \sum_{c \in C_\alpha} \lfloor 2^{order_\alpha(c)-2} \rfloor$.*

Proof. Let $c_t \in C_\alpha$ with $t \geq 2$, $C_\alpha^{[1:c_t]}(c_t) = \{c_1, c_2, \ldots, c_{t-1}\}$ with $c_1 < c_2 < \cdots < c_t$. Let $m = \alpha[c_1] = \alpha[c_2] = \cdots = \alpha[c_t]$. By the definition of Γ, for any $1 < i \leq n$, $\alpha[i] \leq \alpha[i-1]+1$. Then, it follows from $(c_t - 1) - c_{t-1} \geq \alpha[c_t - 1] - \alpha[c_{t-1}]$ that $m + (c_t - 1) - c_{t-1} \geq \alpha[c_t - 1]$. Consequently, by Definition 1, we have $c_t \geq 2c_{t-1} - m$ from $\alpha[c_t - 1] \geq c_{t-1} - 1$. Hence, $c_t \geq 2c_{t-1} - m \geq 2^2 c_{t-2} - m(1+2) \geq \cdots \geq 2^{t-1}c_1 - m\sum_{i=0}^{t-2} 2^i = 2^{t-1}c_1 - m(2^{t-1}-1) = 2^{t-1}(c_1 - m) + m \geq 2^{t-1} + m$. It leads to $\alpha[c_t] - (\alpha[c_t - 1] + 1) \leq m - c_{t-1} \leq -2^{t-2}$. Since $\alpha[i] = 0$ and

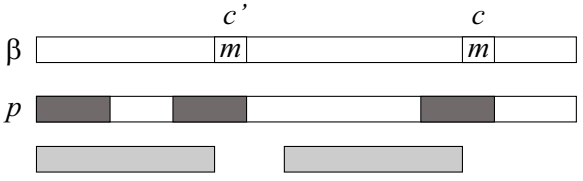


Fig. 2. Let $c, c' \in C_\beta$ and $\beta[c'] = \beta[c] = m$. Then, $c' \in C_\beta(c)$, $p[1:m] = suf(p[1:c'], m) = suf(p[1:c], m)$, and $p[1:c'-1] = suf(p[1:c-1], c'-1)$.

$1 \le \alpha[i] \le \alpha[i-1]+1$ for any $1 < i \le n$, $n-1$ should be greater than the value subtracted over all conflict points. Therefore, the statement holds. □

The relation between conflict points of $\beta \in \mathrm{B}$ and $p \in \mathrm{P}_\beta$ is illustrated in Fig. 2.

Lemma 6 shows a necessary-and-sufficient condition for $\beta[1:i]m$ to be a valid p-border array of some $p[1:i+1] \in \mathcal{N}^*$, when $\beta[1:i]$ is a valid p-border array.

Lemma 6. *Let $\beta[1:i] \in \mathrm{B}$, $m \in \mathcal{N}$, and $p[1:i+1] \in \mathcal{N}^*$. Then, $\beta[1:i]m \in \mathrm{B}$ and $p[1:i+1] \in \mathrm{P}_{\beta[1:i]m}$ if and only if*

$$p[1:i+1] \in \mathrm{P} \wedge p[1:i] \in \mathrm{P}_{\beta[1:i]} \wedge \exists k, \beta^k[i] = m-1 \wedge cut(p[i+1], m) = p[m]$$
$$\wedge \big(C_{\beta[1:i]m}(i+1) \neq \emptyset \Rightarrow \big(p[m] = 0 \wedge \forall c \in C_{\beta[1:i]m}(i+1), p[i+1] \neq p[c]$$
$$\wedge \big(\exists c' \in C_{\beta[1:i]m}(i+1), p[c'] = 0 \Rightarrow m \le p[i+1] < c'\big)\big)\big).$$

Lemma 7 shows a yet stronger result, a necessary-and-sufficient condition for $\beta[1:i]m$ to be a valid p-border array of length $i+1$, when $\beta[1:i]$ is a valid p-border array of length i.

Lemma 7. *Let $\beta[1:i] \in \mathrm{B}$ and $m \in \mathcal{N}$. Then, $\beta[1:i]m \in \mathrm{B}$ if and only if*

$$\exists k, \beta^k[i] = m-1$$
$$\wedge \big(C_{\beta[1:i]m}(i+1) \neq \emptyset \Rightarrow \big(\exists p[1:i] \in \mathrm{P}_{\beta[1:i]} \text{ s.t. } p[m] = 0$$
$$\wedge \big(\exists c' \in C_{\beta[1:i]m}(i+1), p[c'] = 0 \Rightarrow zeros(p[m:c'-1]) \ge |C_{\beta[1:i]m}(i+1)|\big)\big)\big).$$

Proofs of Lemmas 6 and 7 will be shown in a full version of this paper.

In the next section we design our algorithm to solve Problem 1 based on Lemmas 6 and 7.

4 Algorithm

This section presents our $O(n^{1.5})$-time $O(n)$-space algorithm to verify if a given integer array of length n is a valid p-border array for an unbounded alphabet.

4.1 Z-Pattern Representation

Lemma 7 implies that, in order to check if $\beta[1:i]m \in \mathrm{B}$, it suffices for us to know if $p[i]$ is zero or non-zero for each i. Let $\star$ be a special symbol s.t. $\star \neq 0$.

For any $p \in \mathrm{P}$ and $1 \leq i \leq |p|$, let $ptoz(p)[i] = 0$ if $p[i] = 0$, and $ptoz(p)[i] = \star$ otherwise. The sequence $ptoz(p) \in \{0, \star\}^*$ is called the *z-pattern* of p. For any $\beta \in \mathrm{B}$, let $\mathrm{Z}_\beta = \{ptoz(p) \mid p \in \mathrm{P}_\beta\}$.

The next lemma follows from Lemmas 3, 6, and 7.

Lemma 8. *Let $\beta \in \mathrm{B}$ and $z \in \{0, \star\}^*$. Then, $z \in \mathrm{Z}_\beta$ if and only if all of the following conditions hold for any $1 \leq i \leq |z|$:*

1. $i = 1 \Rightarrow z[i] = 0$.
2. $z[\beta[i]] = \star \Rightarrow z[i] = \star$.
3. $\exists c \in C_\beta, \exists k, i = \beta^k[c] \Rightarrow z[i] = 0$.
4. $\exists c \in C_\beta(i), z[c] = 0 \Rightarrow z[i] = \star$.
5. $i \in C_\beta \wedge zeros(z[\beta[i] : i-1]) < maxc_\beta(i) - 1 \Rightarrow z[i] = \star$.
6. $i \in C_\beta \wedge zeros(z[\beta[i] : i-1]) = order_\beta(i) - 1 \Rightarrow z[i] = 0$.

Let $E_\beta = \{i \mid \exists c \in C_\beta, \exists k, i = \beta^k[c]\}$. For any $z \in \mathrm{Z}_\beta$ and $i \in E_\beta$, $z[i]$ is always 0.

We check if a given integer array $y[1:n]$ is a valid p-border array in two steps.

Step 1: While scanning $y[1:n]$ from left to right, check whether $y[1:n] \in \mathrm{A}$ and whether each position i $(1 \leq i \leq n)$ of y satisfies Conditions 3 and 4 of Lemma 8. Also, we compute E_y, and $order_y(i)$ and $maxc_y(i)$ for each $i \in C_y$.

Step 2: For each $i = 1, 2, \ldots, n$, we determine the value of $z[i]$ so that the conditions of Lemma 8 hold.

If we can determine $z[i]$ for all $i = 1, 2, \ldots, n$ in Step 2, then the input array y is a p-border array of some $p \in \mathrm{P}$ such that $ptoz(p) = z$.

4.2 Pruning Techniques

Given an integer array y of length n, we inherently have to search $\{0, \star\}^n$ for a z-pattern $z \in \mathrm{Z}_y$. To achieve an efficient solution, we utilize the following pruning lemmas.

For any $\beta \in \mathrm{B}$ and $1 \leq i \leq |\beta|$, we write as $u[1:i] \in \mathrm{Z}^i_\beta$ if and only if $u[1:i] \in \{0, \star\}^*$ satisfies all the conditions of Lemma 8 for any j $(1 \leq j \leq i)$. For any $h > i$, let $z[h] = 0$ if $h \in E_\beta$, and leave it undefined otherwise. Clearly, for any $z \in \mathrm{Z}_\beta$ and $1 \leq i \leq |\beta|$, $z[1:i] \in Z^i_\beta$.

We can use the contraposition of the next lemma for pruning the search tree at each non-conflict point of y.

Lemma 9. *Let $\beta \in \mathrm{B}$ and $i \notin C_\beta$ $(2 \leq i \leq |\beta|)$. For any $u[1:i-1] \in \mathrm{Z}^{i-1}_\beta$, if $u[\beta[i]] = 0$ and there exists $z \in \mathrm{Z}_\beta$ s.t. $z[1:i] = u[1:i-1]\star$, then there exists $z' \in \mathrm{Z}_\beta$ s.t. $z'[1:i] = u[1:i-1]0$.*

Proof. For any $1 \leq j \leq |\beta|$, let $v[j] = 0$ if $j = i$, and $v[j] = z[j]$ otherwise. Now we show $v \in \mathrm{Z}_\beta$. $v[i]$ clearly holds all the conditions of Lemma 8. Since $v[j] = z[j]$ at any other points, $v[j]$ satisfies Conditions 1, 2, 3 and 4. Furthermore, for any $c \in C_\beta$, $v[c]$ holds Conditions 5 and 6, since $zeros(v[\beta[c] : c-1]) \geq zeros(z[\beta[c] : c-1])$ and $z[c]$ holds those conditions. □

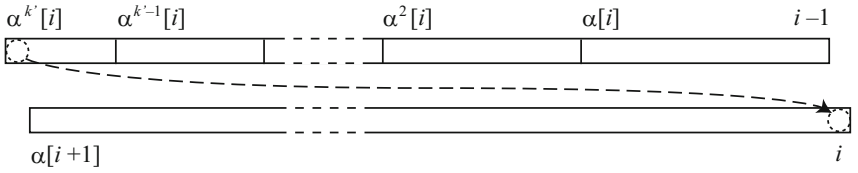


Fig. 3. Illustration for Lemma 11. If $\alpha^{k'}[i] = \alpha[i+1] - 1 \in F_\alpha(b)$, then $i \in F_\alpha(b)$.

Next, we discuss our pruning technique regarding conflict points of y. Let $\beta \in \mathrm{B}$. $c \in C_\beta$ is said to be an *active conflict point* of β, iff $E_\beta \cap (\{c\} \cup C_\beta(c)) = \emptyset$. Obviously, for any $z \in \mathrm{Z}_\beta$ and $c \in C_\beta$, $z[c] = 0$ if $E_\beta \cap \{c\} \neq \emptyset$ and $z[c] = \star$ if $E_\beta \cap C_\beta(c) \neq \emptyset$. Hence we never branch out at any inactive conflict point during the search for $z \in \mathrm{Z}_\beta$. Let AC_β be the set of active conflict points in β. Our pruning method for active conflict points is described in Lemma 10.

Lemma 10. *Let $\beta \in \mathrm{B}, i \in AC_\beta$ and $i \leq r \leq |\beta|$ with $|CT_\beta^{[1,r]}(i)| < 2$. For any $u[1:i-1] \in \mathrm{Z}_\beta^{i-1}$, if $u[1:i-1]0 \in \mathrm{Z}_\beta^i$ and there exists $z[1:r] \in \mathrm{Z}_\beta^r$ s.t. $z[1:i] = u[1:i-1]\star$, then there exists $z'[1:r] \in \mathrm{Z}_\beta^r$ s.t. $z'[1:i] = u[1:i-1]0$.*

In order to prove Lemma 10, particularly to ensure Conditions 5 and 6 of Lemma 8 hold, we will estimate the number of 0's within the range $[\beta[c], c-1]$ for each $c \in C_\beta$ that is obtained when the prefix of a z-pattern is $u[1:i-1]0$. Here, for any $\alpha \in \mathrm{A}$ and $1 \leq b \leq |\alpha|$, let $F_\alpha(b) = \{b\} \cup \{b' \mid \exists k, b = \alpha^k[b']\}$ and $F_\alpha^{[i,j]}(b) = F_\alpha(b) \cap [i,j]$. Then, the number of 0's related to i within the range $[\beta[c], c-1]$ can be estimated by $|F_\beta^{[\beta[c],c-1]}(i)|$. The following lemmas show some properties of $F_\alpha(b)$ that are useful to prove Lemma 10 above.

Lemma 11. *Let $\alpha \in \mathrm{A}$. For any $1 \leq b \leq |\alpha|$ and $1 < i < |\alpha|$,*

$$|F_\alpha^{[\alpha[i+1],i]}(b)| - |F_\alpha^{[\alpha[i],i-1]}(b)| - \sum_{k=1}^{k'-1} |F_\alpha^{[\alpha^{k+1}[i],\alpha^k[i]-1]}(b)| = \begin{cases} 1 & \text{if } i \in F_\alpha(b) \text{ and} \\ & \alpha^{k'}[i] \notin F_\alpha(b), \\ 0 & \text{otherwise,} \end{cases}$$

where k' is the integer such that $\alpha^{k'}[i] = \alpha[i+1] - 1$.

Proof. Since $[\alpha[i+1]-1, i-1] = [\alpha^{k'}[i], \alpha^{k'-1}[i]-1] \cup [\alpha^{k'-1}[i], \alpha^{k'-2}[i]-1] \cup \cdots \cup [\alpha^1[i], i-1]$, $|F_\alpha^{[\alpha[i+1]-1,i-1]}(b)| = |F_\alpha^{[\alpha[i],i-1]}(b)| + \sum_{k=1}^{k'-1} |F_\alpha^{[\alpha^{k+1}[i],\alpha^k[i]-1]}(b)|$ (See Fig. 3). Then, the key is whether each of i and $\alpha[i+1]-1$ is in $F_\alpha(b)$ or not. Obviously, if $\alpha^{k'}[i] = \alpha[i+1] - 1 \in F_\alpha(b)$, then $i \in F_\alpha(b)$. It leads to the statement. □

Lemma 11 implies that $|F_\alpha^{[\alpha[i],i-1]}(b)|$ is monotonically increasing for i.

Lemma 12. *Let $\alpha \in \mathrm{A}$ and $c', c \in C_\alpha$ with $c' \in C_\alpha^{[1,c]}(c)$. For any $1 \le b < c'$,*

$$|F_\alpha^{[m,c-1]}(b)| \ge |F_\alpha^{[\alpha[c-1],c-2]}(b)| + \sum_{k=1}^{k'-1} |F_\alpha^{[\alpha^{k+1}[c-1],\alpha^k[c-1]-1]}(b)| + 1,$$

where $m = \alpha[c'] = \alpha[c]$ and k' is the integer such that $\alpha^{k'}[c-1] = c'-1$.

Proof. In a similar way to the proof of Lemma 11, we have $|F_\alpha^{[m,c-2]}(b)| = |F_\alpha^{[\alpha[c-1],c-2]}(b)| + \sum_{k=1}^{k'-1} |F_\alpha^{[\alpha^{k+1}[c-1],\alpha^k[c-1]-1]}(b)| + |F_\alpha^{[m,c'-2]}(b)|$. Since $c-1 \notin F_\alpha(b) \Rightarrow \alpha^{k'}[c-1] = c'-1 \notin F_\alpha(b)$,

$$|F_\alpha^{[m,c-1]}(b)| \ge |F_\alpha^{[\alpha[c-1],c-2]}(b)| + \sum_{k=1}^{k'-1} |F_\alpha^{[\alpha^{k+1}[c-1],\alpha^k[c-1]-1]}(b)| + |F_\alpha^{[m,c'-1]}(b)|.$$

Also, $|F_\alpha^{[m,c'-1]}(b)| \ge 1$ follows from Lemma 11. Hence, the lemma holds. □

Lemma 13. *For any $\alpha \in \mathrm{A}, 1 \le b < b' \le |\alpha|$ and $1 \le i < |\alpha|$, $|F_\alpha^{[\alpha[i+1],i]}(b)| \ge |F_\alpha^{[\alpha[i+1],i]}(b')|$.*

Proof. We will prove the lemma by induction on i. First, for any $1 \le i < b$, it is clear that $|F_\alpha^{[\alpha[i+1],i]}(b)| = |F_\alpha^{[\alpha[i+1],i]}(b')| = 0$. Second, for any $b \le i < b'$, it follows from Lemma 11 that $|F_\alpha^{[\alpha[i+1],i]}(b)| \ge 1$. Then, $|F_\alpha^{[\alpha[i+1],i]}(b)| \ge 1 > 0 = |F_\alpha^{[\alpha[i+1],i]}(b')|$. Finally, when $b' \le i < |\alpha|$, let k' be the integer such that $\alpha^{k'}[i] = \alpha[i+1]-1$. (I) When $i \notin F_\alpha(b')$ or $\alpha^{k'}[i] = \alpha[i+1]-1 \in F_\alpha(b')$. It follows from Lemma 11 that $|F_\alpha^{[\alpha[i+1],i]}(b)| \ge |F_\alpha^{[\alpha[i],i-1]}(b)| + \sum_{k=1}^{k'-1} |F_\alpha^{[\alpha^{k+1}[i],\alpha^k[i]-1]}(b)|$ and $|F_\alpha^{[\alpha[i+1],i]}(b')| = |F_\alpha^{[\alpha[i],i-1]}(b')| + \sum_{k=1}^{k'-1} |F_\alpha^{[\alpha^{k+1}[i],\alpha^k[i]-1]}(b')|$. By the induction hypothesis, we have $|F_\alpha^{[\alpha[i],i-1]}(b)| \ge |F_\alpha^{[\alpha[i],i-1]}(b')|$ and $|F_\alpha^{[\alpha^{k+1}[i],\alpha^k[i]-1]}(b)| \ge |F_\alpha^{[\alpha^{k+1}[i],\alpha^k[i]-1]}(b')|$ for any $1 \le k \le k'-1$. Hence, $|F_\alpha^{[\alpha[i+1],i]}(b)| \ge |F_\alpha^{[\alpha[i+1],i]}(b')|$. (II) When $i \in F_\alpha(b')$ and $\alpha^{k'}[i] = \alpha[i+1]-1 \notin F_\alpha(b')$. There always exists $b' \in \{i, \alpha^1[i], \ldots, \alpha^{k'-1}[i]\}$, and therefore $|F_\alpha^{[\alpha[b'],b'-1]}(b)| \ge 1 > 0 = |F_\alpha^{[\alpha[b'],b'-1]}(b')|$. Then, $|F_\alpha^{[\alpha[i+1],i]}(b)| \ge |F_\alpha^{[\alpha[i],i-1]}(b)| + \sum_{k=1}^{k'-1} |F_\alpha^{[\alpha^{k+1}[i],\alpha^k[i]-1]}(b)| \ge 1 + |F_\alpha^{[\alpha[i],i-1]}(b')| + \sum_{k=1}^{k'-1} |F_\alpha^{[\alpha^{k+1}[i],\alpha^k[i]-1]}(b')| = |F_\alpha^{[\alpha[i+1],i]}(b')|$. Hence, $|F_\alpha^{[\alpha[i+1],i]}(b)| \ge |F_\alpha^{[\alpha[i+1],i]}(b')|$. □

In a similar way, we have the next lemma.

Lemma 14. *Let $\alpha \in \mathrm{A}$ and $c \in C_\alpha$ with $CT_\alpha(c) = \{c'\}$. For any $1 \le i < |\alpha|$, $|F_\alpha^{[\alpha[i+1],i]}(c)| \ge \sum_{g \in G} |F_\alpha^{[\alpha[i+1],i]}(g)|$, where $G = (C_\alpha^{[c,|\alpha|]}(c) - c')$.*

Now, we are ready to prove Lemma 10. We will use Lemmas 13 and 14.

Proof. Let $G = \{g \mid g \in C_\beta^{[i,r]}(i), z[g] = 0\}$. Let v be the sequence s.t. for each $1 \le j \le r$, $v[j] = 0$ if $j \in F_\beta(i)$, $v[j] = \star$ if there is $g \in G$ s.t. $j \in F_\beta(g)$, and $v[j] = z[j]$ otherwise.

Now we show $v \in \mathrm{Z}_\beta$. By the definition of v and $u[1:i-1]0 \in \mathrm{Z}_\beta^i$, it is clear that $v[j]$ holds Conditions 1, 2, 3 and 4 of Lemma 8 for any $1 \le j \le r$.

Furthermore, $u[1:i-1]\star \in \mathrm{Z}_\beta^i$ means that $zeros(v[\beta[i]:i-1]) \geq maxc_\beta(i)-1$. Hence, $v[c]$ satisfies Conditions 5 and 6 for any $c \in C_\beta^{[1,r]}(i)$ since $zeros(v[\beta[c]:c-1]) \geq zeros(v[\beta[i]:i-1])$ and $maxc_\beta(i)-1 \geq maxc_\beta(c)-1$. Then, as the proof of Lemma 9, we have only to show $zeros(v[\beta[c]:c-1]) \geq zeros(z[\beta[c]:c-1])$ for any $c \in C_\beta$. This can be proven by showing $|F_\beta^{[\beta[c],c-1]}(i)| \geq \sum_{g\in G}|F_\beta^{[\beta[c],c-1]}(g)|$. Since it is clear in case where $G=\emptyset$, we consider the case where $G \neq \emptyset$. Let $c' = CT_\beta(i)$. Note that $|CT_\beta(i)| = 1$ by the assumption. (I) When $z[c'] = 0$. Since $z[1:r]$ satisfies Condition 4 of Lemma 8, $G = \{c'\}$. It follows from Lemma 13 that $|F_\beta^{[\beta[c],c-1]}(i)| \geq |F_\beta^{[\beta[c],c-1]}(c')|$ for any $c \in C_\beta^{[1,r]}$. (II) When $z[c'] \neq 0$. It follows from Lemma 14 that $|F_\beta^{[\beta[c],c-1]}(i)| \geq \sum_{g\in G}|F_\beta^{[\beta[c],c-1]}(g)|$ for any $c \in C_\beta^{[1,r]}$. Therefore, the lemma holds. □

4.3 Complexity Analysis

Algorithm 1 shows our algorithm that solves Problem 1.

Theorem 1. *Algorithm 1 solves Problem 1 in $O(n^{1.5})$ time and $O(n)$ space for an unbounded alphabet.*

Proof. The correctness should be clear from the discussions in the previous subsections.

Let us estimate the time complexity of Algorithm 1 until the `CheckPBA` function is called at Line 1. As in the failure function construction algorithm, the while loop of Line 6 is executed at most n times. Moreover, for any $1 \leq i \leq n$, the values of $z[i]$, $prevc[i]$, and $order[i]$ are updated at most once. When i is a conflict point, Line 20 is executed at most $order_y(i)-1$ times. Hence, it follows from Lemma 5 that the total number of times Line 20 is executed is $\sum_{c\in C_y}(order_y(c)-1) \leq 1+\sum_{c\in C_y}\lfloor 2^{order_y(c)-2}\rfloor \leq n$.

Next, we show the `CheckPBA` function takes in $O(n^{1.5})$ time for any input $\alpha \in \mathrm{A}$. Let $2 \leq r_1 < r_2 < \cdots < r_x \leq n$ be the positions for which we execute Line 6 or 10 when we first visit these positions. If such positions do not exist, `CheckPBA` returns "valid" in $O(n)$ time. Let us consider $x \geq 1$. For any $1 \leq t \leq x$, let $z_t[1:r_t-1]$ denote the z-pattern when we first visit r_t and let $l_t = \min\{c \mid c \in AC_\alpha^{[1,r_t-1]}, z_t[c] = 0\}$. If $x = 1$ and such l_1 does not exist, then `CheckPBA` returns "invalid" in $O(n)$ time. If $x > 1$, then there exists l_1 as we reach r_x. Furthermore, there exists l_t s.t. $l_t < r_1$ since otherwise we cannot get across r_1. Henceforth, we may assume $l_1 \leq l_2 \leq \cdots \leq l_x$ exist. Note that by the definition of active conflict points, all elements of $F_\alpha(l_t)-\{l_t\}$ are not conflict points, and therefore for any $b \in F_\alpha(l_t)$, $z_t[b] = 0$.

Here, let $L_1 = \{c \mid c \in C_\alpha^{[l_1+1,r_1]}, l_1 < \max C_\alpha^{[1,c]}(c)\}$ and $L_t = \{c \mid c \in C_\alpha^{[r_{t-1}+1,r_t]}, l_t < \max C_\alpha^{[1,c]}(c)\}$ for any $1 < t \leq x$. Since $L_1, L_2, \ldots, L_x$ are pairwise disjoint, $|L| = \sum_{t=1}^x |L_t|$, where $L = \bigcup_{t=1}^x L_t$. It follows from Lemma 12 that $|F_\alpha^{[\alpha[r_t],r_t-1]}(l_t)| - |F_\alpha^{[\alpha[r_{t-1}],r_{t-1}-1]}(l_t)| \geq |L_t|$. In addition, for any $1 \leq t \leq x$, let $E_t^{in} = E_\alpha \cap ([\alpha[r_t], r_t-1] - [\alpha[r_{t-1}], r_{t-1}-1])$ and $E_t^{out} = E_\alpha \cap$

Algorithm 1. Algorithm to verify p-border array

Input: an integer array $y[1:n]$
Output: whether y is a valid p-border array or not
/* $zeros[1:n]$: $zeros[i] = zeros(z[1:i])$. $zeros[0] = 0$ for convenience. */
/* $sign[1:n]$: $sign[i] = 1$ if $i \in E_y$, $sign[i] = -1$ if $(C_y^{[i,n]}(i) \cap E_y) \neq \emptyset$. */
/* $prevc[1:n]$: $prevc[i] = \max C_y^{[1,i]}(i)$, $prevc[i] = 0$ otherwise. */
1 **if** $y[1:2] \neq [0,1]$ **then return** invalid;
2 $sign[1:n] \leftarrow [1,0,..,0]$; $prevc[1:n] \leftarrow [0,..,0]$; $order[1:n] \leftarrow [0,..,0]$; $maxc[1:n] \leftarrow [0,..,0]$;
3 **for** $i = 3$ **to** n **do**
4 **if** $y[i] = y[i-1] + 1$ **then** continue;
5 $b' \leftarrow y[i-1]$; $b \leftarrow y[b']$;
6 **while** $b > 0$ & $y[i] \neq y[b'+1]$ & $y[i] \neq b+1$ **do**
7 $b' \leftarrow b$; $b \leftarrow y[b']$;
8 **if** $y[i] = y[b'+1]$ **then** /* i conflicts with $b'+1$ */
9 $j \leftarrow y[i]$;
10 **while** $sign[j] = 0$ & $order[j] = 0$ **do** /* $z[y^1[i]], z[y^2[i]], \ldots, z[0]$ must be 0 */
11 $sign[j] \leftarrow 1$; $j \leftarrow y[j]$;
12 **if** $sign[j] = -1$ **then return** invalid;
13 **if** $sign[j] \neq 1$ **then**
14 $sign[j] \leftarrow 1$; $j \leftarrow prevc[j]$;
15 **while** $j > 0$ **do** /* $\forall j \in C_y^{[1,i]}(i), z[j]$ must be $\star$ */
16 **if** $sign[j] = 1$ **then return** invalid;
17 $sign[j] \leftarrow -1$; $j \leftarrow prevc[j]$;
18 **if** $order[b'+1] = 0$ **then** $order[b'+1] \leftarrow 1$;
19 $prevc[i] \leftarrow b'+1$; $order[i] \leftarrow order[b'+1] + 1$; $maxc[i] \leftarrow order[b'+1] + 1$; $j \leftarrow b'+1$;
20 **while** $j > 0$ & $maxc[j] < order[b'+1] + 1$ **do**
21 $maxc[j] \leftarrow order[b'+1] + 1$; $j \leftarrow prevc[j]$;
22 **else if** $y[i] \neq b+1$ **then return** invalid;
23 $cnt[1:n] \leftarrow [-1,..,-1]$; $zeros[1] \leftarrow 1$;
24 **return** CheckPBA$(2, n, y[1:n], zeros[1:n], sign[1:n], cnt[1:n], prevc[1:n], order[1:n], maxc[1:n])$;

$([\alpha[r_{t-1}], r_{t-1} - 1] - [\alpha[r_t], r_t - 1]\})$, where $[\alpha[r_0], r_0 - 1] = \emptyset$. Since for any $1 < t \leq x$, $zeros(z_t[\alpha[r_{t-1}] : r_{t-1} - 1]) \geq zeros(z_{t-1}[\alpha[r_{t-1}] : r_{t-1} - 1]) + 1$,

$$\begin{aligned} & zeros(z_t[\alpha[r_t] : r_t - 1]) \\ \geq\ & zeros(z_t[\alpha[r_{t-1}] : r_{t-1} - 1]) + |E_t^{in}| - |E_t^{out}| \\ & + |F_\alpha^{[\alpha[r_t], r_t - 1]}(l_t)| - |F_\alpha^{[\alpha[r_{t-1}], r_{t-1} - 1]}(l_t)| \\ \geq\ & zeros(z_{t-1}[\alpha[r_{t-1}] : r_{t-1} - 1]) + 1 + |E_t^{in}| - |E_t^{out}| + |L_t|. \end{aligned}$$

Function. `CheckPBA`$(i, n, y[1:n], zeros[1:n], sign[1:n], cnt[1:n], prevc[1:n], order[1:n], maxc[1:n])$

Result: whether y is a valid p-border array or not

if $i = n$ **then return** valid;
if $order[i] = 0$ **then** /* i is not a conflict point */
  $zeros[i] \leftarrow zeros[i-1] + zeros[y[i]] - zeros[y[i]-1]$;
  return `CheckPBA`$(i+1, n, y[1:n], \ldots, maxc[1:n])$;
if $sign[i] = 1$ **then** /* $z[i]$ must be 0 */
  if $zeros[i-1] - zeros[y[i]-1] < maxc[i] - 1$ **then return** invalid;
  $zeros[i] \leftarrow zeros[i-1] + 1$;
  return `CheckPBA`$(i+1, n, y[1:n], \ldots, maxc[1:n])$;
if $sign[i] = -1 \parallel zeros[i-1] - zeros[y[i]-1] < maxc[i] - 1$ **then** /* $z[i]$ must be $\star$ */
  if $zeros[i-1] - zeros[y[i]-1] < order[i]$ **then return** invalid;
  $zeros[i] \leftarrow zeros[i-1]$;
  return `CheckPBA`$(i+1, n, y[1:n], \ldots, maxc[1:n])$;
/* from here $sign[i] = 0$ and $zeros[i-1] - zeros[y[i]-1] \geq maxc[i] - 1$ */
if $cnt[i] = -1$ **then** /* first time arriving at i */
  $cnt[i]{+}{+}$; $cnt[prevc[i]]{+}{+}$
if $prevc[i] > 0$ & $sign[prevc[i]] = 1$ **then** /* $\exists c \in C_y^{[1,i]}(i), z[c] = 0$ */
  $sign[i] \leftarrow 1$; $zeros[i] \leftarrow zeros[i-1]$;
  $ret \leftarrow$ `CheckPBA`$(i+1, n, y[1:n], \ldots, maxc[1:n])$; $sign[i] \leftarrow 0$;
  return ret;
$sign[i] \leftarrow 1$; $zeros[i] \leftarrow zeros[i-1] + 1$;
$ret \leftarrow$ `CheckPBA`$(i+1, n, y[1:n], \ldots, maxc[1:n])$; $sign[i] \leftarrow 0$;
if $ret =$ valid $\parallel cnt[i] < 2$ **then return** ret;
$zeros[i] \leftarrow zeros[i-1]$;
return `CheckPBA`$(i+1, n, y[1:n], \ldots, maxc[1:n])$;

By recursive procedures, we have $order_\alpha(r_x) \geq 1 + zeros(z_x[\alpha[r_x] : r_x - 1]) \geq zeros(z_1[\alpha[r_1] : r_1 - 1]) + x + \sum_{t=2}^{x} |E_t^{in}| - \sum_{t=2}^{x} |E_t^{out}| + \sum_{t=2}^{x} |L_t|$. Since $zeros(z_1[\alpha[r_1] : r_1 - 1]) \geq 1 + |E_1^{in}| + |L_1|$ and $\sum_{t=1}^{x} |E_t^{in}| - \sum_{t=2}^{x} |E_t^{out}| \geq 1$, then $order_\alpha(r_x) \geq 2 + x + |L|$.

Now, we evaluate the number of z-patterns we search for during the calls of `CheckPBA`. Let $C_2(t) = \{c \mid c \in C_\alpha^{[l_t, r_t]}, |CT_\alpha^{[l_t, r_t]}(c)| \geq 2\}$ for any $1 \leq t \leq x$ and $T' = \{1\} \cup \{t \mid 1 < t \leq x, l_{t-1} < l_t, |CT_\alpha^{[l_t, r_{t-1}]}(l_t)| = 0\}$. Let us assume $T' = \{t'_1, t'_2, \ldots, t'_{x'}\}$ with $1 = t'_1 < t'_2 < \cdots < t'_{x'} \leq x$. By Lemmas 9 and 10, the number of z-patterns searched for between $l_{t'_j}$ and $r_{t'_{j+1}-1}$ is at most $2^{|C'_2(t'_j)|}$ for any $1 \leq j \leq x'$, where $t'_{x'+1} - 1 = x$ and $C'_2(t'_j) = \bigcup_{t=t'_j}^{t'_{j+1}-1} C_2(t)$. Then, the total number of z-patterns is at most $\sum_{j=1}^{x'} 2^{|C'_2(t'_j)|}$. By Lemma 10, for any $1 \leq j < x'$, $l_{t'_j}$ must be in $C'_2(t'_j)$ and by the definition of T', $l_{t'_j}$ is only in $C'_2(t'_j)$. Hence, if $C_2 = \bigcup_{t=1}^{x} C_2(t)$, then $|C'_2(t'_j)| \leq |C_2| - (x' - 2)$, and therefore $\sum_{j=1}^{x'} 2^{|C'_2(t'_j)|} \leq 4x'2^{|C_2|-x'}$.

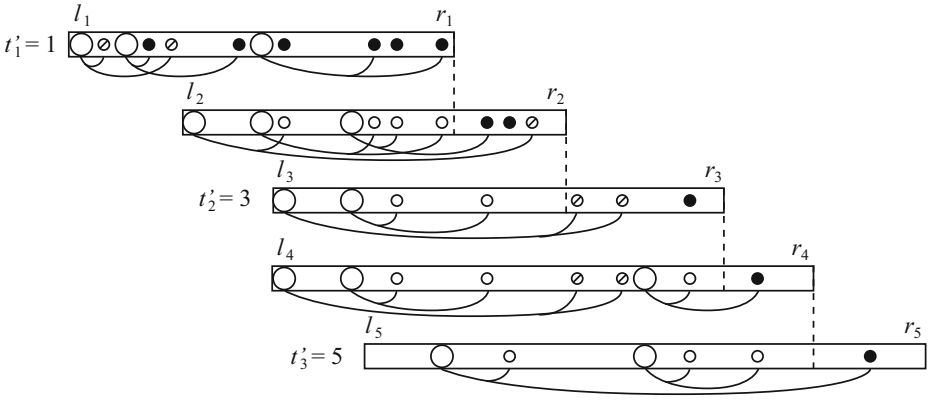


Fig. 4. Relation between L and C_2. A pair of a big circle and a small circle connected by an arc represents a parent-child relation in the conflict tree. ◯ is a position in C. ● or ○ is a position in L. ⊘ is a position not in L.

Finally, we consider the relation between L and C_2 (See Fig. 4). By the definition of L and C_2, for any $c \in (C_2 - \{l_1, l_2, \ldots, l_x\})$, $|CT_\alpha(c) \cap L| \geq 2$. In addition, by the definition of T', for any $c \in (C_2 \cap \{l_1, l_2, \ldots, l_x\} - \{l_{t'_1}, l_{t'_2}, \ldots, l_{t'_{x'}}\})$, $|CT_\alpha(c) \cap L| \geq 1$. Here, let $x'' = |\{l_1, l_2, \ldots, l_x\} - \{l_{t'_1}, l_{t'_2}, \ldots, l_{t'_{x'}}\}|$. Clearly, $x'+x'' \leq x$. For these reasons, $order_\alpha(r_x) \geq 2+x+|L| \geq 2+x+2|C_2|-2(x'+x'')+x'' \geq 2+2|C_2|-x'$. It follows from Lemma 5 that $n \geq 1+\sum_{c\in C_\alpha} \lfloor 2^{order_\alpha(c)-2} \rfloor > 1+\sum_{i=2}^{2+2|C_2|-x'} 2^{i-2} = 2^{2|C_2|-x'+1}$ and $\sqrt{n} > 2^{\frac{1+x'}{2}} 2^{|C_2|-x'} > x' 2^{|C_2|-x'}$. Hence, the total time complexity is proportional to $n \sum_{j=1}^{x'} 2^{|C'_2(t'_j)|} \leq 4nx' 2^{|C_2|-x'} < 4n\sqrt{n}$.

The space complexity is $O(n)$ as we use only a constant number of arrays of length n. □

5 Conclusions and Open Problems

We presented an $O(n^{1.5})$-time $O(n)$-space algorithm to verify if a given integer array y of length n is a valid p-border array for an unbounded alphabet. In case y is a valid p-border array, the proposed algorithm also computes a z-pattern $z \in \{0, \star\}^*$ s.t. $z \in \mathrm{Z}_y$, and we remark that some sequence $p \in \mathrm{P}_y$ s.t. $ptoz(p) = z$ is then computable in linear time from z.

Open problems of interest are: (1) Can we solve the p-border array reverse problem for an unbounded alphabet in $o(n^{1.5})$ time? (2) Can we efficiently solve the p-border array reverse problem for a bounded alphabet? (3) Can we efficiently count p-border arrays of length n?

References

1. Baker, B.S.: Parameterized pattern matching: Algorithms and applications. Journal of Computer and System Sciences 52(1), 28–42 (1996)
2. Amir, A., Farach, M., Muthukrishnan, S.: Alphabet dependence in parameterized matching. Information Processing Letters 49(3), 111–115 (1994)
3. Kosaraju, S.: Faster algorithms for the construction of parameterized suffix trees. In: Proc. FOCS 1995, pp. 631–637 (1995)
4. Hazay, C., Lewenstein, M., Sokol, D.: Approximate parameterized matching. ACM Transactions on Algorithms 3(3), Article No. 29 (2007)
5. Apostolico, A., Erdös, P.L., Lewenstein, M.: Parameterized matching with mismatches. Journal of Discrete Algorithms 5(1), 135–140 (2007)
6. I, T., Deguchi, S., Bannai, H., Inenaga, S., Takeda, M.: Lightweight parameterized suffix array construction. In: Proc. IWOCA, pp. 312–323 (2009)
7. Idury, R.M., Schäffer, A.A.: Multiple matching of parameterized patterns. Theoretical Computer Science 154(2), 203–224 (1996)
8. Morris, J.H., Pratt, V.R.: A linear pattern-matching algorithm. Technical Report 40, University of California, Berkeley (1970)
9. I, T., Inenaga, S., Bannai, H., Takeda, M.: Counting parameterized border arrays for a binary alphabet. In: Dediu, A.H., Ionescu, A.M., Martín-Vide, C. (eds.) LATA 2009. LNCS, vol. 5457, pp. 422–433. Springer, Heidelberg (2009)
10. Franek, F., Gao, S., Lu, W., Ryan, P.J., Smyth, W.F., Sun, Y., Yang, L.: Verifying a border array in linear time. J. Comb. Math. and Comb. Comp. 42, 223–236 (2002)
11. Duval, J.P., Lecroq, T., Lefevre, A.: Border array on bounded alphabet. Journal of Automata, Languages and Combinatorics 10(1), 51–60 (2005)
12. Duval, J.P., Lefebvre, A.: Words over an ordered alphabet and suffix permutations. Theoretical Informatics and Applications 36, 249–259 (2002)
13. Bannai, H., Inenaga, S., Shinohara, A., Takeda, M.: Inferring strings from graphs and arrays. In: Rovan, B., Vojtáš, P. (eds.) MFCS 2003. LNCS, vol. 2747, pp. 208–217. Springer, Heidelberg (2003)
14. Schürmann, K.B., Stoye, J.: Counting suffix arrays and strings. Theoretical Computer Science 395(2-1), 220–234 (2008)
15. Clément, J., Crochemore, M., Rindone, G.: Reverse engineering prefix tables. In: Proc. STACS 2009, pp. 289–300 (2009)
16. Duval, J.P., Lecroq, T., Lefebvre, A.: Efficient validation and construction of border arrays and validation of string matching automata. RAIRO - Theoretical Informatics and Applications 43(2), 281–297 (2009)
17. Gawrychowski, P., Jez, A., Jez, L.: Validating the Knuth-Morris-Pratt failure function, fast and online. In: Proc. CSR 2010 (to appear 2010)
18. Crochemore, M., Iliopoulos, C., Pissis, S., Tischler, G.: Cover array string reconstruction. In: Proc. CPM 2010 (to appear 2010)
19. Moore, D., Smyth, W., Miller, D.: Counting distinct strings. Algorithmica 23(1), 1–13 (1999)

Cover Array String Reconstruction

Maxime Crochemore[1,2], Costas S. Iliopoulos[1,3],
Solon P. Pissis[1], and German Tischler[1,4]

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

Abstract. A proper factor u of a string y is a cover of y if every letter of y is within some occurrence of u in y. The concept generalises the notion of periods of a string. An integer array C is the minimal-cover (resp. maximal-cover) array of y if $C[i]$ is the minimal (resp. maximal) length of covers of $y[0\,..\,i]$, or zero if no cover exists.

In this paper, we present a constructive algorithm checking the validity of an array as a minimal-cover or maximal-cover array of some string. When the array is valid, the algorithm produces a string over an unbounded alphabet whose cover array is the input array. All algorithms run in linear time due to an interesting combinatorial property of cover arrays: the sum of important values in a cover array is bounded by twice the length of the string.

1 Introduction

The notion of periodicity in strings is well studied in many fields like combinatorics on words, pattern matching, data compression and automata theory (see [11,12]), because it is of paramount importance in several applications, not to talk about its theoretical aspects.

The concept of quasiperiodicity is a generalisation of the notion of periodicity, and was defined by Apostolico and Ehrenfeucht in [2]. In a periodic repetition the occurrences of the single periods do not overlap. In contrast, the quasiperiods of a quasiperiodic string may overlap. We call a proper factor u of a nonempty string y a cover of y, if every letter of y is within some occurrence of u in y. In this paper, we consider the so-called *aligned covers*, where the cover u of y needs to be a border (i.e. a prefix and a suffix) of y. The array C is called the *minimal-cover* (resp. *maximal-cover*) *array* of the string y of length n, if for each i, $0 \leq i < n$, $C[i]$ stores either the length of the shortest (resp. longest) cover of $y[0\,..\,i]$, when such a cover exists, or zero otherwise. In particular, we do not consider a string to be a cover of itself.

Apostolico and Breslauer [1,4] gave an online linear runtime algorithm computing the minimal-cover array of a string. In their definition, a string is a cover

A. Amir and L. Parida (Eds.): CPM 2010, LNCS 6129, pp. 251–259, 2010.

of itself, but it is straightforward to modify their algorithm to accommodate our definition. Li and Smyth [10] provided an online linear runtime algorithm for computing the maximal-cover array.

In this paper, we present a constructive algorithm checking if an integer array is the minimal-cover or maximal-cover array of some string. When the array is valid, the algorithm produces a string over an unbounded alphabet whose cover array is the input array. For our validity checking algorithm, we use the aforementioned algorithms that compute cover arrays.

All algorithms run in linear time. This is essentially due to a combinatorial property of cover arrays: the sum of important values in a cover array is bounded by twice the length of the string.

The result of the paper completes the series of algorithmic characterisations of data structures that store fundamental features of strings. They concern Border arrays [6,7], Parameterized Border arrays [9] and Prefix arrays [5] that stores periods of all the prefixes of a string, as well as the element of Suffix arrays [3,8] that memorises the list of positions of lexicographically sorted suffixes of the string. The question is not applicable to complete Suffix trees or Suffix automata since the relevant string is part of these data structures. The algorithms may be regarded as reverse engineering processes and, beyond their obvious theoretical interest, they are useful to test the validity of some constructions. Their linear runtime is an important part of their quality.

The rest of the paper is structured as follows. Section 2 presents the basic definitions used throughout the paper and the problem. In Section 3, we prove some properties of minimal-cover arrays used later for the design or the analysis of algorithms. In Section 4, we describe our constructive cover array validity checking algorithms. Section 5 provides some combinatorially interesting numerical data on minimal-cover arrays.

2 Definitions and Problems

Throughout this paper we consider a string y of length $|y| = n$ on an unbounded alphabet. It is represented as $y[0\,..\,n-1]$. A string w is a *factor* of y if $y = uwv$ for two strings u and v. It is a *prefix* of y if u is empty and a *suffix* of y if v is empty. A string u is a *period* of y if y is a prefix of u^k for some positive integer k, or equivalently if y is a prefix of uy. The period of y is the shortest period of y. A string x of length m is a *cover* of y if both $m < n$ and there exists a set of positions $P \subseteq \{0, \ldots, n-m\}$ satisfying $y[i\,..\,i+m-1] = x$ for all $i \in P$ and $\bigcup_{i \in P}\{i, \ldots, i+m-1\} = \{0, \ldots, n-1\}$. Note that this requires x to be a prefix as well as a suffix of y. The *minimal-cover array* C of y is the array of integers $C[0\,..\,n-1]$ for which $C[i]$, $0 \le i < n$, stores the length of the shortest cover of the prefix $y[0\,..\,i]$, if such a cover exists, or zero otherwise. The *maximal-cover array* C^{M} stores longest cover at each position instead. The following table provides the minimal-cover array C and the maximal-cover array C^{M} of the string $y = \texttt{abaababaabababaababaaba}$.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
$y[i]$	a	b	a	a	b	a	b	a	a	b	a	b	a	a	b	a	a	b	a	b	a	a	b	a
$C[i]$	0	0	0	0	0	3	0	3	0	5	3	7	3	9	5	3	0	5	3	0	3	9	5	3
$C^{\mathrm{M}}[i]$	0	0	0	0	0	3	0	3	0	5	6	7	8	9	10	11	0	5	6	0	8	9	10	11

We consider the following problems for an integer array A:

Problem 1 (Minimal Validity Problem). Decide if A is the minimal-cover array of some string.

Problem 2 (Maximal Validity Problem). Decide if A is the maximal-cover array of some string.

Problem 3 (Minimal Construction Problem). When A is a valid minimal-cover array, exhibit a string over an unbounded alphabet whose minimal-cover array is A.

Problem 4 (Maximal Construction Problem). When A is a valid maximal-cover array, exhibit a string over an unbounded alphabet whose maximal-cover array is A.

3 Properties of the Minimal-Cover Array

In this section, we assume that C is the minimal-cover array of y. Its first element is 0, as we do not consider a string to be a cover of itself. Next elements are 1 only for prefixes of the form a^k for some letter a. We will use the following fact in our argumentation.

Fact 1 (Transitivity). *If u and v cover y and $|u| < |v|$, then u covers v.*

For the rest of the section we assume that $n > 1$ and prove several less obvious properties of the minimal-cover array.

Lemma 1. *If $0 \leq i < n$ and $C[i] \neq 0$, then $C[C[i] - 1] = 0$.*

Proof. Immediate from Fact 1. □

Lemma 2. *Let i and j be positions such that $j < i$, $j - C[j] \geq i - C[i]$, $C[i] \neq 0$ and $C[j] \neq 0$. Furthermore let $r = j - (i - C[i] + 1)$. If $i - C[i] = j - C[j]$ then $C[r] = 0$, otherwise if $i - C[i] < j - C[j]$, then $C[r] = C[j]$.*

Proof. First assume that $i - C[i] = j - C[j]$. Then $C[j - (i - C[i] + 1)] = C[j - (j - C[j] + 1)] = C[C[j] - 1] = 0$ according to Lemma 1. Now assume that $i - C[i] < j - C[j]$. This situation is depicted in Figure 1. The string $u = y[j - C[j] + 1 \,..\, j]$ of length $C[j]$ covers the string $y[0 \,..\, j]$. By precondition $(j < i, j - C[j] > i - C[i])$ u also covers $y[0 \,..\, r]$, as there exists an occurrence of u at position $r - C[j] + 1$. Thus we have $C[r] \leq C[j]$. The assumption $C[r] < C[j]$ leads to a contradiction to the minimality of C. Thus we have $C[r] = C[j]$. □

Fig. 1. Case $j - C[j] > i - C[i]$ of Lemma 2 ($i' = i - C[i] + 1$, $j' = j - C[j] + 1$)

Lemma 3. *Let i and j be positions such that $j < i$ and $j - C[j] < i - C[i]$. Then $(i - C[i]) - (j - C[j]) > C[j]/2$.*

Proof. For ease of notation let $p = C[i]$, $q = C[j]$ and $r = (i - p) - (j - q)$. Assume the statement does not hold, i.e. $r \leq \frac{q}{2}$. Let $u = y[0\,..\,r-1]$. Then due to the overlap of $y[i - p + 1\,..\,i]$ and $y[j - q + 1\,..\,j]$ both $y[0\,..\,p-1]$ and $y[0\,..\,q-1]$ are powers of u. Let $y[0\,..\,q-1] = u^e$ for some exponent e. Observe that $e = q/r \geq q/(q/2) = 2$. However $y[0\,..\,q-1]$ is also covered by $v = u^{1+e-\lfloor e \rfloor}$. As $|v| < q$ we obtain a contradiction. □

Definition 1. *A position $j \neq 0$ of C is called* totally covered*, if there is a position $i > j$ for which $C[i] \neq 0$ and $i - C[i] + 1 \leq j - C[j] + 1 < j$.*

Let C^{p} be obtained from C by setting $C[i] = 0$ for all totally covered indices i on C. We call C^{p} the *pruned minimal-cover array* of y. The next table shows the pruned minimal-cover array of the example string above.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
$y[i]$	a	b	a	a	b	a	b	a	a	b	a	b	a	a	b	a	a	b	a	b	a	a	b	a
$C[i]$	0	0	0	0	0	3	0	3	0	5	3	7	3	9	5	3	0	5	3	0	3	9	5	3
$C^{\mathrm{p}}[i]$	0	0	0	0	0	3	0	0	0	0	0	0	0	9	5	0	0	0	0	0	0	9	5	3

Lemma 4. *The sum of the elements of C^{p} does not exceed $2n$.*

Proof. Let $I_i = \{i - C[i] + 1, i - C[i] + 2, \ldots i\}$ for $i = 0, \ldots, n-1$ if $C[i] \neq 0$ and $I_i = \emptyset$ otherwise. Let I'_i denote the lower half of I_i (if $C[i]$ is uneven, the middle element is included). According to Lemma 3, $i \neq j$ implies $I'_i \cap I'_j = \emptyset$. Thus the relation $\sum_{i=0}^{n-1} |I'_i| \leq n$ holds, which in turn implies $\sum_{i=0}^{n-1} |I_i| \leq \sum_{i=0}^{n-1} 2|I'_i| \leq 2n$. □

The bound of Lemma 4 is asymptotically tight. For an integer $k > 1$, let $x_k = (a^k b a^{k+1} b)^{n/(2k+3)}$. For $k = 2$ and $n = 23$ we get:

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
$y[i]$	a	a	b	a	a	a	b	a	a	b	a	a	a	b	a	a	b	a	a	a	b	a	a
$C^{\mathrm{p}}[i]$	0	0	0	0	0	0	0	0	5	0	0	0	0	7	0	5	0	0	0	0	7	0	5

It is straightforward to see that all segments of length $2k + 3$ of C^{p} contain the values $2k + 1$ and $2k + 3$, except at the beginning of the string. Therefore the sum of elements in C^{p} is $(4k + 4)(\frac{n}{2k+3} - 1)$, which tends to $2n$ when k (and n) goes to infinity.

4 Reverse Engineering a Cover Array

We solve the stated problems in three steps: array transformation, string inference and validity checking.

Transforming Maximal to Minimal-Cover Arrays. We first show how to transform a maximal-cover array into a minimal-cover array in linear time. The following algorithm MAXTOMIN converts the maximal-cover array C of y to its minimal-cover array in linear time.

```
MAXTOMIN(C, n)
1  for i ← 0 to n − 1 do
2      if C[i] ≠ 0 and C[C[i] − 1] ≠ 0 then
3          C[i] ← C[C[i] − 1]
```

The algorithm works in the following way. Assume the for loop is executing for some $i > 0$. At this time the prefix $C[0\,..\,i-1]$ of the array has been converted to a minimal-cover array and we want to determine the value of $C[i]$. If $C[i]$ is zero then there is no cover and we leave the value as it is. If $C[i]$ is not zero, then we know from Fact 1 that if there is a shorter cover, then it covers $y[0\,..\,C[i]-1]$. Thus we look up $C[C[i]-1]$ to see if there is a shorter cover we can substitute for $C[i]$. As the segment before index i is already the minimal-cover array up to this point, we know that such a value is minimal.

String Inference. In this step, we assume that the integer array C of length n is the minimal- or maximal-cover array of at least one string. From what we wrote above, we can assume without loss of generality that C is a minimal-cover array.

The nonzero values in C induce an equivalence relation on the positions of every string that has the minimal-cover array C. More precisely, if we find the value $\ell \neq 0$ in position i of C, then this imposes the constraints

$$y[k] = y[i - \ell + 1 + k]$$

for $k = 0, \ldots, \ell-1$. We say that the positions k and $i-\ell+1+k$ are bidirectionally linked. Let the undirected graph $G(V, E)$ be defined by $V = \{0, \ldots, n-1\}$ and

$$E = \bigcup_{i=0,\ldots,n-1} \bigcup_{j=0,\ldots C[i]-1} (\{(j, i - C[i] + 1 + j)\})\,.$$

Then the nonzero values in C state that the letters at positions i and j of any word y such that C is the minimal-cover array of y need to be equal, if i and j are connected in G. According to Lemma 2, we do not lose connectivity between vertices of G, if we remove totally covered indices from C, i.e. the graph induced by C has the same connected components as the one induced by its pruned version C^{p}. The number of edges in the graph induced by C^{p} is bounded by $2n$ according to Lemma 4.

The pruned minimal-cover array C^{p} can be obtained from the minimal-cover array C using the following algorithm PRUNE in linear time.

```
PRUNE(C, n)
 1  ℓ ← 0
 2  for i ← n − 1 downto 0 do
 3       if ℓ ≥ C[i] then
 4            C[i] ← 0
 5       ℓ ← max(0, max(ℓ, C[i]) − 1)
 6  return C
```

A non-zero value at index i in C defines an interval $[i-C[i]+1, i]$. The algorithm scans C from large to small indices, where the value ℓ stores the minimal lower bound of all the intervals encountered so far. If an interval starting at a smaller upper bound has a greater lower bound than ℓ, we erase the corresponding value in C by setting it to zero. Thus we remove all totally covered indices from C and obtain the pruned array C^{P}.

So far we know how to extract information from the non-zero values of C by computing connected components in a graph which has no more than $2n$ edges. The vertices in each connected component designate positions in any produced string which need to have equal letters. By assigning a different letter to each of these components, we make sure not to violate any constraints set by the zero values in C. The following algorithm MINARRAYTOSTRING produces a string y from an array A assumed to be a pruned minimal-cover array.

```
MINARRAYTOSTRING(A, n)
 1  ▷ Produce edges
 2  for i ← 0 to n − 1 do
 3       E[i] ← empty list
 4  for i ← 0 to n − 1 do
 5       for j ← 0 to A[i] − 1 do
 6            E[i − A[i] + 1 + j].add(j), E[j].add(i − A[i] + 1 + j)
 7  ▷ Compute connected components by Depth First Search
 8  ▷ and assign letters to output string
 9  (S, ℓ) ← (empty stack, −1)
10  for i ← 0 to n − 1 do
11       if y[i] is undefined then
12            S.push(i)
13            ℓ ← ℓ + 1
14            while not S.empty() do
15                 p ← S.pop()
16                 y[p] ← ℓ
17                 for each element j of E[p] do
18                      if y[j] is undefined then
19                           S.push(j)
20  return y
```

The first two for loops produce the edges E in the graph G induced by A, where we implement the transition relation by assigning a linear list of outgoing edges

to each vertex. The third for loop computes the connected components in the graph by using depth first search and assigns the letters to the output string. Each connected component is assigned a different letter. The runtime of the algorithm is linear in the number of edges of the graph, which is bounded by $2n$.

Theorem 1. *The problems* Minimal Construction Problem *and* Maximal Construction Problem *are solved in linear time by the algorithm* MinArrayToString *and the sequence of algorithms* MaxToMin *and* MinArrayToString *respectively.*

Validity Checking. In the third step we use the MinArrayToString algorithm as a building block for our validity checking algorithm. Thus we have to ensure some basic constraints so that the algorithm does not firstly access any undefined positions in the input and secondly runs in linear time. As a first step, we have to make sure that the algorithm will not try to define edges for which at least one vertex number is not valid. This check is performed by the following algorithm Precheck, which runs in linear time.

```
Precheck(A, n)
1  for i ← 0 to n − 1 do
2      if i − A[i] + 1 < 0 then
3          return false
4  return true
```

If Precheck returns true, then MinArrayToString will only generate edges from valid to valid vertices. If we are checking for validity of a maximal-cover array, we then pass the array through the algorithm MaxToMin as a next step. In both cases (minimal and maximal) the next step is to prune the array using the algorithm Prune. After this, we can call the algorithm MinArrayToString with the so-modified array, but it may not run in linear time, as the constraints imposed by Lemma 3 may not hold if the original input array is invalid. We avoid this situation with the following algorithm Postcheck.

```
Postcheck(A, n)
1  j ← −1
2  for i ← 0 to n − 1 do
3      if A[i] ≠ 0 then
4          if j ≠ −1 and (i − A[i]) − (j − A[j]) ≤ ⌊A[j]/2⌋ then
5              return false
6          j ← i
7  return true
```

If Postcheck returns false, then the input array was invalid. Otherwise we can call MinArrayToString and be sure that it will run in linear time. At this point we have obtained a string from the input array in linear time. We know that if the input array is valid, the minimal- or maximal- (depending on what kind of input we are given) cover array of this string matches the input.

If the input array is not valid, we cannot obtain it by computing the minimal- or maximal-cover array from the obtained string. Thus we can check whether an array A is a valid maximal-cover array using the following algorithm CheckMaximal.

CheckMaximal(A, n)
1 **if** Precheck$(A, n) =$ false **then**
2 **return** false
3 $A \leftarrow$ MaxToMin(A, n)
4 $A \leftarrow$ Prune(A, n)
5 **if** Postcheck$(A, n) =$ false **then**
6 **return** false
7 $y \leftarrow$ MinArrayToString(A, n)
8 **if** the maximal-cover array of y equals A **then**
9 **return** true
10 **else return** false

The algorithm CheckMinimal for checking whether an array A is a valid minimal-cover array is obtained from CheckMaximal by removing the call to the function MaxToMin in line 3 and checking whether the minimal instead of the maximal-cover array of the string y equals A in line 8.

Theorem 2. *The problems* Minimal Validity Problem *and* Maximal Validity Problem *are solved by the algorithms* CheckMinimal *and* CheckMaximal *respectively in linear time.*

5 Experiments and Numerical Results

Figure 2 shows the maximal ratio of sums of elements of pruned minimal-cover array, for all words over a two-letter alphabet, using even word lengths 8 to 30. These ratios are known to be smaller than 2 by Lemma 4. However, values close to this bound are not observed for small word length.

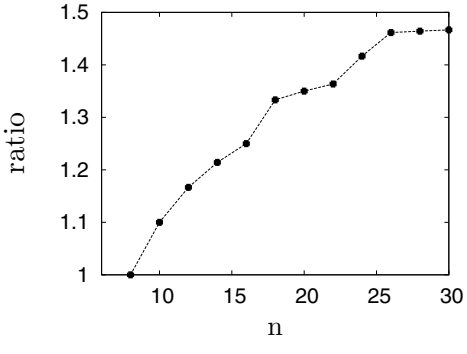


Fig. 2. Maximal ratio of sum over pruned minimal-cover array and word length for words over the binary alphabet of even length 8 to 30

We were able to verify the linear runtime of our algorithm in experiments. The implementation for the CheckMinimal function is available at the Website `http://www.dcs.kcl.ac.uk/staff/tischler/src/recovering-0.0.0.tar.bz2`, which is set up for maintaining the source code and the documentation.

6 Conclusion

In this paper, we have provided linear runtime algorithms for checking the validity of minimal- and maximal-cover arrays and algorithms to infer strings from valid minimal- and maximal-cover arrays. The linear time inference of strings using the least possible alphabet size from cover arrays remains an open problem.

References

1. Apostolico, A., Breslauer, D.: Of periods, quasiperiods, repetitions and covers. In: Mycielski, J., Rozenberg, G., Salomaa, A. (eds.) Structures in Logic and Computer Science. LNCS, vol. 1261, pp. 236–248. Springer, Heidelberg (1997)
2. Apostolico, A., Ehrenfeucht, A.: Efficient detection of quasiperiodicities in strings. Theoretical Computer Science 119(2), 247–265 (1993)
3. Bannai, H., Inenaga, S., Shinohara, A., Take, M.: Inferring strings from graphs and arrays. In: Rovan, B., Vojtáš, P. (eds.) MFCS 2003. LNCS, vol. 2747, pp. 208–217. Springer, Heidelberg (2003)
4. Breslauer, D.: An on-line string superprimitivity test. Information Processing Letters 44(6), 345–347 (1992)
5. Clement, J., Crochemore, M., Rindone, G.: Reverse engineering prefix tables. In: Albers, S., Marion, J.-Y. (eds.) 26th International Symposium on Theoretical Aspects of Computer Science (STACS 2009), Dagstuhl, Germany, pp. 289–300. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany (2009), `http://drops.dagstuhl.de/opus/volltexte/2009/1825`
6. Duval, J.-P., Lecroq, T., Lefebvre, A.: Border array on bounded alphabet. Journal of Automata, Languages and Combinatorics 10(1), 51–60 (2005)
7. Franek, F., Gao, S., Lu, W., Ryan, P.J., Smyth, W.F., Sun, Y., Yang, L.: Verifying a Border array in linear time. Journal on Combinatorial Mathematics and Combinatorial Computing 42, 223–236 (2002)
8. Franek, F., Smyth, W.F.: Reconstructing a Suffix Array. International Journal of Foundations of Computer Science 17(6), 1281–1295 (2006)
9. Tomohiro, I., Inenaga, S., Bannai, H., Takeda, M.: Counting parameterized border arrays for a binary alphabet. In: Dediu, A.H., Ionescu, A.M., Martín-Vide, C. (eds.) LATA 2009. LNCS, vol. 5457, pp. 422–433. Springer, Heidelberg (2009)
10. Li, Y., Smyth, W.F.: Computing the cover array in linear time. Algorithmica 32(1), 95–106 (2002)
11. Lothaire, M. (ed.): Algebraic Combinatorics on Words. Cambridge University Press, Cambridge (2001)
12. Lothaire, M. (ed.): Applied Combinatorics on Words. Cambridge University Press, Cambridge (2005)

Compression, Indexing, and Retrieval for Massive String Data*

Wing-Kai Hon[1], Rahul Shah[2], and Jeffrey Scott Vitter[3]

[1] National Tsing Hua University, Taiwan
wkhon@cs.nthu.edu.tw
[2] Louisiana State University, USA
rahul@csc.lsu.edu
[3] Texas A&M University, USA
jsv@ku.edu

Abstract. The field of compressed data structures seeks to achieve fast search time, but using a compressed representation, ideally requiring less space than that occupied by the original input data. The challenge is to construct a compressed representation that provides the same functionality and speed as traditional data structures. In this invited presentation, we discuss some breakthroughs in compressed data structures over the course of the last decade that have significantly reduced the space requirements for fast text and document indexing. One interesting consequence is that, for the first time, we can construct data structures for text indexing that are competitive in time and space with the well-known technique of inverted indexes, but that provide more general search capabilities. Several challenges remain, and we focus in this presentation on two in particular: building I/O-efficient search structures when the input data are so massive that external memory must be used, and incorporating notions of relevance in the reporting of query answers.

1 Introduction

The world is drowning in data! Massive data sets are being produced at unprecedented rates from sources like the World-Wide Web, genome sequencing, scientific experiments, business records, image processing, and satellite imagery. The proliferation of data at massive scales poses serious challenges in terms of storing, managing, retrieving, and mining information from the data.

Pattern matching — in which a pattern is matched against a massively sized text or sequence of data — is a traditional field of computer science that forms the basis for biological databases and search engines. Previous work has concentrated for the most part on the internal memory RAM model. However, we are increasingly having to deal with massive data sets that do not easily fit into internal memory and thus must be stored on secondary storage, such as disk drives, or in a distributed fashion in a network.

* Supported in part by Taiwan NSC grant 96–2221–E–007–082–MY3 (W. Hon) and USA National Science Foundation grant CCF–0621457 (R. Shah and J. S. Vitter).

A. Amir and L. Parida (Eds.): CPM 2010, LNCS 6129, pp. 260–274, 2010.

Suffix trees and suffix arrays, which are the traditional data structures used for pattern matching and a variety of other string processing tasks, are often "bloated" in that they require much more space than that occupied by the uncompressed input data. Moreover, the input data are typically highly compressible, often by a factor of 5–10. When compared with the size of the input data in compressed form, the size of suffix trees and suffix arrays can be prohibitively large, often 20–150 times larger than the compressed data size. This extra space blowup results in increased memory resources and energy usage, slower data access (because the bloated data must reside in the slower levels of the memory hierarchy), and reduced bandwidth.

1.1 Key Themes in This Presentation

In this presentation we focus on some emerging themes in the area of pattern matching for massive data. One theme deals with the exciting new field called *compressed data structures*, which addresses the bloat exhibited by suffix trees and suffix arrays. There are two simultaneous goals: space-efficient compression and fast indexing. The last decade has seen much progress, both in theory and in practice. A practical consequence is that, for the first time, we have space-efficient indexing methods for pattern matching and other tasks that can compete in terms of space and time with the well-known technique of inverted indexes [73,52,74] used in search engines, while offering more general search capabilities. Some compressed data structures are in addition *self-indexing* [61,19,20,28], and thus the original data can be discarded, making them especially space-efficient. The two main techniques we discuss — compressed suffix array (CSA) and FM-index — are self-indexing techniques that require space roughly equal to the space occupied by the input data in compressed format.

A second theme deals with external memory access in massive data applications [1,71,70], in which we measure performance in terms of number of I/Os. A key disadvantage of CSAs and the FM-index is that they do not exhibit locality of reference and thus do not perform well in terms of number of I/Os. If the input data are so massive that the CSA and FM-index do not fit in internal memory, their performance is slowed significantly. There is much interesting work on compressed data structures in external memory (e.g., [2,4,27,17,16,38,48,55]), but major challenges remain.

The technique of sparsification allows us to reduce space usage but at the same time exploit locality for good I/O performance and multicore utilization. We discuss sparsification in two settings: One involves a new transform called the geometric Burrows-Wheeler transform (GBWT) [9,34] that provides a link between text indexing and the field of range searching, which has been studied extensively in the external memory setting. In this case, a sparse subset of suffix array pointers are used to reduce space, and multiple offsets in the pattern must be searched, which can be done especially fast on multicore processors. The other setting introduces the notion of relevance in queries so that only the most relevant (or top-k) matches [53,6,64,69,36] are reported. The technique of sparsification provides approximate answers quickly in a small amount of space [36].

Besides the external memory scenario, other related models of interest worth exploring include the cache-oblivious model [25], data streams model [54], and practical programming paradigms such as multicore [65] and MapReduce [11].

2 Background

2.1 Text Indexing for Pattern Matching

We use $T[1..n]$ to denote an input string or text of n characters, where the characters are drawn from an alphabet Σ of size σ. The fundamental task of text indexing is to build an index for T so that, for any query pattern P (consisting of p characters), we can efficiently determine if P occurs in T. Depending upon the application, we may want to report all the *occ* locations of where P occurs in T, or perhaps we may merely want to report the number *occ* of such occurrences.

The string T has n suffixes, starting at each of the n locations in the text. The ith suffix, which starts at position i, is denoted by $T[i..n]$. The *suffix array* [26,49] $SA[1..n]$ of T is an array of n integers that gives the sorted order of the suffixes of T. That is, $SA[i] = j$ if $T[j..n]$ is the ith smallest suffix of T in lexicographical order. Similarly, the inverse suffix array is defined by $SA^{-1}[j] = i$. The *suffix tree ST* is a compact trie on all the suffixes of the text [51,72,68]. Suffix trees are often augmented with suffix links. The suffix tree can list all *occ* occurrences of P in $O(p + occ)$ time in the RAM model. Suffix arrays can also be used for pattern matching. If P appears in T, there exist indices ℓ and r such that $SA[\ell], SA[\ell+1], \ldots, SA[r]$ store all the starting positions in text T where P occurs. We can use the longest common prefix array to improve the query time from $O(p \log n + occ)$ to $O(p + \log n + occ)$ time.

Suffix trees and suffix arrays use $O(n)$ words of storage, which translates to $O(n \log n)$ bits. This size can be much larger than that of the text, which is $n \log \sigma$ bits, and substantially larger than the size of the text in compressed format, which we approximate by $nH_k(T)$, where $H_k(T)$ represents the kth-order empirical entropy of the text T.

2.2 String B-Trees

Ferragina and Grossi introduced the *string B-tree* (SBT) [16], an elegant and efficient index in the external memory model. The string B-tree acts conceptually as a B-tree over the suffix array; each internal node does B-way branching. Each internal node is represented as a "blind trie" with B leaves; each leaf is a pointer to one of the B child nodes. The blind trie is formed as the compact trie on the B leaves, except that all but the first character on each edge label is removed. When searching within a node in order to determine the proper leaf (and therefore child node) to go to next, the search may go awry since only the first character on each edge is available for comparison. The search will always end up at the right place when the pattern correctly matches one of the leaves, but in the case where there is no match and the search goes awry, a simple scanning of the original text can discover the mistake and find the corrected position where the pattern belongs.

Each block of the text is never scanned more than once and thus the string B-tree supports predecessor and range queries in $O(p/B + \log_B n + occ/B)$ I/Os using $O(n)$ words (or $O(n/B)$ blocks) of storage.

3 Compressed Data Structures

In the field of compressed data structures, the goal is to build data structures whose space usage is provably close to the entropy-compressed size of the text. A simultaneous goal is to maintain fast query performance.

3.1 Wavelet Trees

The *wavelet tree*, introduced by Grossi et al. [28,24], has become a key tool in modern text indexing. It supports rank and select queries on arrays of characters from Σ. (A rank query $rank(c, i)$ counts how many times character c occurs in the first i positions of the array. A select query $select(c, j)$ returns the location of the jth occurrence of c.) In a sense, the wavelet tree generalizes the rank and select operations from a bit array [59,57] to an arbitrary multicharacter text array T, and it uses $nH_0(T) + t + O(n/\log_\sigma n)$ bits of storage, where n is the length of the array T, and t is the number of distinct characters in T.

The wavelet tree is conceptually a binary tree (often a balanced tree) of logical bit arrays. A value of 0 (resp., 1) indicates that the corresponding entry is stored in one of the leaves of the left (resp., right) child. The collective size of the bit arrays at any given level of the tree is bounded by n, and they can be stored in compressed format, giving the 0th-order entropy space bound. When $\sigma = O(\operatorname{polylog} n)$, the height and traversal time of the wavelet tree can be made $O(1)$ by making the branching factor proportional to σ^ϵ for some $\epsilon > 0$ [21].

Binary wavelet trees have also been used to index an integer array $A[1\,..\,n]$ in linear space so as to efficiently support *position-restricted queries* [35,45]: given any index range $[\ell, r]$ and values x and y, we want to report all entries in $A[\ell\,..\,r]$ with values between x and y. We can traverse each level of the wavelet tree in constant time, so that the above query can be reported in $O(occ \log t)$ time, where *occ* denotes the number of the desired entries.

Wavelet tree also work in the external memory setting [35]. Instead of using a binary wavelet tree, we can increase the branching factor and obtain a B-ary (or $\sqrt{B}$-ary) wavelet tree so that each query is answered in $O(occ \log_B t)$ I/Os.

3.2 Compressed Text Indexes

Kärkkäinen [37] exploited Lempel-Ziv compression to develop a text index that, in addition to the text, used extra space proportional to the size of the text (later improved to $O(nH_k(T)) + o(n \log \sigma)$ bits). Query time was quadratic in p plus the time for p 2D range searches. Subsequent work focused on achieving faster query times of the form $O((p + occ)\operatorname{polylog} n)$, more in line with that provided by suffix trees and suffix arrays. In this section we focus on two parallel efforts — compressed suffix arrays and the FM-index — that achieve the desired goal.

Compressed Suffix Array (CSA). Grossi and Vitter [30,31] introduced the *compressed suffix array* (CSA), which settled the open problem of whether it was possible to simultaneously achieve fast query performance and break the $(n \log n)$-space barrier. In addition to the text, it used space proportional to the text size, specifically, $2n \log \sigma + O(n)$ bits, and answered queries in $O(p/\log_\sigma n + occ \log_\sigma n)$ time. The key idea was to store a sparse representation of the full suffix array, namely, the values that are multiples of 2^j for certain j. The *neighbor function* $\Phi(i) = SA^{-1}[SA[i] + 1]$ allows suffix array values to be computed on demand from the sparse representation in $O(\log_\sigma n)$ time.

Sadakane [61,62] showed how to make the CSA *self-indexing* by adding auxiliary data structures so that the Φ function was entire and defined for all i, which allowed the text values to be computed without need for storing the text T. Queries took $O((p + occ) \log n)$ time. Sadakane also introduced an entropy analysis, showing that its space was bounded by $nH_0(T) + O(n \log \log \sigma)$ bits.[1]

Grossi et al. [28] gave the first self-index that provably achieved asymptotic space optimality (i.e., with constant factor of 1 in the leading term). It used $nH_k(T)+o(n)$ bits and achieved $O(p \log \sigma + occ(\log^4 n)/((\log^2 \log n) \log \sigma))$ query time.[2] For $0 \leq \epsilon \leq 1/3$, there are various tradeoffs, such as $\frac{1}{\epsilon} nH_k(T) + o(n)$ bits of space and $O(p/\log_\sigma n + occ(\log^{2\epsilon/(1-\epsilon)} n) \log^{1-\epsilon} \sigma)$ query time. The Φ function is encoded by representing a character in terms of the contexts of its following k characters. For each character c in the text, the suffix array indices for the contexts following c form an increasing sequence. The CSA achieves high-order compression by encoding these increasing sequences in a context-by-context manner, using 0th-order statistics for each context. A wavelet tree is used to reduce redundancy in the sequence encodings.

FM-index. In parallel with the development of the CSA, Ferragina and Manzini introduced the elegant *FM-index* [19,20], based upon the *Burrows-Wheeler transform* (BWT) [7,50] data compressor. The FM-index was the first self-index shown to have both fast performance and space usage within a constant factor of the desired entropy bound for constant-sized alphabets. It used $5nH_k(T) + O(n^\epsilon \sigma^{\sigma+1} + n\sigma/\log n) + o(n)$ bits and handled queries in $O(p + occ \log^\epsilon n)$ time. The BWT of T is a permutation of T denoted by T_{bwt}, where $T_{\text{bwt}}[i]$ is the character in the text immediately preceding the ith lexicographically smallest suffix of T. That is, $T_{\text{bwt}}[i] = T[SA[i]-1]$. Intuitively, the sequence $T_{\text{bwt}}[i]$ is easy to compress because adjacent entries often share the same higher-order context. The "last to first" function LF is used to walk backwards through the text; $LF(i) = j$ if the ith lexicographically smallest suffix, when prepended with its preceding character, becomes the jth lexicographically smallest suffix.

The FM-index and the CSA are closely related: The LF function and the CSA neighbor function Φ are inverses. That is, $SA[LF(i)] = SA[i] - 1$; equivalently $LF(i) = SA^{-1}[SA[i] - 1] = \Phi^{-1}(i)$. A partition-based implementation

[1] We assume for convenience in this presentation that the alphabet size satisfies $\sigma = O(\text{polylog } n)$ so that the auxiliary data structures are negligible in size.

[2] We assume that $k \leq \alpha \log_\sigma n - 1$ for any constant $0 \leq \alpha \leq 1$, so that the kth-order model complexity is relatively small.

and analysis of the FM-index, similar to the context-based CSA space analysis described above [28], reduced the constant factor in the FM-index space bound to 1, achieving $nH_k(T) + o(n)$ bits and various query times, such as $O(p + occ \log^{1+\epsilon} n)$ [21,29]. Intuitively, the BWT T_{bwt} (and the CSA lists) can be partitioned into contiguous segments, where in each segment the context of subsequent text characters is the same. The context length may be fixed (say, k) or variable. We can code each segment of T_{bwt} (or CSA lists) using the statistics of character occurrences for that particular context. A particularly useful tool for encoding each segment is the wavelet tree, which reduces the coding problem from encoding vectors of characters to encoding bit vectors. Since each individual partition (context) is encoded by a separate wavelet tree using 0th-order compression, the net result is higher-order compression. This idea is behind the notion of "compression boosting" of Ferragina et al. [14].

Simpler implementations for the FM-index and CSA achieve higher-order compression without explicit partitioning into separate contexts. In fact, the original BWT was typically implemented by encoding T_{bwt} using the move-to-front heuristic [19,20]. Grossi et al. [24] proposed using a single wavelet tree to encode the entire T_{bwt} and CSA lists rather than a separate wavelet tree for each partition or context. Each wavelet tree node is encoded using run-length encoding, such as Elias's γ or δ codes [12]. (The γ code represents $i > 0$ with $2\lfloor \log i \rfloor + 1$ bits, and the δ code uses $\lfloor \log i \rfloor + 2\lfloor \log(\log i + 1) \rfloor + 1$ bits.) Most analyses of this simpler approach showed higher-order compression up to a constant factor [50,24,44,13]. The intuition is that encoding a run of length i by $O(\log i)$ bits automatically tunes itself to the statistics of the particular context.

Mäkinen and Navarro [46] showed how to use a single wavelet tree and achieve a space bound with a constant factor of 1, namely, $nH_k(T) + o(n)$ bits. They used a compressed block-based bit representation [59,57] to encode each bit array within the single wavelet tree. A similar bound can be derived if we instead encode each bit array using δ coding, enhanced with rank and select capabilities, as done by Sadakane [61,62]; however, the resulting space bound contains an additional additive term of $O(n \log H_k(T)) = O(n \log \log \sigma)$ bits, which arises from the $2 \log \log i$ term in δ encoding. This additive term increases the constant factor in the linear space term $nH_k(T)$ when the entropy or alphabet size is bounded by a constant, and under our assumptions on σ and k, it is bigger than the secondary $o(n)$ term. Mäkinen and Navarro [46] also apply their boosting technique to achieve high-order compression for dynamic text indexes [8,47].

Extensions. In recent years, compressed data structures has been a thriving field of research. The CSA and FM-index can be extended to support more complex queries, including dictionary matching [8], approximate matching [40], genome processing [22,41], XML subpath queries [18], multilabeled trees [3], and general suffix trees [63,60,23]. Puglisi et al. [58] showed that compressed text indexes provide faster searching than inverted indexes. However, they also showed that if the number of occurrences (matching locations) are too many, then inverted indexes perform better in terms of document retrieval. The survey

by Navarro and Mäkinen [55] also discusses index construction time and other developments, and Ferragina et al. [15] report experimental comparisons.

4 Geometric Burrows-Wheeler Transform (GBWT)

Range search is a useful tool in text indexing (see references in [9]). Chien et al. [9] propose two transformations that convert a set S of points $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ into text T, and vice-versa. These transformations show a two-way connectivity between problems in text indexing and orthogonal range search, the latter being a well-studied problem in the external memory setting and in terms of lower bounds. Let $\langle x \rangle$ be the binary encoding of x seen as a string, and let $\langle x \rangle^R$ be its reverse string. For each point (x_i, y_i) in S, the first transform constructs a string $\langle x_i \rangle^R \# \langle y_i \rangle \$$. The desired text T is formed by concatenating the above string for each point, so that $T = \langle x_1 \rangle^R \# \langle y_1 \rangle \$ \langle x_2 \rangle^R \# \langle y_2 \rangle \$ \ldots \langle x_n \rangle^R \# \langle y_n \rangle \$$. An orthogonal range query on S translates into $O(\log^2 n)$ pattern matching queries on T. This transformation provides a framework for translating (pointer machine as well as external memory) lower bounds known for range searching to the problem of compressed text indexing. An extended version of this transform, which maps 3D points into text, can be used to derive lower bounds for the position-restricted pattern matching problem.

For upper bounds, Chien et al. introduced the *geometric Burrows-Wheeler transform* (GBWT) to convert pattern matching problems into range queries. Given a text T and blocking factor d, let $T'[1 \ldots n/d]$ be the text formed by blocking every consecutive d characters of T to form a single metacharacter, as shown in Figure 1. Let $SA'[1 \ldots n/d]$ be the sparse suffix array of T'. The GBWT of T consists of the 2D points (i, c_i), for $1 \leq i \leq n/d$, where c_i is the reverse of the metacharacter that precedes $T'[SA'[i]]$. The parameter d is set to $\frac{1}{2} \log_\sigma n$ so that the data structures require only $O(n \log \sigma)$ bits.

To perform a pattern matching query for pattern P, we find, for each possible offset k between 0 and $d-1$, all occurrences of P that start k characters from the

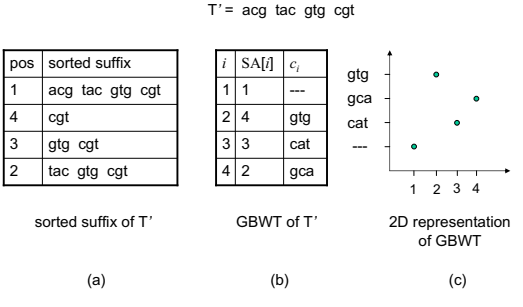


Fig. 1. Example of the GBWT for text $T =$ `acgtacgtgcgt`. The text of metacharacters is $T' =$ `acg tac gtg cgt`. (a) The suffixes of T' sorted into lexicographical order. (b) The suffix array SA' and the reverse preceding metacharacters c_i; the GBWT is the set of tuples (i, c_i), for all i. (c) The 2D representation of GBWT.

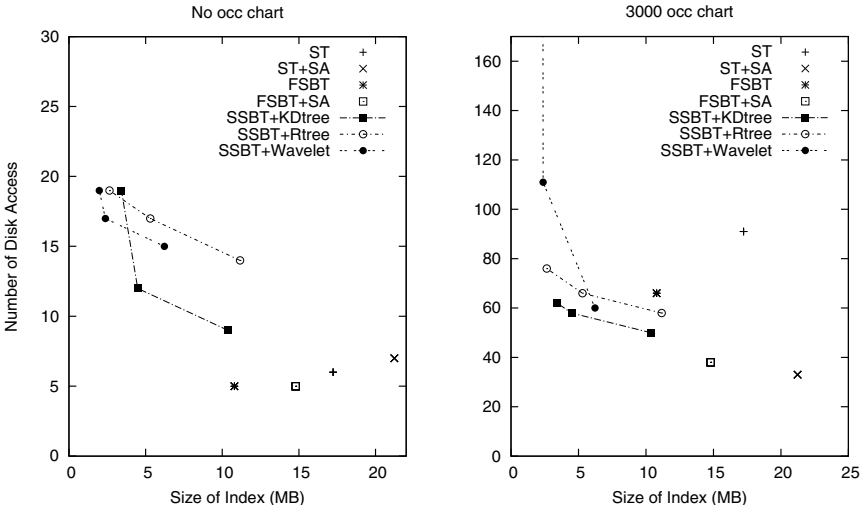


Fig. 2. I/Os per query. On the left, there is no output (i.e., the searches are unsuccessful). On the right, there are 3,000 occurrences on average per query.

beginning of a metacharacter. For $k \neq 0$, this process partitions P into $\langle \widehat{P}, \widetilde{P} \rangle$, where $\widetilde{P}$ matches a prefix of a suffix of T', and $\widehat{P}$ has length k and matches a suffix of the preceding metacharacter. By reversing $\widehat{P}$, both subcomponents must match prefixes, which corresponds to a 2D range query on the set S of 2D points defined above. The range of indices in the sparse suffix array SA' can be found by a string B-tree, and the 2D search can be done using a wavelet tree or using alternative indexes, such as kd-trees or R-trees.

If the pattern is small and fits entirely within a metacharacter, table lookup techniques (akin to inverted indexes) provide the desired answer using a negligible amount of space. The resulting space bound for GBWT is $O(n \log \sigma)$ bits, and the I/O query bound is the same as for four-sided 2D range search, namely, $O(p/B + (\log_\sigma n) \log_B n + occ \log_B n)$ or $O(p/B + \sqrt{n/B} \log_\sigma n + occ/B)$ [34]. Faster performance can often be achieved in practice using kd-trees or R-trees [10].

Hon et al. [34] introduce a variable-length sparsification so that each metacharacter corresponds to roughly d bits in *compressed* form. Assuming $k = o(\log_\sigma n)$, this compression further reduces the space usage from linear to $O(nH_k(T)+n)+o(n \log \sigma)$ bits of blocked storage. The query time for reporting pattern matches is $O(p/(B \log_\sigma n) + (\log^4 n)/\log \log n + occ \log_B n)$ I/Os.

4.1 Experimental Results for GBWT

In Figure 2, we compare the pattern matching performance of several indexes:

1. `ST`: Suffix tree (with naive blocking) and a parenthesis encoding of subtrees.
2. `ST + SA`: Suffix tree (with naive blocking strategy) and the suffix array.

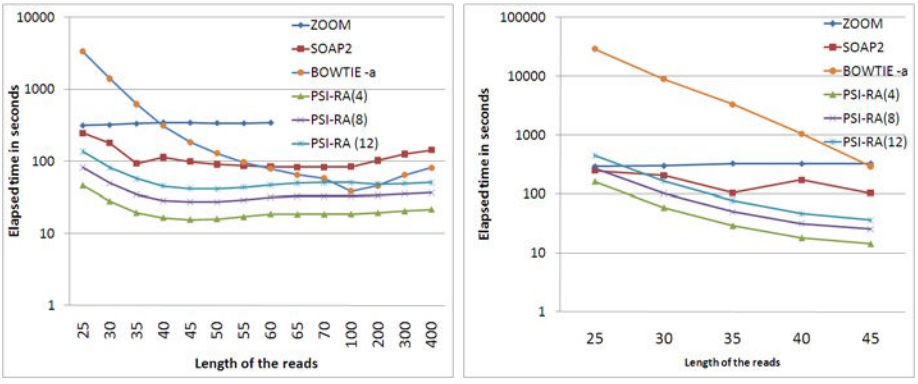


Fig. 3. Finding all exact matches of 1 million short read patterns P with the human genome. The left graph uses short read patterns sampled from the target genome; the right graph uses short read patterns obtained from the SRR001115 experiment [56].

3. `FSBT`: Full version of string B-tree containing all suffixes. The structure of each blind trie uses parentheses encoding, saving ≈ 1.75 bytes per trie node.
4. `FSBT + SA`: Full version of string B-tree and the suffix array.
5. `SSBT(d) + Rtree`: Sparse version of the string B-tree with the R-tree 2D range search data structure. Metacharacter sizes are $d = 2, 4, 8$.
6. `SSBT(d) +` k`d-tree`: Sparse version of the string B-tree with the kd-tree range search data structure. Metacharacter sizes are $d = 2, 4, 8$.
7. `SSBT(d) + Wavelet`: Sparse version of the string B-tree with the wavelet tree used for 2D queries. Metacharacter sizes are $d = 2, 4, 8$.

The first four indexes are not compressed data structures and exhibit significant space bloat; however, they achieve relatively good I/O performance. The latter three use sparsification, which slows query performance but requires less space.

4.2 Parallel Sparse Index for Genome Read Alignments

In this section, we consider the special case in the internal memory setting in which P is a "short read" that we seek to align with a genome sequence, such as the human genome. The human genome consists of about 3 billion bases (A, T, C, or G) and occupies roughly 800MB of raw space. In some applications, the read sequence P may be on the order of 30 bases, while with newer equipment, the length of P may be more than 100. We can simplify our GBWT approach by explicitly checking, for each match of $\widetilde{P}$, whether $\hat{P}$ also matches. We use some auxiliary data structures to quickly search the sparse suffix array SA' and employ a backtracking mechanism to find approximate matches. The reliability of each base in P typically degrades toward the end of P, and so our algorithm prioritizes mismatches toward the end of the sequence.

Figure 3 gives timings of short read aligners for a typical instance of the problem, in which all exact matches between each P and the genome are reported:

1. `SOAP2` [42]: Index size is 6.1 GB, based on 2way-BWT, run with parameters -r 1 -M 0 -v 0 (exact search).
2. `BOWTIE` [41]: Index size 2.9 GB, based upon BWT, run with -a option.
3. `ZOOM` [43]: No index, based on a multiple-spaced seed filtering technique, run with -mm 0 (exact search).
4. `Ψ-RA(4)`: Index size 3.4 GB, uses sparse suffix array with sparsification factor of $d = 4$ bases, finds all occurrences of the input patterns.
5. `Ψ-RA(8)`: Index size 2.0 GB, uses sparse suffix array with sparsification factor of $d = 8$ bases, finds all occurrences of the input patterns.
6. `Ψ-RA(12)`: Index size 1.6 GB, uses sparse suffix array with sparsification factor of $d = 12$ bases, finds all occurrences of the input patterns.

The size listed for each index includes the space for the original sequence data. Our simplified *parallel sparse index read aligner* (a.k.a. `Ψ-RA`) [39] achieves relatively high throughput compared with other methods. The experiments were performed on an Intel i7 with eight cores and 8GB memory. The `Ψ-RA` method can take advantage of multicore processors, since each of the d offset searches can be trivially parallelized. However, for fairness in comparisons, the timings used a single-threaded implementation and did not utilize multiple cores.

5 Top-k Queries for Relevance

Inverted indexes have several advantages over compressed data structures that need to be considered: (1) Inverted indexes are highly space-efficient, and they naturally provide the demarcation between RAM storage (dictionary of words) and disk storage (document lists for the words). (2) They are easy to construct in external memory. (3) They can be dynamically updated and also allow distributed operations [74]. (4) They can be easily tuned (by using frequency-ordered or PageRank-ordered lists) to retrieve top-k most relevant answers to the query, which is often required in search engines like Google.

Top-k query processing is an emerging field in databases [53,6,64,69,36]. When there are too many query results, certain notions of relevance may make some answers preferable to others. Database users typically want to see those answers first. In the problem of top-k document retrieval, the input data consist of D documents $\{d_1, d_2, \ldots, d_D\}$ of total length n. Given a query pattern P, the goal is to list which documents contain P; there is no need to report where in a document the matches occur. If a relevance measure is supplied (such as frequency of matches, proximity of matches, or PageRank), the goal is to output only the most relevant matching documents. The problem could specify an absolute threshold K on the relevance, in which case all matching documents are reported whose relevance value is $\geq K$; alternatively, given parameter k, the top-k most relevant documents are reported.

Early approaches to the problem did not consider relevance and instead reported all matches [53,64,69]. They used a generalized suffix tree, and for each leaf, they record which document it belongs to. On top of this basic data structure, early approaches employed either a chaining method to link together entries

from the same document or else a wavelet tree built over the document array. As a result, these data structures exhibit significant bloat in terms of space usage.

Hon et al. [36] employ a more space-conscious approach. They use a suffix tree, and every node of the suffix tree is augmented with additional arrays. A relevance queries can be seen as a $(2, 1, 1)$-query in 3D, where the two x-constraints come from specifying the subtree that matches the pattern P, the one-sided y-constraint is for preventing redundant output of the same document, and the one-sided z-constraint is to get the highest relevance scores. This $(2, 1, 1)$-query in 3D can be converted to at most p $(2, 1)$-queries in 2D, which in turn can be answered quickly using range-maximum query structures, thus achieving space-time optimal results. The result was the first $O(n)$-word index that takes $O(p + k \log k)$ time to answer top-k queries.

Preliminary experimental results show that for 2MB of input data, the index size is 30MB and can answer top-k queries in about 4×10^{-4} seconds (for $k = 10$). This implementation represents a major improvement because previous solutions, such as an adaptation of [53], take about 500MB of index size and are not as query-efficient. Further improvements are being explored.

Many challenging problems remain. One is to make the data structures compressed. The space usage is $\Omega(n)$ nodes, and thus $\Omega(n \log n)$ bits, which is larger than the input data. To reduce the space usage to that of a compressed representation, Hon et al. [36] employ sparsification to selectively augment only $O(n/\log^2 n)$ carefully chosen nodes of the suffix tree with additional information, achieving high-order compression, at the expense of slower search times. Other challenges include improved bounds and allowing approximate matching and approximate relevance. Thankachan et al. [67] develop top-k data structures for searching two patterns using $O(n)$ words of space with times related to 2D range search; the approach can be generalized for multipattern queries.

6 Conclusions

We discussed recent trends in compressed data structures for text and document indexing, with the goal of achieving the time and space efficiency of inverted indexes, but with greater functionality. We focused on two important challenging issues: I/O efficiency in external memory settings and building relevance into the query mechanism. Sparsification can help address both questions, and it can also be applied to the dual problem of dictionary matching, where the set of patterns is given and the query is the text [32,33,66,5]. Much work remains to be done, including addressing issues of parallel multicore optimization, dynamic updates, online data streaming, and approximate matching.

References

1. Aggarwal, A., Vitter, J.S.: The Input/Output complexity of sorting and related problems. Communications of the ACM 31(9), 1116–1127 (1988)
2. Arroyuelo, D., Navarro, G.: A Lempel-Ziv text index on secondary storage. In: Ma, B., Zhang, K. (eds.) CPM 2007. LNCS, vol. 4580, pp. 83–94. Springer, Heidelberg (2007)

3. Barbay, J., He, M., Munro, J.I., Rao, S.S.: Succinct indexes for strings, binary relations and multi-labeled trees. In: Proc. ACM-SIAM Symp. on Discrete Algorithms, pp. 680–689 (2007)
4. Bayer, R., Unterauer, K.: Prefix B-trees. ACM Transactions on Database Systems 2(1), 11–26 (1977)
5. Belazzougui, D.: Succinct dictionary matching with no slowdown. In: Proc. Symp. on Combinatorial Pattern Matching (June 2010)
6. Bialynicka-Birula, I., Grossi, R.: Rank-sensitive data structures. In: Consens, M.P., Navarro, G. (eds.) SPIRE 2005. LNCS, vol. 3772, pp. 79–90. Springer, Heidelberg (2005)
7. Burrows, M., Wheeler, D.: A block sorting data compression algorithm. Technical report, Digital Systems Research Center (1994)
8. Chan, H.L., Hon, W.K., Lam, T.W., Sadakane, K.: Compressed indexes for dynamic text collections. ACM Transactions on Algorithms 3(2) (2007)
9. Chien, Y.-F., Hon, W.-K., Shah, R., Vitter, J.S.: Geometric Burrows-Wheeler transform: Linking range searching and text indexing. In: Proc. IEEE Data Compression Conf., pp. 252–261 (2008)
10. Chiu, S.-Y., Hon, W.-K., Shah, R., Vitter, J.S.: I/O-efficient compressed text indexes: From theory to practice. In: Proc. IEEE Data Compression Conf., pp. 426–434 (2010)
11. Dean, J., Ghemawat, S.: MapReduce: Simplified data processing on large clusters. In: Proc. Symp. on Operating Systems Design and Implementation. December 2004, pp. 137–150, USENIX (2004)
12. Elias, P.: Universal codeword sets and representations of the integers. IEEE Transactions on Information Theory IT-21, 194–203 (1975)
13. Ferragina, P., Giancarlo, R., Manzini, G.: The myriad virtues of wavelet trees. Information and Computation 207(8), 849–866 (2009)
14. Ferragina, P., Giancarlo, R., Manzini, G., Sciortino, M.: Boosting textual compression in optimal linear time. Journal of the ACM 52(4), 688–713 (2005)
15. Ferragina, P., González, R., Navarro, G., Venturini, R.: Compressed text indexes: From theory to practice. ACM Journal of Experimental Algorithmics 12, article 1.12 (2008)
16. Ferragina, P., Grossi, R.: The String B-tree: A new data structure for string search in external memory and its applications. Journal of the ACM 46(2), 236–280 (1999)
17. Ferragina, P., Grossi, R., Gupta, A., Shah, R., Vitter, J.S.: On searching compressed string collections cache-obliviously. In: Proc. ACM Conf. on Principles of Database Systems, Vancouver, June 2008, pp. 181–190 (2008)
18. Ferragina, P., Luccio, F., Manzini, G., Muthukrishnan, S.: Structuring labeled trees for optimal succinctness, and beyond. In: Proc. IEEE Symp. on Foundations of Computer Science, pp. 184–196 (2005)
19. Ferragina, P., Manzini, G.: Opportunistic data structures with applications. In: Proc. IEEE Symp. on Foundations of Computer Science, November 2000, vol. 41, pp. 390–398 (2000)
20. Ferragina, P., Manzini, G.: Indexing compressed texts. Journal of the ACM 52(4), 552–581 (2005)
21. Ferragina, P., Manzini, G., Mäkinen, V., Navarro, G.: Compressed representations of sequences and full-text indexes. ACM Transactions on Algorithms 3(2) (May 2007) Conference version in SPIRE 2004
22. Ferragina, P., Venturini, R.: Compressed permuterm index. In: Proc. ACM SIGIR Conf. on Res. and Dev. in Information Retrieval, pp. 535–542 (2007)

23. Fischer, J., Mäkinen, V., Navarro, G.: Faster entropy-bounded compressed suffix trees. Theoretical Computer Science 410(51), 5354–5364 (2009)
24. Foschini, L., Grossi, R., Gupta, A., Vitter, J.S.: When indexing equals compression: Experiments on suffix arrays and trees. ACM Transactions on Algorithms 2(4), 611–639 (2004); Conference versions in SODA 2004 and DCC 2004
25. Frigo, M., Leiserson, C.E., Prokop, H., Ramachandran, S.: Cache-oblivious algorithms. In: Proc. IEEE Symp. on Foundations of Computer Science, vol. 40, pp. 285–298 (1999)
26. Gonnet, G.H., Baeza-Yates, R.A., Snider, T.: New indices for text: PAT trees and PAT arrays. In: Information Retrieval: Data Structures And Algorithms, ch. 5, pp. 66–82. Prentice-Hall, Englewood Cliffs (1992)
27. González, R., Navarro, G.: A compressed text index on secondary memory. In: Proc. Intl. Work. Combinatorial Algorithms, Newcastle, Australia, pp. 80–91. College Publications (2007)
28. Grossi, R., Gupta, A., Vitter, J.S.: High-order entropy-compressed text indexes. In: Proc. ACM-SIAM Symp. on Discrete Algorithms (January 2003)
29. Grossi, R., Gupta, A., Vitter, J.S.: Nearly tight bounds on the encoding length of the Burrows-Wheeler transform. In: Proc. Work. on Analytical Algorithmics and Combinatorics (January 2008)
30. Grossi, R., Vitter, J.S.: Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In: Proc. ACM Symp. on Theory of Computing, May 2000, vol. 32, pp. 397–406 (2000)
31. Grossi, R., Vitter, J.S.: Compressed suffix arrays and suffix trees with applications to text indexing and string matching. SIAM Journal on Computing 35(32), 378–407 (2005)
32. Hon, W.-K., Lam, T.-W., Shah, R., Tam, S.-L., Vitter, J.S.: Compressed index for dictionary matching. In: Proc. IEEE Data Compression Conf., pp. 23–32 (2008)
33. Hon, W.-K., Lam, T.-W., Shah, R., Tam, S.-L., Vitter, J.S.: Succinct index for dynamic dictionary matching. In: Dong, Y., Du, D.-Z., Ibarra, O. (eds.) ISAAC 2009. LNCS, vol. 5878. Springer, Heidelberg (2009)
34. Hon, W.-K., Shah, R., Thankachan, S.V., Vitter, J.S.: On entropy-compressed text indexing in external memory. In: Hyyro, H. (ed.) SPIRE 2009. LNCS, vol. 5721, pp. 75–89. Springer, Heidelberg (2009)
35. Hon, W.-K., Shah, R., Vitter, J.S.: Ordered pattern matching: Towards full-text retrieval. In: Purdue University Tech. Rept. (2006)
36. Hon, W.-K., Shah, R., Vitter, J.S.: Space-efficient framework for top-k string retrieval problems. In: Proc. IEEE Symp. on Foundations of Computer Science, Atlanta (October 2009)
37. Kärkkäinen, J.: Repetition-Based Text Indexes. Ph.d., University of Helsinki (1999)
38. Kärkkäinen, J., Rao, S.S.: Full-text indexes in external memory. In: Meyer, U., Sanders, P., Sibeyn, J. (eds.) Algorithms for Memory Hierarchies, ch. 7, pp. 149–170. Springer, Berlin (2003)
39. Külekci, M.O., Hon, W.-K., Shah, R., Vitter, J.S., Xu, B.: A parallel sparse index for read alignment on genomes (2010)
40. Lam, T.-W., Sung, W.-K., Wong, S.-S.: Improved approximate string matching using compressed suffix data structures. Algorithmica 51(3), 298–314 (2008)
41. Langmead, B., Trapnell, C., Pop, M., Salzberg, S.: Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. Genome Biology 10(3), article R25 (2009)

42. Li, R., Yu, C., Li, Y., Lam, T.-W., Yiu, S.-M., Kristiansen, K., Wang, J.: SOAP2: An improved ultrafast tool for short read alignment. Bioinformatics 25(15), 1966–1967 (2009)
43. Lin, H., Zhang, Z., Zhang, M.Q., Ma, B., Li, M.: ZOOM: Zillions of oligos mapped. Bioinformatics 24(21), 2431–2437 (2008)
44. Mäkinen, V., Navarro, G.: Succinct suffix arrays based on run-length encoding. Nordic Journal of Computing 12(1), 40–66 (2005)
45. Mäkinen, V., Navarro, G.: Position-restricted substring searching. In: Proc. Latin American Theoretical Informatics Symp., pp. 703–714 (2006)
46. Mäkinen, V., Navarro, G.: Implicit compression boosting with applications to self-indexing. In: Ziviani, N., Baeza-Yates, R. (eds.) SPIRE 2007. LNCS, vol. 4726, pp. 229–241. Springer, Heidelberg (2007)
47. Mäkinen, V., Navarro, G.: Dynamic entropy-compressed sequences and full-text indexes. ACM Transactions on Algorithms 4(3), article 12 (June 2008)
48. 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)
49. Manber, U., Myers, G.: Suffix arrays: A new method for on-line string searches. SIAM Journal on Computing 22(5), 935–948 (1993)
50. Manzini, G.: An analysis of the Burrows-Wheeler transform. Journal of the ACM 48(3) (2001); Conference version in SODA 1999
51. McCreight, E.M.: A space-economical suffix tree construction algorithm. Journal of the ACM 23(2), 262–272 (1976)
52. Moffat, A., Zobel, J.: Self-indexing inverted files for fast text retrieval. ACM Transactions on Information Systems 14(4), 349–379 (1996)
53. Muthukrishnan, S.: Efficient Algorithms for Document Retrieval Problems. In: Proc. ACM-SIAM Symp. on Discrete Algorithms, pp. 657–666 (2002)
54. Muthukrishnan, S.: Data Streams: Algorithms and Applications. Foundations and Trends in Theoretical Computer Science. now Publishers, Hanover (2005)
55. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Computing Surveys 39(1), article 2 (2007)
56. NCBI short read archive SRR001115, `http://www.ncbi.nlm.nih.gov/`
57. Patrascu, M.: Succincter. In: Proc. IEEE Symp. on Foundations of Computer Science, pp. 305–313 (2008)
58. 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)
59. Raman, R., Raman, V., Rao, S.S.: Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. ACM Transactions on Algorithms 3(4), article 43 (2007)
60. Russo, L., Navarro, G., Oliveira, A.: Fully-compressed suffix trees. In: Laber, E.S., Bornstein, C., Nogueira, L.T., Faria, L. (eds.) LATIN 2008. LNCS, vol. 4957, pp. 362–373. Springer, Heidelberg (2008)
61. Sadakane, K.: Compressed text databases with efficient query algorithms based on the compressed suffix array. In: Lee, D.T., Teng, S.-H. (eds.) ISAAC 2000, LNCS, vol. 1969, pp. 410–421. Springer, Heidelberg (December 2000)
62. Sadakane, K.: New text indexing functiionalities of the compressed suffix arrays. Journal of Algorithms 48(2), 294–313 (2003)
63. Sadakane, K.: Compressed suffix trees with full functionality. Theory of Computing Systems 41(4), 589–607 (2007)

64. Sadakane, K.: Succinct Data Structures for Flexible Text Retrieval Systems. Journal of Discrete Algorithms 5(1), 12–22 (2007)
65. Sodan, A.C., Machina, J., Deshmeh, A., Macnaughton, K., Esbaugh, B.: Parallelism via multithreaded and multicore CPUs. IEEE Computer 43(3), 24–32 (2010)
66. Tam, A., Wu, E., Lam, T.W., Yiu, S.-M.: Succinct text indexing with wildcards. In: Proc. Intl. Symp. on String Processing Information Retrieval, August 2009, pp. 39–50 (2009)
67. Thankachan, S.V., Hon, W.-K., Shah, R., Vitter, J.S.: String retrieval for multi-pattern queries (2010)
68. Ukkonen, E.: On-line construction of suffix trees. Algorithmica 14(3), 249–260 (1995)
69. Välimäki, N., Mäkinen, V.: Space-Efficient Algorithms for Document Retrieval. In: Ma, B., Zhang, K. (eds.) CPM 2007. LNCS, vol. 4580, pp. 205–215. Springer, Heidelberg (2007)
70. Vitter, J.S.: Algorithms and Data Structures for External Memory. Foundations and Trends in Theoretical Computer Science. now Publishers, Hanover (2008)
71. Vitter, J.S., Shriver, E.A.M.: Algorithms for parallel memory I: Two-level memories. Algorithmica 12(2–3), 110–147 (1994)
72. Weiner, P.: Linear pattern matching algorithm. In: Proc. IEEE Symp. on Switching and Automata Theory, Washington, DC, vol. 14, pp. 1–11 (1973)
73. Witten, I.H., Moffat, A., Bell, T.C.: Managing Gigabytes: Compressing and Indexing Documents and Images, 2nd edn. Morgan Kaufmann, Los Altos (1999)
74. Zobel, J., Moffat, A.: Inverted files for text search engines. ACM Computing Surveys 38(2) (2006)

Building the Minimal Automaton of A^*X in Linear Time, When X Is of Bounded Cardinality

Omar AitMous[1], Frédérique Bassino[1], and Cyril Nicaud[2]

[1] LIPN UMR 7030, Université Paris 13 - CNRS, 93430 Villetaneuse, France
[2] LIGM, UMR CNRS 8049, Université Paris-Est, 77454 Marne-la-Vallée, France
{aitmous,bassino}@lipn.univ-paris13.fr, nicaud@univ-mlv.fr

Abstract. We present an algorithm for constructing the minimal automaton recognizing A^*X, where the pattern X is a set of m (that is a fixed integer) non-empty words over a finite alphabet A whose sum of lengths is n. This algorithm, inspired by Brzozowski's minimization algorithm, uses sparse lists to achieve a linear time complexity with respect to n.

1 Introduction

This paper addresses the following issue: given a *pattern* X, that is to say a non-empty language which does not contain the empty word ε, and a text $T \in A^+$, assumed to be very long, how to efficiently find occurrences of words of X in the text T?

A usual approach is to precompute a deterministic automaton recognizing the language A^*X and use it to sequentially treat the text T. To find the occurrences of words of X, we simply read the text and move through the automaton. An occurrence of the pattern is found every time a final state is reached. Once built, this automaton can of course be used for other texts.

The pattern X can be of different natures, and we can reasonably consider three main categories: a single word, a finite set of words and a regular language. Depending on the nature of the pattern, the usual algorithms [6] build a deterministic automaton that is not necessary minimal.

For a single word u, very efficient algorithms such as the ones of Knuth, Morris and Pratt [10,6] or Boyer and Moore [4,6] are used. Knuth-Morris-Pratt algorithm simulates the minimal automaton recognizing A^*u. Aho-Corasick algorithm [1] treats finite sets of words by constructing a deterministic yet non-minimal automaton. And Mohri in [11] proposed an algorithm for regular languages given by a deterministic automaton.

In this article, we consider the case of a set of m non-empty words whose sum of lengths is n, where m is fixed and n tends toward infinity. Aho-Corasick algorithm [1] builds a deterministic automaton that recognizes A^*X with linear time and space complexities. Experimentally we remark, by generating uniformly at random patterns of m words whose sum of lengths is n, that the probability for Aho-Corasick automaton to be minimal is very small for large n. One can

A. Amir and L. Parida (Eds.): CPM 2010, LNCS 6129, pp. 275–287, 2010.

apply a minimization algorithm such as Hopcroft's algorithm [8] to Aho-Corasick automaton, but this operation costs an extra $\mathcal{O}(n \log n)$ time.

We propose another approach to directly build the minimal automaton of A^*X. It is based on Brzozowski's minimization algorithm described in [5]. This algorithm considers a non-deterministic automaton $\mathcal{A}$ recognizing a language $\mathcal{L}$, and computes the minimal automaton in two steps. First the automaton $\mathcal{A}$ is reversed and determinized. Second the resulting automaton is reversed and determinized too. Though the complexity of Brzozowski's algorithm is exponential in the worst case, our adaptation is linear in time and quadratic in space, using both automata constructions and an efficient implementation of sparse lists. The fact that the space complexity is greater than the time complexity is typical for that kind of sparse list implementation (see [3] for another such example, used to minimize local automata in linear time).

Outline of the paper: Our algorithm consists in replacing the first step of Brzozowski's algorithm by a direct construction of a co-deterministic automaton recognizing A^*X, and in changing the basic determinization algorithm into an ad hoc one using the specificity of the problem in the second step. With appropriate data structures, the overall time complexity is linear.

In Section 2 basic definitions and algorithms for words and automata are recalled. A construction of a co-deterministic automaton recognizing A^*X is described in Section 3. The specific determinization algorithm that achieves the construction of the minimal automaton is presented in Section 4. Section 5 present the way of using sparse lists and the analysis the global complexity of the construction.

2 Preliminary

In this section, the basic definitions and constructions used throughout this article are recalled. For more details, the reader is referred to [9] for automata and to [6,7] for algorithms on strings.

Automata. A *finite automaton* $\mathcal{A}$ over a finite alphabet A is a quintuple $\mathcal{A} = (A, Q, I, F, \delta)$, where Q is a finite set of *states*, $I \subset Q$ is the set of *initial states*, $F \subset Q$ is the set of *final states* and δ is a transition function from $Q \times A$ to $\mathcal{P}(Q)$, where $\mathcal{P}(Q)$ is the *power set* of Q. The automaton $\mathcal{A}$ is *deterministic* if it has only one initial state and if for any $(p, a) \in Q \times A$, $|\delta(q, a)| \leq 1$. It is *complete* if for any $(p, a) \in Q \times A$, $|\delta(q, a)| \geq 1$. A deterministic finite automaton $\mathcal{A}$ is *accessible* when for each state $q \in Q$, there exists a path from the initial state to q. The *size* of an automaton $\mathcal{A}$ is its number of states. The *minimal automaton* of a regular language is the unique smallest accessible and deterministic automaton recognizing this language.

The transition function δ is first extended to $\mathcal{P}(Q) \times A$ by $\delta(P, a) = \cup_{p \in P} \delta(p, a)$, then inductively to $\mathcal{P}(Q) \times A^*$ by $\delta(P, \varepsilon) = P$ and $\delta(P, w \cdot a) = \delta(\delta(P, w), a)$. A word u is recognized by $\mathcal{A}$ if there exists an initial state $i \in I$ such that $\delta(i, u) \cap F \neq \emptyset$. The set of words recognized by $\mathcal{A}$ is the language $\mathcal{L}(\mathcal{A})$.

The *reverse* of an automaton $\mathcal{A} = (A, Q, I, F, \delta)$ is the automaton ${}^t\mathcal{A} = (A, Q, F, I, {}^t\delta)$. For every $(p, q, a) \in Q \times Q \times A$, $p \in {}^t\delta(q, a)$ if and only if $q \in \delta(p, a)$. We denote by $\widetilde{w}$ the mirror word of w. The automaton ${}^t\mathcal{A}$ recognizes the language $\widetilde{L(\mathcal{A})} = \{\widetilde{w} \mid w \in \mathcal{L}(\mathcal{A})\}$. An automaton $\mathcal{A}$ is *co-deterministic* if its reverse automaton is deterministic.

Any finite automaton $\mathcal{A} = (A, Q, I, F, \delta)$ can be transformed by the *subset construction* into a deterministic automaton $\mathcal{B} = (A, \mathcal{P}(Q), \{I\}, F_{\mathcal{B}}, \delta_{\mathcal{B}})$ recognizing the same language and in which $F_{\mathcal{B}} = \{P \in \mathcal{P}(Q) \mid P \cap F \neq \emptyset\}$ and $\delta_{\mathcal{B}}$ is a function from $\mathcal{P}(Q) \times A$ to $\mathcal{P}(Q)$ defined by $\delta_{\mathcal{B}}(P, a) = \{q \in Q \mid \exists p \in P \text{ such that } q \in \delta(p, a)\}$. In the following we consider that the determinization of $\mathcal{A}$ only produces the accessible and complete part of $\mathcal{B}$.

Two complete deterministic finite automata $\mathcal{A} = (A, Q, i_0, F, \delta)$ and $\mathcal{A}' = (A, Q', i'_0, F', \delta')$ on the same alphabet are *isomorphic* when there exists a bijection ϕ from Q to Q' such that $\phi(i_0) = i'_0$, $\phi(F) = F'$ and for all $(q, a) \in Q \times A$, $\phi(\delta(q, a)) = \delta'(\phi(q), a)$. Two isomorphic automata only differ by the labels of their states.

Combinatorics on words. A word y is a *factor* of a word x if there exist two words u and v such that $x = u \cdot y \cdot v$. The word y is a *prefix* of x if $u = \varepsilon$; it is a *suffix* of x if $v = \varepsilon$. We say that y is a *proper* prefix (resp. suffix) of x if y is a prefix (resp. suffix) such that $y \neq \varepsilon$ and $y \neq x$.

A word y is called *a border* of x if $y \neq x$ and y is both a prefix and a suffix of x. *The* border of a non-empty word x denoted by $Border(x)$ is the longest of its borders. Note that any other border of x is a border of $Border(x)$. The set of all borders of x is $\{Border(x), Border(Border(x)), \ldots\}$.

In the following we note $x[i]$ the i-th letter of x, starting from position 0; the factor of x from position i to j is denoted by $x[i \cdots j]$. If $i > j$, $x[i \cdots j] = \varepsilon$.

To compute all borders of a word x of length ℓ, we construct the *border array* of x defined from $\{1, \ldots, \ell\}$ to $\{0, 1, \ldots, \ell-1\}$ by $border[i] = \mid Border(x[0 \cdots i-1]) \mid$. An efficient algorithm that constructs the border array is given in [6,7]. Its time and space complexities are $\Theta(|x|)$. It is based on the following formula that holds for any $x \in A^+$ and any $a \in A$

$$Border(x \cdot a) = \begin{cases} Border(x) \cdot a & \text{if } Border(x) \cdot a \text{ is a prefix of } x, \\ Border(Border(x) \cdot a) & \text{otherwise.} \end{cases} \quad (1)$$

3 A Co-deterministic Automaton Recognizing A^*X

In this section we give a direct construction of a co-deterministic automaton recognizing A^*X that can be interpreted as the first step of a Brzozowski-like algorithm.

Remark that if there exist two words $u, v \in X$ such that u is a suffix of v, one can remove the word v without changing the language, since $A^*v \subset A^*u$ and thus $A^*X = A^*(X \setminus \{v\})$. Hence, in the following we only consider finite suffix sets X, i.e. there are not two distinct words $u, v \in X$ such that u is a suffix of v.

Proposition 1. *Let X be a set of m non-empty words whose sum of lengths is n. There exists a deterministic automaton recognizing the language $\widetilde{X}A^*$ whose number of states is at most $n - m + 2$.*

Proof. (By construction) Let $\mathcal{A}$ be the automaton that recognizes $\widetilde{X}$, built directly from the tree of $\widetilde{X}$ by adding an initial state to the root and final states to the leaves. The states are labelled by the prefixes of $\widetilde{X}$. As we are basically interested in X, change every state label by its mirror, so that the states of the automaton are labelled by the suffixes of X. Merge all the final states into one new state labelled i, and add a loop on i for every letter in A. The resulting automaton is deterministic and recognizes the language $\widetilde{X}A^*$. □

The space and time complexities of this construction are linear in the length of X. This automaton is then reversed to obtain a co-deterministic automaton recognizing A^*X. For a given finite set of words X, we denote by $\mathcal{C}_X$ this co-deterministic automaton.

Example 1. Let $A = \{a, b\}$ be the alphabet and $X = \{aaa, abaa, abab\}$ be a set of $m = 3$ words whose sum of lengths is $n = 11$. The steps of the process are given in Figure 1.

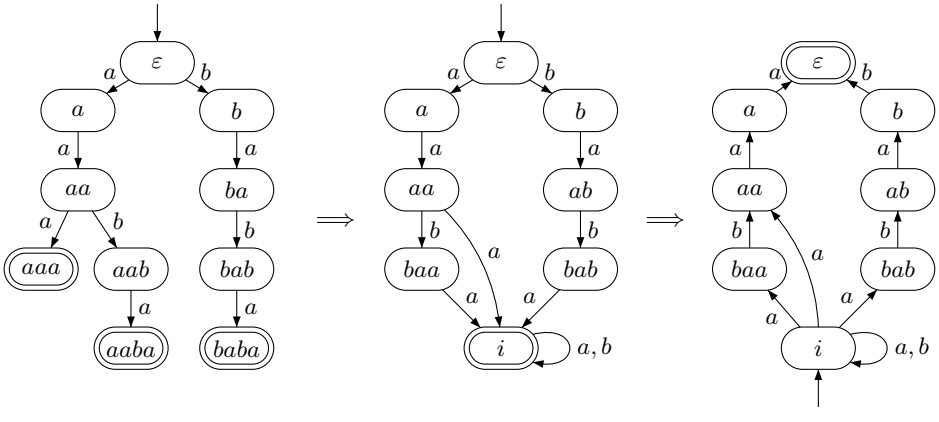


Fig. 1. Co-deterministic automaton $\mathcal{C}_X$ recognizing A^*X, where $X = \{aaa, abaa, abab\}$

4 Computing the Minimal Automaton

Once $\mathcal{C}_X$ is built, its determinization produces the minimal automaton recognizing the same language. It comes from the property used by Brzozowski's algorithm, namely that the determinization of a co-deterministic automaton gives the minimal automaton. According to Aho-Corasick algorithm this minimal automaton has at most $n + 1$ states.

It remains to efficiently handle sets of states in the determinization process. The subset construction produces the accessible part $\mathcal{B}$ of the automaton

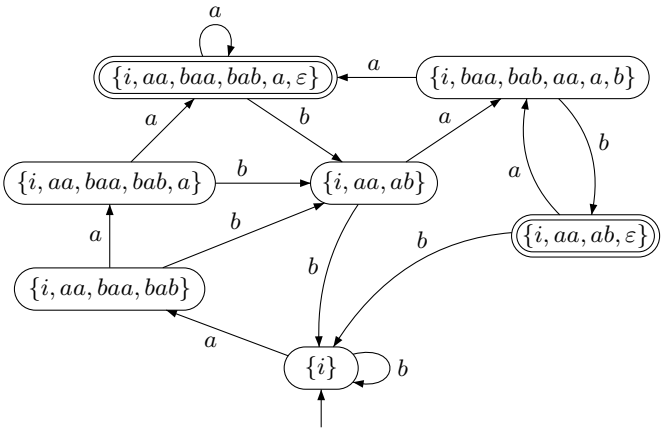


Fig. 2. Minimal automaton recognizing A^*X, with 7 states (by subset construction)

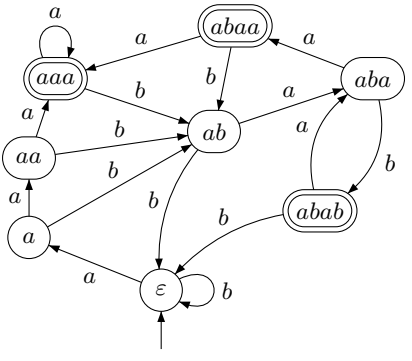


Fig. 3. Aho Corasick automaton recognizing A^*X, with 8 states

$(A, \mathcal{P}(Q), \{I\}, F_{\mathcal{B}}, \delta_{\mathcal{B}})$ from an automaton $\mathcal{A} = (A, Q, I, F, \delta)$. The states of $\mathcal{B}$ are labelled by subsets of Q.

Applied to the automaton of Figure 1 the subset construction leads to the minimal automaton depicted in Figure 2. Figure 3 shows Aho-Corasick automaton recognizing the same language A^*X where $X = \{aaa, abaa, abab\}$. The states are labelled by prefixes of words of X. This automaton is not minimal since the states aaa and $abaa$ are equivalent.

4.1 Cost of the Naive Subset Construction

When computing the subset construction, one has to handle sets of states: starting from the set of initial states, all the accessible states are built from the fly, using a depth-first traversal (for instance) of the result. At each step, given a set

of states P and a letter a, one has to compute the set $\delta_{\mathcal{B}}(P, a)$ and then check whether this state has already been built.

In the co-deterministic automaton $\mathcal{C}_X$, only the initial state i has non-deterministic transitions, and for every letter a, the image of i by a is of size at most $m + 1$, m being the number of words in X and one corresponding to the loop on the initial state i. Hence $\delta_{\mathcal{B}}(P, a)$ is of cardinality at most $m + 1 + |P|$ and is computed in time $\Theta(|P|)$, assuming that it is done using a loop through the elements of P. So even without taking into account the cost of checking whether $\delta_{\mathcal{B}}(P, a)$ has already been built, the time complexity of the determinization is $\Omega(\sum_P |P|)$, where the sum ranges over all P in the accessible part of the subset construction.

For the pattern $X = \{a^{n-1}, b\}$, the states of the result are $\{i\}$, $\{i, a^{n-2}\}$, $\{i, a^{n-2}, a^{n-3}\}$, ..., $\{i, a^{n-2}, a^{n-3}, \ldots, a\}$, $\{i, a^{n-2}, a^{n-3}, \ldots, a, \varepsilon\}$ and $\{i, \varepsilon\}$, so that $\sum_P |P| = \Omega(n^2)$. Therefore the time complexity of the naive subset construction is at least quadratic.

In the sequel, we present an alternative way to compute the determinization of $\mathcal{C}_X$ whose time complexity is linear.

4.2 Outline of the Construction

We make use of the following observations on $\mathcal{C}_X$. In the last automaton of Figure 1, when the state labelled b is reached, a word $u = v \cdot aba$ has been read, and the state bab has also been reached. This information can be obtained from the word b and the borders of prefixes of words in X: aba is a prefix of the word $x = abab \in X$, and $Border(aba) = a$. Our algorithm is based on limiting the length of the state labels of the minimal automaton by storing only one state per word of X, and one element to mark the state as final or not (ε or $\notin$). Hence if aba is read, only b is stored for the word $x = abab$.

When, for a letter $c \in A$, $\delta(b, c)$ is undefined, we jump to the state corresponding to the longest border of aba (the state bab in our example). We continue until either a transition we are looking for is found, or the unique initial state i is reached. More formally define the *failure* function f from $X \times Q \setminus \{i, \varepsilon\} \times A$ to $Q \setminus \{\varepsilon\}$ in the following way: $f(x, p, a)$ is the smallest suffix q of x, with $q \neq x$, satisfying:

- $x = up = vq$, v being a border of u
- $\delta(q, a)$ is defined.

If no such q exists, $f(x, p, a) = i$.

4.3 Precomputation of the Failure Function

Our failure function is similar to Aho-Corasick one in [1]. The difference is that ours is not based on suffixes but on borders of words. The value of $Border(v \cdot a)$ for every proper prefix v of a word $u \in X$ and every letter $a \in A$ is needed for the computation.

Table 1. Extended border array for $u = abab$, given $A = \{a, b\}$

	Prefix w of u with $w \neq u$			
Letter	ε	a	ab	aba
a	**0**	1	**1**	1
b	/	**0**	0	**2**

Extended border array. Let u be a word of length ℓ. We define an *extended* border array from $\{0, 1, \ldots, \ell-1\} \times A$ to $\{0, 1, \ldots, \ell-1\}$ by $border\_ext[0][u[0]] = 0$ and $border\_ext[i][a] = |Border(u[0 \cdot\cdot i-1] \cdot a)|$ for all $i \in \{1, \ldots, \ell-1\}$. Recall that $u[0 \cdot\cdot i]$ is the prefix of u of length $i+1$. Remark that $|Border(u[0 \cdot\cdot i])| = |Border(u[0 \cdot\cdot i-1] \cdot u[i])| = border\_ext[i-1][u[i]]$.

Table 1 depicts the extended border array of the word $abab$. Values computed by a usual border array algorithm are represented in bold.

Algorithm 1 (see Figure 4) computes the extended border array for a word u of length ℓ, considering the given alphabet A.

Standard propositions concerning the border array algorithm given in [7] are extended to Algorithm 1.

Proposition 2. *Extended_Borders algorithm above computes the extended border array of a given word u of length ℓ considering the alphabet A. Its space and time complexities are linear in the length of the word u.*

Proof. The routine Extended_Borders computes sequentially $|Border(v \cdot a)|$ for every proper prefix v of u and every letter $a \in A$. As the size of the alphabet

Algorithm 1. Extended_Borders

```
Inputs: u ∈ X, ℓ = |u|, alphabet A
 1  border_ext[0][u[0]] ← 0
 2  for j ← 1 to ℓ − 1 do
 3     for a ∈ A do
 4        i ← border_ext[j − 1][u[j − 1]]
 5        while i ≥ 0 and a ≠ u[i] do
 6           if i = 0 then
 7              i ← −1
 8           else
 9              i ← border_ext[i−1][u[i−1]]
10           end if
11        end while
12        i ← i + 1
13        border_ext[j][a] ← i
14     end for
15  end for
16  return border_ext
```

For every word $u \in X$ we compute its extended border array using the routine Extended_Borders. It contains for every proper prefix x of u and every letter $a \in A$, $|Border(x \cdot a)|$.

To compute $border\_ext[j][a] = |Border(u[0 \cdot\cdot j–1]) \cdot a|$, we need the length of $Border(u[0 \cdot\cdot j-1]) = Border(u[0 \cdot\cdot j-2] \cdot u[j-1])$. Thus $|Border(u[0 \cdot\cdot j-1])| = border\_ext[j-1][u[j-1]]$.

According to Equation (1), if $Border(u[0 \cdot\cdot i-1]) \cdot a$ is not a prefix of $u[0 \cdot\cdot i-1] \cdot a$, we need to find the longest border of the prefix of u of length i.
Since $Border(u[0 \cdot\cdot i-1]) = Border(u[0 \cdot\cdot i-2] \cdot u[i-1])$, we have $|Border(u[0 \cdot\cdot i-1])| = border\_ext[i-1][u[i-1]])$.

Fig. 4. Extended border array construction algorithm

A is considered to be constant, the space complexity of the construction is linear in ℓ.

A comparison between two letters a and b is said to be *positive* if $a = b$ and *negative* if $a \neq b$. The time complexity of the algorithm is linear in the number of letter comparisons. The algorithm computes, for $j = 1, \ldots, \ell - 1$ and $a \in A$, $border\_ext[j][a] = |Border(u[0 \cdot \cdot j - 1] \cdot a)|$. For a given letter $a \in A$, $border\_ext[j][a]$ is obtained from $|Border(u[0 \cdot \cdot j - 1])|$ that is already computed. The quantity $2j - i$ increases by at least one after each letter comparison: both i and j are incremented by one after a positive comparison; in the case of a negative comparison j is unchanged while i is decreased by at least one.

When $\ell = |u| \geq 2$, $2j - i$ is equal to 2 at the first comparison and $2\ell - 2$ at the last one. Thus the number of comparisons for a given letter a is at most $2\ell - 3$. The total time complexity is linear in the length of u. □

The extended border array can be represented by an automaton. Given a word $u \in A^+$, we construct the minimal automaton recognizing u. The states are labelled by prefixes of u. We then define a *border link* for all prefixes p of u and all letters $a \in A$ by:

$$BorderLink(p, a) = Border(p \cdot a)$$

that can be computed using Equation (1). This extended border array shows a strong similarity with the classical String Matching Automata (SMA) [6,7]. An adaptation of the SMA construction could be used as an alternative algorithm.

Example 2. Figure 5 shows this construction for the word $u = abab \in X$.

Failure function. The value $f(u, p, a)$ of the failure function is precomputed for every word $u \in X$, every proper suffix p of u and every letter $a \in A$ using Algorithm 2. The total time and space complexities of this operation are linear in the length of X. Remark that if $f(u, p, a) \neq i$ then $|\delta(f(u, p, a), a)| = 1$.

4.4 Determinization Algorithm

Let $X = \{u_1, \ldots, u_m\}$ be a set of m non-empty words whose sum of lengths is n and let $\mathcal{C}_X = (A, Q, \{i\}, \{\varepsilon\}, \delta)$ be the co-deterministic automaton recognizing the language A^*X obtained in Section 3. We denote by $\mathcal{B}_X$ the accessible part of the automaton $(A, I_{\mathcal{B}}, Q_{\mathcal{B}}, F_{\mathcal{B}}, \delta_{\mathcal{B}})$, where $Q_{\mathcal{B}} = (Q \setminus \{\varepsilon\})^m \times \{\varepsilon, \notin\}$,

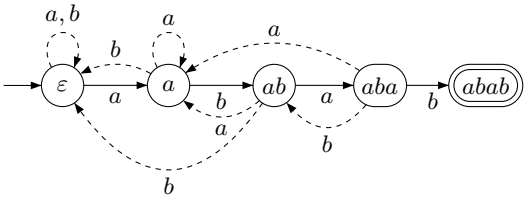


Fig. 5. Automaton $u = abab$ with border links

Algorithm 2. FAILURE_FUNCTION

Inputs: $u \in X$, p proper suffix of u, $a \in A$

```
1  if p[0] = a and |p| > 1 then
2     return p
3  end if
4  j ← border_ext[|u| − |p|][a]
5  if j ≤ 1 then
6     return i
7  end if
8  return u[j − 1 .. |u| − 1]
```

Let v be the prefix of u such that $u = v \cdot p$.
If $\delta(p, a)$ is defined and different than ε then $f(u, p, a) = p$.
If $|Border(v \cdot a)| = 0$ then $f(u, p, a) = i$, where i is the unique initial state of the co-deterministic automaton $\mathcal{A}$ recognizing A^*X (see Section 3).
If $|Border(v \cdot a)| \geq 1$ then $Border(v \cdot a) = w \cdot a$, with $w \in A^*$. If $w = \varepsilon$ then $f(u, p, a) = i$.
Otherwise, $f(u, p, a) = q$, with $Border(v \cdot a) = w_1 \cdot a$ and $u = w_1 \cdot q$.

Fig. 6. Failure function

$I_{\mathcal{B}} = \{(i, \ldots, i, \not\in)\}$ and for all $P \in F_{\mathcal{B}}$, $P = (v_1, v_2, \ldots, v_m, \varepsilon)$, where $v_r \in Q \backslash \{\varepsilon\}$ for all $r \in \{1, \ldots, m\}$. Given a state $P \in Q_{\mathcal{B}}$ and a letter $a \in A$ we use Algorithm 3 (see Figure 7) to compute $\delta_{\mathcal{B}}(P, a)$. Note that the automaton $\mathcal{B}_X$ is complete.

Theorem 1. *$\mathcal{B}_X$ is the minimal automaton recognizing A^*X.*

Proof. (Sketch) The idea of the proof is to show that $\mathcal{B}_X$ and the automaton produced by the classical subset construction are isomorphic.

Denote by $\mathcal{M} = (A, Q_{\mathcal{M}}, I_{\mathcal{M}}, F_{\mathcal{M}}, \delta_{\mathcal{M}})$ the minimal automaton built by the subset construction. Given a state $P \in Q_{\mathcal{B}}$ (resp. $P \in Q_{\mathcal{M}}$) and the smallest word v (the shortest one, and if there are several words of minimal length, we use the lexicographical order) such that $\delta_{\mathcal{B}}(I_{\mathcal{B}}, v) = P$ (resp. $\delta_{\mathcal{M}}(I_{\mathcal{M}}, v) = P$) we construct the unique corresponding state R in $\mathcal{M}$ (resp. in $\mathcal{B}_X$) using the same idea as in Section 4.2. Notice that i is in every state of $\mathcal{M}$. A word $s \in A^+$ is in R if there exist two words $x \in X$ and $u \in A^+$ such that $x = u \cdot s$ and either $u = v$ or u is a non-empty border of v. The state R is final and contains ε if and only if P is final. In the example of Figure 8 the word $v = aa$ is the smallest word such that $\delta_{\mathcal{B}}(I_{\mathcal{B}}, v) = P = (a, baa, bab, \not\in)$, and the corresponding state in $\mathcal{M}$ (see Figure 2) is $R = \{i, a, aa, baa, bab\}$. The minimality of the automaton is guaranteed by Brzozowski's construction [5]. □

Example 3. Algorithm 3 produces the automaton depicted in Figure 8 that is the minimal automaton recognizing A^*X, where $X = \{aaa, abaa, abab\}$.

5 Sparse Lists

In this section we present data structures and analyze the complexity of the construction of the minimal automaton. The co-deterministic automaton $\mathcal{C}_X$ of size at most $n - m + 2$ recognizing A^*X is built in time $\mathcal{O}(n)$, where X is a set

of m words whose sum of lengths is n. As stated before, the analysis is done for a fixed m, when n tends toward infinity. Minimizing $\mathcal{C}_X$ produces an automaton $\mathcal{B}_X$ whose number of states is linear in n and our determinization process creates only states labelled with sequences of $m+1$ elements. Sparse lists are used to encode these states.

Let $g : \{0, \ldots, \ell-1\} \to F$ be a partial function and denote by $Dom(g)$ the domain of g. A *sparse list* (see [2][Exercise 2.12 p.71] or [6][Exercise 1.15 p.55]) is a data structure that one can use to implement g and perform the following operations in constant time: initializing g with $Dom(g) = \emptyset$; setting a value $g(x)$ for a given $x \in \{0, \ldots, \ell-1\}$; testing whether $g(x)$ is defined or not; finding the value for $g(x)$ if it is defined; removing x from $Dom(g)$. The space complexity of a sparse list is $\mathcal{O}(\ell)$.

As we are interested in storing the states during the determinization, we illustrate here how to initialize, insert and test the existence of a value $g(x)$. To represent g, we use two arrays and a list (also represented as an array). The initialization consists in allocating these three arrays of size ℓ without initializing them. The number of elements in the list will be stored in an extra variable *size*. The values of the image by g are stored in the first array. The second array and the list are used to discriminate these values due to the random ones coming from the lack of initialization.

Algorithm 3. TRANSITION_FUNCTION

Inputs: $P = (v_1, v_2, \ldots v_m, j) \in Q_{\mathcal{B}}, a \in A$

```
1  j' ← ∉
2  for r ∈ {1...m} do
3     v'_r ← i
4     if δ(v_r, a) = ε then
5        j' ← ε
6     end if
7  end for
8  for ℓ = 1 to m do
9     v_ℓ ← f(u_ℓ, v_ℓ, a)
10    if v_ℓ ≠ i then
11       if v'_ℓ = i or |δ(v_ℓ, a)| < |v'_ℓ| then
12          v'_ℓ ← δ(v_ℓ, a)
13       end if
14    else
15       for r = 1 to s such that x_r ≠ ε do
16          if v'_t = i or |x_r| < |v'_t| then
17             v'_t ← x_r
18          end if
19       end for
20    end if
21 end for
22 return R = (v'_1, v'_2, ..., v'_m, j')
```

We initialize the first m elements of R to the unique initial state i in $\mathcal{A}$. The value of the last term of R is calculated (marking the state as final or non-final).

For each member v_ℓ we check the value of the failure function $f(u_\ell, v_\ell, a)$. If $f(u_\ell, v_\ell, a) \neq i$ then $|\delta(f(u_\ell, v_\ell, a), a)| = 1$ and we have found a potential value for v'_ℓ that is a suffix of $u_l \in X$. It remains to compare it to the already existing one and store the smallest in length different than i.

When the initial state i is reached, we are at the beginning of all the words in X. We define variables used in lines 15-17 as follows. From the definition of the automaton $\mathcal{A}$, $\delta(i, a) = \{x_1, x_2, \ldots, x_s\}$ where $0 \leq s \leq m$ and $a \cdot x_1 \in X, \ldots, a \cdot x_s \in X$. For every couple of integers $(r_1, r_2) \in \{1, \ldots, s\}^2$ such that $r_1 \neq r_2$, $a \cdot x_{r_1} \neq a \cdot x_{r_2}$. For all $r \in \{1, \ldots, s\}$ there exists a unique $t \in \{1, \ldots, m\}$ such that $a \cdot x_r = u_t \in X$.

Fig. 7. Transition function

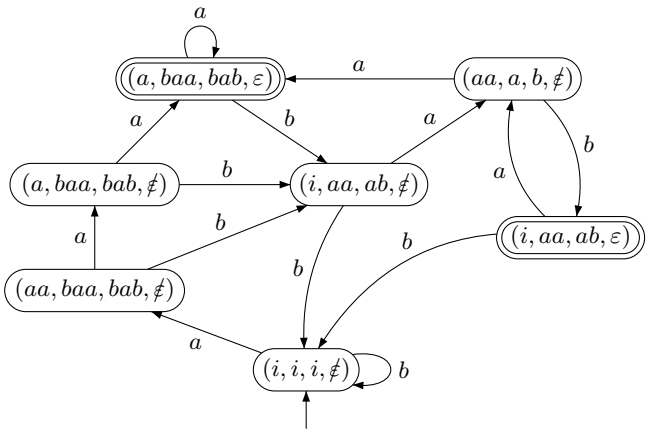


Fig. 8. Minimal automaton recognizing A^*X (by our construction), where $X = \{aaa, abaa, abab\}$

Figure 9 illustrates the sparse list initialization. Inserting an element $g(x) = y$ requires the following steps: $g[x] = y$; $index[x] = size$; $list[size] = x$ and $size = size + 1$. The result is shown in Figure 10. A value $g(x)$ is defined if and only if $index[x] < size$ and $list[index[x]] = x$.

Since the states we build are labelled by sequences of size $m + 1$, and each of the m first elements is either the initial state i of the automaton $\mathcal{C}_X$ or a proper suffix of the corresponding word in the pattern, we use a tree of sparse lists to store our states. Let $X = \{u_1, \ldots, u_m\}$ be the pattern and denote by $\mathrm{Suff}(u_r)$ the set of all proper suffixes of u_r for $1 \leq r \leq m$. We define a partial function g on $\{0, \ldots, |u_1| - 1\}$ whose values are partial functions $g(|v_1|)$ for $v_1 \in \mathrm{Suff}(u_1) \cup \{i\}$. We consider that $|i| = 0$. These functions $g(v_1)$ are defined on $\{0, \ldots, |u_2 - 1|\}$ and their values are again partial functions, denoted by $g(|v_1|, |v_2|)$ for $v_1 \in \mathrm{Suff}(u_1) \cup \{i\}$ and $v_2 \in \mathrm{Suff}(u_2) \cup \{i\}$. By extension we

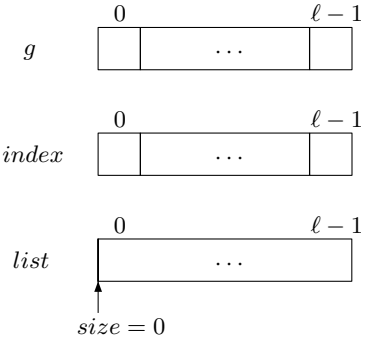


Fig. 9. Sparse list initialization

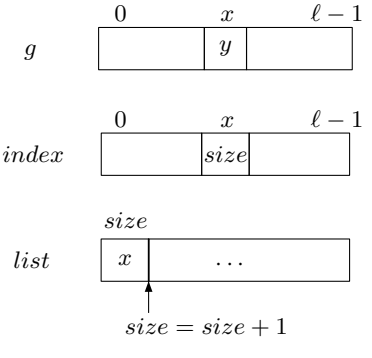


Fig. 10. Sparse list insertion of $g(x) = y$

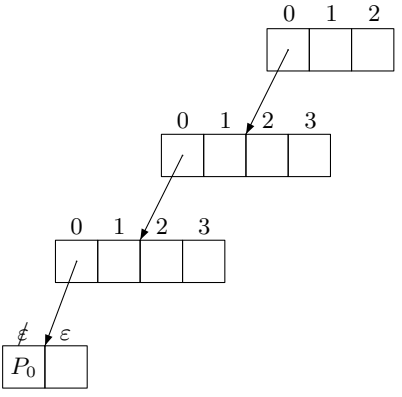


Fig. 11. Tree of sparse lists after inserting $P_0 = \{i, i, i, \not\in\}$

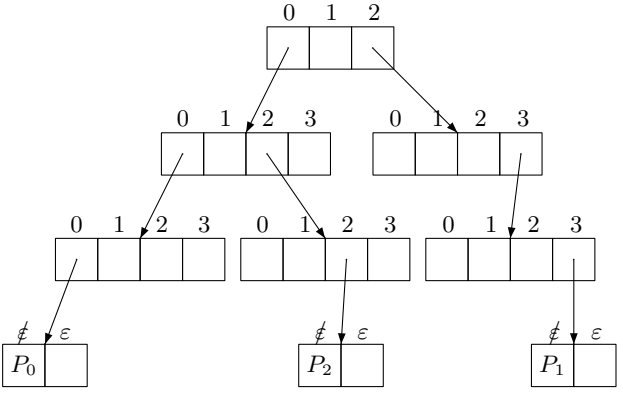


Fig. 12. Tree of sparse lists

build functions $g(|v_1|, |v_2|, \ldots, |v_m|) : \{0, \ldots, |u_1| - 1\} \times \{0, \ldots, |u_2| - 1\} \times \ldots \times \{0, \ldots, |u_m| - 1\} \times \{\varepsilon, \not\in\} \to Q_{\mathcal{B}}$ where $v_1 \in \text{Suff}(u_1) \cup \{i\}$, $v_2 \in \text{Suff}(u_2) \cup \{i\}$, $\ldots$, $v_m \in \text{Suff}(u_m) \cup \{i\}$ and $Q_{\mathcal{B}}$ is the set of states in the automaton $\mathcal{B}_X$. Then for a given state $P = (v_1, v_2, \ldots, v_m, j) \in Q_{\mathcal{B}}$, $g(|v_1|, |v_2|, \ldots, |v_m|, j) = P$.

When inserting a state $P = (v_1, v_2, \ldots, v_m, j)$ into our data structure, the existence of $g(|v_1|)$ is tested and, if it does not exist, a sparse list representing this partial function is created. Then the existence of $g(|v_1|, |v_2|)$ is checked. The same process is repeated until a function $g(|v_1|, |v_2|, \ldots, |v_m|)$ is found. Finally we test whether $g(|v_1|, |v_2|, \ldots, |v_m|, j)$ is defined, and if not the value for $g(|v_1|, |v_2|, \ldots, |v_m|, j)$ is set to P. If $g(|v_1|, |v_2|, \ldots, |v_m|, j)$ is already defined, then the state P already exists in the tree.

Figure 11 shows the insertion of the initial state $P_0 = (i, i, i, \not\in)$ for $X = \{aaa, abaa, abab\}$ and Figure 12 depicts the tree of sparse lists after inserting $P_0 = \{i, i, i, \not\in\}$, $P_1 = \{aa, baa, bab, \not\in\}$ and $P_2 = \{i, aa, ab, \not\in\}$.

Testing the existence of a state works in the same way, but if a partial function is not found then the state is not in the data structure.

Theorem 2. *Using sparse lists, the construction of the minimal automaton recognizing A^*X runs in time $\mathcal{O}(n)$ and requires $\mathcal{O}(n^2)$ space where n is the length of the pattern X.*

Proof. From Aho-Corasick's result the minimal automaton is of size at most $n+1$. As each state requires $m+1$ sparse lists of size $|u_1|, |u_2|, \ldots |u_m|, 2$, the total space complexity is quadratic in n. The time complexity of the determinization is linear in n since searching and inserting a state take $\mathcal{O}(1)$ time. □

Remark 1. In practice a hash table can be used to store these states. Under the hypothesis of a simple uniform hashing the average time and space complexities of the determinization are linear.

The natural continuation of this work is to investigate constructions based on Brzozowski's algorithm when m is not fixed anymore.

References

1. Aho, A.V., Corasick, M.J.: Efficient string matching: an aid to bibliographic search. Communications of the ACM 18(6), 333–340 (1975)
2. Aho, A.V., Hopcroft, J.E., Ullman, J.D.: The Design and Analysis of Computer Algorithms. Addison-Wesley, Reading (1974)
3. Béal, M.P., Crochemore, M.: Minimizing local automata. In: Caire, G., Fossorier, M. (eds.) IEEE International Symposium on Information Theory (ISIT 2007), 07CH37924C, pp. 1376–1380. IEEE Catalog (2007)
4. Boyer, R.S., Moore, J.S.: A fast string searching algorithm. Communications of the ACM 20(10), 62–72 (1977)
5. Brzozowski, J.A.: Canonical regular expressions and minimal state graphs for definite events. In: Mathematical theory of Automata. MRI Symposia Series, vol. 12, pp. 529–561. Polytechnic Press, Polytechnic Institute of Brooklyn, N.Y (1962)
6. Crochemore, M., Hancart, C., Lecroq, T.: Algorithms on Strings, 392 pages. Cambridge University Press, Cambridge (2007)
7. Crochemore, M., Rytter, W.: Jewels of Stringology. World Scientific Publishing Company, Singapore (2002)
8. Hopcroft, J.E.: An n log n algorithm for minimizing states in a finite automaton. In: Theory of Machines and computations, pp. 189–196. Academic Press, London (1971)
9. Hopcroft, J.E., Ullman, J.D.: Introduction To Automata Theory, Languages, And Computation. Addison-Wesley Longman Publishing Co., Inc., Boston (1990)
10. Knuth, D.E., Morris, J.H., Pratt, V.R.: Fast pattern matching in strings. SIAM Journal of Computing 6(2), 323–350 (1977)
11. Mohri, M.: String-matching with automata. Nordic Journal of Computing 4(2), 217–231 (Summer 1997)

A Compact Representation of Nondeterministic (Suffix) Automata for the Bit-Parallel Approach

Domenico Cantone, Simone Faro, and Emanuele Giaquinta

Università di Catania, Dipartimento di Matematica e Informatica
Viale Andrea Doria 6, I-95125 Catania, Italy
{cantone,faro,giaquinta}@dmi.unict.it

Abstract. We present a novel technique, suitable for bit-parallelism, for representing both the nondeterministic automaton and the nondeterministic suffix automaton of a given string in a more compact way. Our approach is based on a particular factorization of strings which on the average allows to pack in a machine word of w bits automata state configurations for strings of length greater than w. We adapted the Shift-And and BNDM algorithms using our encoding and compared them with the original algorithms. Experimental results show that the new variants are generally faster for long patterns.

1 Introduction

The string matching problem consists in finding all the occurrences of a pattern P of length m in a text T of length n, both defined over an alphabet Σ of size σ. The Knuth-Morris-Pratt (KMP) algorithm was the first linear-time solution (cf. [5]), whereas the Boyer-Moore (BM) algorithm provided the first sublinear solution on average [3]. Subsequently, the BDM algorithm reached the $\mathcal{O}(n \log_\sigma(m)/m)$ lower bound time complexity on the average (cf. [4]). Both the KMP and the BDM algorithms are based on finite automata; in particular, they respectively simulate a deterministic automaton for the language $\Sigma^* P$ and a deterministic suffix automaton for the language of the suffixes of P.

The bit-parallelism technique, introduced in [2], has been used to simulate efficiently the nondeterministic version of the KMP automaton. The resulting algorithm, named Shift-Or, runs in $\mathcal{O}(n\lceil m/w \rceil)$, where w is the number of bits in a computer word. Later, a variant of the Shift-Or algorithm, called Shift-And, and a very fast BDM-like algorithm (BNDM), based on the bit-parallel simulation of the nondeterministic suffix automaton, were presented in [6].

Bit-parallelism encoding requires one bit per pattern symbol, for a total of $\lceil m/w \rceil$ computer words. Thus, as long as a pattern fits in a computer word, bit-parallel algorithms are extremely fast, otherwise their performances degrades considerably as $\lceil m/w \rceil$ grows. Though there are a few techniques to maintain good performance in the case of long patterns, such limitation is intrinsic.

In this paper we present an alternative technique, still suitable for bit-parallelism, to encode both the nondeterministic automaton and the nondeterministic suffix automaton of a given string in a more compact way. Our encoding

A. Amir and L. Parida (Eds.): CPM 2010, LNCS 6129, pp. 288–298, 2010.

is based on factorizations of strings in which no character occurs more than once in any factor. This is the key towards separating the nondeterministic part from the determinstic one of the corresponding automata. It turns out that the nondeterministic part can be encoded with k bits, where k is the size of the factorization. Though in the worst case $k = m$, on the average k is much smaller than m, making it possible to encode large automata in a single or few computer words. As a consequence, bit-parallel algorithms based on such approach tend to be faster in the case of sufficiently long patterns. We will illustrate this point by comparing experimentally different implementations of the Shift-And and the BNDM algorithms.

2 Basic Notions and Definitions

Given a finite alphabet Σ, we denote by Σ^m, with $m \geq 0$, the collection of strings of length m over Σ and put $\Sigma^* = \bigcup_{m \in \mathbb{N}} \Sigma^m$. We represent a string $P \in \Sigma^m$, also called an m-gram, as an array $P[0\,..\,m-1]$ of characters of Σ and write $|P| = m$ (in particular, for $m = 0$ we obtain the empty string ε). Thus, $P[i]$ is the $(i+1)$-st character of P, for $0 \leqslant i < m$, and $P[i\,..\,j]$ is the substring of P contained between its $(i+1)$-st and the $(j+1)$-st characters, for $0 \leqslant i \leqslant j < m$. Also, we put $\mathit{first}(P) = P[0]$ and $\mathit{last}(P) = P[|P|-1]$. For any two strings P and P', we say that P' is a suffix of P if $P' = P[i\,..\,m-1]$, for some $0 \leqslant i < m$, and write $\mathit{Suff}(P)$ for the set of all suffixes of P. Similarly, P' is a prefix of P if $P' = P[0\,..\,i]$, for some $0 \leqslant i < m$. In addition, we write $P.P'$, or more simply PP', for the concatenation of P and P', and P^r for the reverse of the string P, i.e. $P^r = P[m-1]P[m-2]\ldots P[0]$.

Given a string $P \in \Sigma^m$, we indicate with $\mathcal{A}(P) = (Q, \Sigma, \delta, q_0, F)$ the nondeterministic automaton for the language $\Sigma^* P$ of all words in Σ^* ending with an occurrence of P, where:

- $Q = \{q_0, q_1, \ldots, q_m\}$ (q_0 is the initial state)
- the transition function $\delta : Q \times \Sigma \longrightarrow \mathscr{P}(Q)$ is defined by:

$$\delta(q_i, c) =_{\text{Def}} \begin{cases} \{q_0, q_1\} & \text{if } i = 0 \text{ and } c = P[0] \\ \{q_0\} & \text{if } i = 0 \text{ and } c \neq P[0] \\ \{q_{i+1}\} & \text{if } 1 \leqslant i < m \text{ and } c = P[i] \\ \emptyset & \text{otherwise} \end{cases}$$

- $F = \{q_m\}$ (F is the set of final states).

Likewise, for a string $P \in \Sigma^m$, we denote by $\mathcal{S}(P) = (Q, \Sigma, \delta, I, F)$ the nondeterministic suffix automaton with ε-transitions for the language $\mathit{Suff}(P)$ of the suffixes of P, where:

- $Q = \{I, q_0, q_1, \ldots, q_m\}$ (I is the initial state)
- the transition function $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \longrightarrow \mathscr{P}(Q)$ is defined by:

$$\delta(q, c) =_{\text{Def}} \begin{cases} \{q_{i+1}\} & \text{if } q = q_i \text{ and } c = P[i] \quad (0 \leqslant i < m) \\ \{q_0, q_1, \ldots, q_m\} & \text{if } q = I \text{ and } c = \varepsilon \\ \emptyset & \text{otherwise} \end{cases}$$

– $F = \{q_m\}$ (F is the set of final states).

The valid configurations $\delta^*(q_0, S)$ reachable by the automata $\mathcal{A}(P)$ on input $S \in \Sigma^*$ are defined recursively as follows:

$$\delta^*(q_0, S) =_{Def} \begin{cases} \{q\} & \text{if } S = \varepsilon, \\ \bigcup_{q' \in \delta(q_0, S')} \delta^*(q', c) & \text{if } S = S'c, \text{ for some } c \in \Sigma \text{ and } S' \in \Sigma^*. \end{cases}$$

Much the same definition of reachable configurations holds for the automata $\mathcal{S}(P)$, but in this case one has to use $\delta(I, \varepsilon) = \{q_0, q_1, \ldots, q_m\}$ for the base case.

Finally, we recall the notation of some bitwise infix operators on computer words, namely the bitwise `and` "&", the bitwise `or` "|", the `left shift` "$\ll$" operator (which shifts to the left its first argument by a number of bits equal to its second argument), and the unary bitwise `not` operator "$\sim$".

3 The Bit-Parallelism Technique

Bit-parallelism is a technique introduced by Baeza-Yates and Gonnet in [2] that takes advantage of the intrinsic parallelism of the bit operations inside a computer word, allowing to cut down the number of operations that an algorithm performs by a factor up to w, where w is the number of bits in the computer word. Bit-parallelism is particularly suitable for the efficient simulation of nondeterministic (suffix) automata; the first algorithms based on it are the well-known Shift-And [2] and BNDM [6]. The Shift-And algorithm simulates the nondeterministic automaton (NFA, for short) that recognizes the language $\Sigma^* P$, for a given string P of length m. Its bit-parallel representation uses an array B of $|\Sigma|$ bit-vectors, each of size m, where the i-th bit of $B[c]$ is set iff $\delta(q_i, c) = q_{i+1}$ or equivalently iff $P[i] = c$, for $c \in \Sigma$, $0 \leqslant i < m$. Automaton configurations $\delta^*(q_0, S)$ on input $S \in \Sigma^*$ are then encoded as a bit-vector D of m bits (the initial state does not need to be represented, as it is always active), where the i-th bit of D is set iff state q_{i+1} is active, i.e. $q_{i+1} \in \delta^*(q_0, S)$, for $i = 0, \ldots, m-1$. For a configuration D of the NFA, a transition on character c can then be implemented by the bitwise operations

$$D \leftarrow ((D \ll 1) \mid 1) \;\&\; B[c]\,.$$

The bitwise or with 1 (represented as $0^{m-1}1$) is performed to take into account the self-loop labeled with all the characters in Σ on the initial state. When a search starts, the initial configuration D is initialized to 0^m. Then, while the text is read from left to right, the automaton configuration is updated for each text character, as described before.

The nondeterministic suffix automaton for a given string P is an NFA with ε-transitions that recognizes the language $Suff(P)$. The BNDM algorithm simulates the suffix automaton for P^r with the bit-parallelism technique, using an encoding similar to the one described before for the Shift-And algorithm. The i-th bit of D is set iff state q_{i+1} is active, for $i = 0, 1, \ldots, m-1$, and D is initialized

to 1^m, since after the ε-closure of the initial state I all states q_i represented in D are active. The first transition on character c is implemented as $D \leftarrow (D \,\&\, B[c])$, while any subsequent transition on character c can be implemented as

$$D \leftarrow ((D \ll 1) \,\&\, B[c])\,.$$

The BNDM algorithm works by shifting a window of length m over the text. Specifically, for each window alignment, it searches the pattern by scanning the current window backwards and updating the automaton configuration accordingly. Each time a suffix of P^r (i.e., a prefix of P) is found, namely when prior to the left shift the m-th bit of $D\&B[c]$ is set, the window position is recorded. A search ends when either D becomes zero (i.e., when no further prefixes of P can be found) or the algorithm has performed m iterations (i.e., when a match has been found). The window is then shifted to the start position of the longest recognized proper prefix.

When the pattern size m is larger than w, the configuration bit-vector and all auxiliary bit-vectors need to be splitted over $\lceil m/w \rceil$ multiple words. For this reason the performance of the Shift-And and BNDM algorithms, and of bit-parallel algorithms more in general, degrades considerably as $\lceil m/w \rceil$ grows. A common approach to overcome this problem consists in constructing an automaton for a substring of the pattern fitting in a single computer word, to filter possible candidate occurrences of the pattern. When an occurrence of the selected substring is found, a subsequent naive verification phase allows to establish whether this belongs to an occurrence of the whole pattern. However, besides the costs of the additional verification phase, a drawback of this approach is that, in the case of the BNDM algorithm, the maximum possible shift length cannot exceed w, which could be much smaller than m.

In the next section we illustrate an alternative encoding for automata configurations, which in general requires less than one bit per pattern character and still is suitable for bit-parallelism.

4 Tighter Packing for Bit-Parallelism

We present a new encoding of the configurations of the nondeterministic (suffix) automaton for a given pattern P of length m, which on the average requires less than m bits and is still suitable to be used within the bit-parallel framework. The effect is that bit-parallel string matching algorithms based on such encoding scale much better as m grows, at the price of a larger space complexity. We will illustrate such point experimentally with the Shift-And and the BNDM algorithms, but our proposed encoding can also be applied to other variants of the BNDM algorithm as well.

Our encoding will have the form (D, a), where D is a k-bit vector, with $k \leqslant m$ (on the average k is much smaller than m), and a is an alphabet symbol (the last text character read) which will be used as a parameter in the bit-parallel simulation with the vector D.

The encoding (D, a) is obtained by suitably factorizing the simple bit-vector encoding for NFA configurations presented in the previous section. More specifically, it is based on the following pattern factorization.

Definition (1-factorization)**.** *Let $P \in \Sigma^m$. A 1-factorization of size k of P is a sequence $\langle u_1, u_2, \ldots, u_k \rangle$ of nonempty substrings of P such that:*

(a) $P = u_1 u_2 \ldots u_k$;
(b) each factor u_j contains at most one *occurrence for any of the characters in the alphabet Σ, for $j = 1, \ldots, k$.*

A 1-factorization of P is minimal *if such is its size.*

Remark. It can easily be checked that a 1-factorization $\langle u_1, u_2, \ldots, u_k \rangle$ of P is minimal if $first(u_{i+1})$ occurs in u_i, for $i = 1, \ldots, k-1$.

Observe, also, that $\lceil \frac{m}{\sigma} \rceil \leqslant k \leqslant m$ holds, for any 1-factorization of size k of a string $P \in \Sigma^m$, where $\sigma = |\Sigma|$. The worst case occurs when $P = a^m$, in which case P has only the 1-factorization of size m whose factors are all equal to the single character string a.

A 1-factorization $\langle u_1, u_2, \ldots, u_k \rangle$ of a given pattern $P \in \Sigma^*$ induces naturally a partition $\{Q_1, \ldots, Q_k\}$ of the set $Q \setminus \{q_0\}$ of nonstarting states of the canonical automaton $\mathcal{A}(P) = (Q, \Sigma, \delta, q_0, F)$ for the language $\Sigma^* P$, where

$$Q_i =_{\text{Def}} \left\{ q_{\sum_{j=1}^{i-1} |u_j|+1}, \ldots, q_{\sum_{j=1}^{i} |u_j|} \right\}, \text{ for } i = 1, \ldots, k\,.$$

Notice that the labels of the arrows entering the states

$$q_{\sum_{j=1}^{i-1} |u_j|+1}, \ldots, q_{\sum_{j=1}^{i} |u_j|}\,,$$

in that order, form exactly the factor u_i, for $i = 1, \ldots, k$. Hence, if for any alphabet symbol a we denote by $Q_{i,a}$ the collection of states in Q_i with an incoming arrow labeled a, it follows that $|Q_{i,a}| \leqslant 1$, since by condition (b) of the above definition of 1-factorization no two states in Q_i can have an incoming transition labeled by a same character. When $Q_{i,a}$ is nonempty, we write $q_{i,a}$ to indicate the unique state q of $\mathcal{A}(P)$ for which $q \in Q_{i,a}$, otherwise $q_{i,a}$ is undefined. On using $q_{i,a}$ in any expression, we will also implicitly assert that $q_{i,a}$ is defined.

For any valid configuration $\delta^*(q_0, Sa)$ of the automaton $\mathcal{A}(P)$ on some input of the form $Sa \in \Sigma^*$, we have that $q \in \delta^*(q_0, Sa)$ only if the state q has an incoming transition labeled a. Therefore, $Q_i \cap \delta^*(q_0, Sa) \subseteq Q_{i,a}$ and, consequently, $|Q_i \cap \delta^*(q_0, Sa)| \leqslant 1$, for each $i = 1, \ldots, k$. The configuration $\delta^*(q_0, Sa)$ can then be encoded by the pair (D, a), where D is the bit-vector of size k such that $D[i]$ is set iff Q_i contains an active state, i.e., $Q_i \cap \delta^*(q_0, Sa) \neq \emptyset$, iff $q_{i,a} \in \delta^*(q_0, Sa)$. Indeed, if $i_1, i_2, \ldots, i_l$ are all the indices i for which $D[i]$ is set, we have that $\delta^*(q_0, Sa) = \{q_{i_1,a}, q_{i_2,a}, \ldots, q_{i_l,a}\}$ holds, showing that the above encoding (D, a) can be inverted.

To show how to compute D' in a transition $(D, a) \xrightarrow{\mathcal{A}} (D', c)$ on character c using bit-parallelism, it is convenient to give some further definitions.

```
F-PREPROCESS (P, m)

for c ∈ Σ do S[c] ← L[c] ← 0
for c, c' ∈ Σ do B[c][c'] ← 0
b ← 0, e ← 0, k ← 0
while e < m do
    while e < m and S[P[e]] = 0 do
        S[P[e]] ← 1, e ← e + 1
    for i ← b to e − 1 do S[P[i]] ← 0
    for i ← b + 1 to e − 1 do
        B[P[i − 1]][P[i]] ← B[P[i − 1]][P[i]] | (1 ≪ k)
    L[P[e − 1]] ← L[P[e − 1]] | (1 ≪ k)
    if e < m then
        B[P[e − 1]][P[e]] ← B[P[e − 1]][P[e]] | (1 ≪ k)
    b ← e
    k ← k + 1
M ← (1 ≪ (k − 1))
return (B, L, M, k)
```

Fig. 1. Preprocessing procedure for the construction of the arrays B and L relative to a minimal 1-factorization of the pattern

For $i = 1, \ldots, k-1$, we put $\overline{u}_i = u_i.first(u_{i+1})$. We also put $\overline{u}_k = u_k$ and call each set $\overline{u}_i$ the *closure* of u_i.

Plainly, any 2-gram can occur at most once in the closure $\overline{u}_i$ of any factor of our 1-factorization $\langle u_1, u_2, \ldots, u_k \rangle$ of P. We can therefore encode the 2-grams present in the closure of the factors u_i by a $|\Sigma| \times |\Sigma|$ matrix B of k-bit vectors, where the i-th bit of $B[c_1][c_2]$ is set iff the 2-gram c_1c_2 is present in $\overline{u}_i$ or, equivalently, iff

$$\begin{aligned}(last(u_i) \neq c_1 \wedge q_{i,c_2} \in \delta(q_{i,c_1}, c_2)) \vee \\ (i < k \wedge last(u_i) = c_1 \wedge q_{i+1,c_2} \in \delta(q_{i,c_1}, c_2))\,,\end{aligned} \tag{1}$$

for every 2-gram $c_1c_2 \in \Sigma^2$ and $i = 1, \ldots, k$.

To properly take care of transitions from the last state in Q_i to the first state in Q_{i+1}, it is also useful to have an array L, of size $|\Sigma|$, of k-bit vectors encoding for each character $c \in \Sigma$ the collection of factors ending with c. More precisely, the i-th bit of $L[c]$ is set iff $last(u_i) = c$, for $i = 1, \ldots, k$.

We show next that the matrix B and the array L, which in total require $(|\Sigma|^2 + |\Sigma|)k$ bits, are all is needed to compute the transition $(D, a) \xrightarrow{\mathcal{A}} (D', c)$ on character c. To this purpose, we first state the following basic property, which can easily be proved by induction.

Transition Lemma. *Let $(D, a) \xrightarrow{\mathcal{A}} (D', c)$, where (D, a) is the encoding of the configuration $\delta^*(q_0, Sa)$ for some string $S \in \Sigma^*$, so that (D', c) is the encoding of the configuration $\delta^*(q_0, Sac)$.*

Then, for each $i = 1, \ldots, k$, $q_{i,c} \in \delta^*(q_0, Sac)$ if and only if either

(i) $last(u_i) \neq a$, $q_{i,a} \in \delta^*(q_0, Sa)$, and $q_{i,c} \in \delta(q_{i,a}, c)$, or
(ii) $i \geqslant 1$, $last(u_{i-1}) = a$, $q_{i-1,a} \in \delta^*(q_0, Sa)$, and $q_{i,c} \in \delta(q_{i-1,a}, c)$. □

Now observe that, by definition, the i-th bit of D' is set iff $q_{i,c} \in \delta^*(q_0, Sac)$ or, equivalently by the Transition Lemma and (1), iff (for $i = 1, \ldots, k$)

$$
\begin{aligned}
&(D[i] = 1 \wedge B[a][c][i] = 1 \wedge \sim L[a][i] = 1) \vee \\
&\qquad (i \geqslant 1 \wedge D[i-1] = 1 \wedge B[a][c][i-1] = 1 \wedge L[a][i-1] = 1) && \text{iff} \\
&((D \;\&\; B[a][c] \;\&\; \sim L[a])[i] = 1 \vee (i \geqslant 1 \wedge (D \;\&\; B[a][c] \;\&\; L[a])[i-1] = 1)) && \text{iff} \\
&((D \;\&\; B[a][c] \;\&\; \sim L[a])[i] = 1 \vee ((D \;\&\; B[a][c] \;\&\; L[a]) \ll 1)[i] = 1) && \text{iff} \\
&((D \;\&\; B[a][c] \;\&\; \sim L[a]) \mid ((D \;\&\; B[a][c] \;\&\; L[a]) \ll 1))[i] = 1\,.
\end{aligned}
$$

Hence $D' = (D \;\&\; B[a][c] \;\&\; \sim L[a]) \mid ((D \;\&\; B[a][c] \;\&\; L[a]) \ll 1)$, so that D' can be computed by the following bitwise operations:

$$
\begin{aligned}
&D \leftarrow D \;\&\; B[a][c] \\
&H \leftarrow D \;\&\; L[a] \\
&D \leftarrow (D \;\&\; \sim H) | (H \ll 1)\,.
\end{aligned}
$$

To check whether the final state q_m belongs to a configuration encoded as (D, a), we have only to verify that $q_{k,a} = q_m$. This test can be broken into two steps: first, one checks if any of the states in Q_k is active, i.e. $D[k] = 1$; then, one verifies that the last character read is the last character of u_k, i.e. $L[a][k] = 1$. The whole test can then be implemented with the bitwise test $D \;\&\; M \;\&\; L[a] \neq 0^k$, where $M = (1 \ll (k-1))$.

The same considerations also hold for the suffix automaton $\mathcal{S}(P)$. The only difference is in the handling of the initial state. In the case of the automaton $\mathcal{A}(P)$, state q_0 is always active, so we have to activate state q_1 when the current text symbol is equal to $P[0]$. To do so it is enough to perform a bitwise or of

```
F-Shift-And (P, m, T, n)

(B, L, M, k) ← F-PREPROCESS(P, m)
D ← 0^k
a ← T[0]
for j ← 1 to n − 1
    if a = P[0] then D ← D | 0^{k−1}1
    if (D & M & L[a]) ≠ 0^k
        then Output(j − 1)
    D ← D & B[a][T[j]]
    H ← D & L[a]
    D ← (D & ∼H) | (H ≪ 1)
    a ← T[j]
```

(a)

```
F-BNDM (P, m, T, n)

(B, L, M, k) ← F-PREPROCESS(P^r, m)
j ← m − 1
while j < n do
    k ← 1, l ← 0
    D ← ∼0^k, a ← T[j]
    while D ≠ 0^k do
        if (D & M & L[a]) ≠ 0^k then
            if k < m then
                l ← k
            else Output(j)
        D ← D & B[a][T[j − k]]
        H ← D & L[a]
        D ← (D & ∼H) | (H ≪ 1)
        a ← T[j − k]
        k ← k + 1
    j ← j + m − l
```

(b)

Fig. 2. Variants of Shift-And and BNDM based on the 1-factorization encoding

D with $0^{k-1}1$ when $a = P[0]$, as $q_1 \in Q_1$. Instead, in the case of the suffix automaton $\mathcal{S}(P)$, as the initial state has an ε-transition to each state, all the bits in D must be set, as in the BNDM algorithm.

The preprocessing procedure which builds the arrays B and L described above and relative to a minimal 1-factorization of the given pattern $P \in \Sigma^m$ is reported in Figure 1. Its time complexity is $\mathcal{O}(|\Sigma|^2 + m)$. The variants of the Shift-And and BNDM algorithms based on our encoding of the configurations of the automata $\mathcal{A}(P)$ and $\mathcal{S}(P)$ are reported in Figure 2 (algorithms F-Shift-And and F-BNDM, respectively). Their worst-case time complexities are $\mathcal{O}(n\lceil k/w \rceil)$ and $\mathcal{O}(nm\lceil k/w \rceil)$, respectively, while their space complexity is $\mathcal{O}(|\Sigma|^2\lceil k/w \rceil)$, where k is the size of a minimal 1-factorization of the pattern.

5 Experimental Results

In this section we present and comment the experimental results relative to an extensive comparison of the BNDM and the F-BNDM algorithms and the Shift-And and F-Shift-And algorithms. In particular, in the BNDM case we have implemented two variants for each algorithm, named *single word* and *multiple words*, respectively. Single word variants are based on the automaton for a suitable substring of the pattern whose configurations can fit in a computer word; a naive check is then used to verify whether any occurrence of the subpattern can be extended to an occurrence of the complete pattern: specifically, in the case of the BNDM algorithm, the prefix pattern of length $\min(m, w)$ is chosen, while in the case of the F-BNDM algorithm the longest substring of the pattern which is a concatenation of at most w consecutive factors is selected. Multiple words variants are based on the automaton for the complete pattern whose configurations are splitted, if needed, over multiple machine words. The resulting implementations are referred to in the tables below as BNDM* and F-BNDM*.

We have also included in our tests the LBNDM algorithm [8]. When the alphabet is considerably large and the pattern length is at least two times the word size, the LBNDM algorithm achieves larger shift lengths. However, the time for its verification phase grows proportionally to m/w, so there is a treshold after which its performance degrades significantly.

For the Shift-And case, only test results relative to the multiple words variant have been included in the tables below, since the overhead due to a more complex bit-parallel simulation in the single word case is not paid off by the reduction of the number of calls to the verification phase.

The main two factors on which the efficiency of BNDM-like algorithms depends are the maximum shift length and the number of words needed for representing automata configurations. For the variants of the first case, the shift length can be at most the length of the longest substring of the pattern that fits in a computer word. This, for the BNDM algorithm, is plainly equal to $\min(w, m)$, so the word size is an upper bound for the shift length, whereas in the case of the F-BNDM algorithm it is generally possible to achieve shifts of length larger

than w, as our encoding allows to pack more state configurations per bit on the average as shown in a table below. In the multi-word variants, the shift lengths for both algorithms, denoted BNDM* and F-BNDM*, are always the same, as they use the very same automaton; however, the 1-factorization based encoding involves a smaller number of words on the average, especially for long patterns, thus providing a considerable speedup.

All algorithms have been implemented in the C programming language and have been compiled with the `GNU C Compiler`, using the optimization options `-O2 -fno-guess-branch-probability`. All tests have been performed on a 2 GHz Intel Core 2 Duo and running times have been measured with a hardware cycle counter, available on modern CPUs. We used the following input files: (i) the English King James version of the "Bible" (with $\sigma = 63$); (ii) a protein sequence from the *Saccharomyces cerevisiae* genome (with $\sigma = 20$); and (iii) a genome sequence of $4,638,690$ base pairs of *Escherichia coli* (with $\sigma = 4$).

Files (i) and (iii) are from the Canterbury Corpus [1], while file (ii) is from the Protein Corpus [7]. For each input file, we have generated sets of 200 patterns of fixed length m randomly extracted from the text, for m ranging over the values 32, 64, 128, 256, 512, 1024, 1536, 2048, 4096. For each set of patterns we reported the mean over the running times of the 200 runs.

m	Shift-And*	F-Shift-And*	LBNDM	BNDM	BNDM*	F-BNDM	F-BNDM*
32	8.85	22.20	2.95	**2.81**	2.92	2.92	2.99
64	51.45	22.20	**1.83**	2.82	3.31	2.00	1.97
128	98.42	22.21	**1.82**	2.83	3.58	2.35	2.23
256	142.27	92.58	**1.38**	2.82	2.79	1.91	2.14
512	264.21	147.79	**1.09**	2.84	2.47	1.81	1.75
1024	508.71	213.70	**1.04**	2.84	2.67	1.77	1.72
1536	753.02	283.57	**1.40**	2.84	2.95	1.77	1.73
2048	997.19	354.32	2.24	2.84	3.45	**1.75**	1.90
4096	1976.09	662.06	10.53	2.83	6.27	**1.72**	2.92

Experimental results on the King James version of the Bible ($\sigma = 63$)

m	Shift-And*	F-Shift-And*	LBNDM	BNDM	BNDM*	F-BNDM	F-BNDM*
32	6.33	15.72	1.50	1.58	1.64	**1.43**	1.56
64	38.41	15.70	0.99	1.57	1.70	**0.89**	0.96
128	70.59	40.75	0.70	1.57	1.42	**0.64**	1.01
256	104.42	73.59	**0.52**	1.57	1.39	0.59	1.01
512	189.16	108.33	**0.41**	1.57	1.29	0.56	0.88
1024	362.83	170.52	**0.54**	1.58	1.46	0.55	0.91
1536	540.25	227.98	2.09	1.57	1.73	**0.56**	1.04
2048	713.87	290.24	7.45	1.58	2.12	**0.56**	1.20
4096	1413.76	541.53	32.56	1.58	4.87	**0.59**	2.33

Experimental results on a protein sequence from the *Saccharomyces cerevisiae* genome ($\sigma = 20$)

m	Shift-And*	F-Shift-And*	LBNDM	BNDM	BNDM*	F-BNDM	F-BNDM*
32	10.19	25.04	4.60	**3.66**	3.82	5.18	4.88
64	59.00	42.93	3.42	3.64	5.39	**2.94**	3.69
128	93.97	114.22	3.43	3.65	5.79	**2.66**	5.37
256	162.79	167.11	11.68	3.64	4.79	**2.59**	4.11
512	301.55	281.37	82.94	3.66	4.16	**2.53**	3.54
1024	579.92	460.37	96.13	3.64	4.21	**2.50**	3.42
1536	860.84	649.88	91.45	3.64	4.54	**2.49**	3.66
2048	1131.50	839.32	89.45	3.64	4.98	**2.48**	3.98
4096	2256.37	1728.71	85.87	3.64	7.81	**2.48**	6.22

Experimental results on a genome sequence of *Escherichia coli* ($\sigma = 4$)

(A)	ecoli	protein	bible
32	32	32	32
64	63	64	64
128	72	122	128
256	74	148	163
512	77	160	169
1024	79	168	173
1536	80	173	176
2048	80	174	178
4096	82	179	182

(B)	ecoli	protein	bible
32	15	8	6
64	29	14	12
128	59	31	26
256	119	60	50
512	236	116	102
1024	472	236	204
1536	705	355	304
2048	944	473	407
4096	1882	951	813

(C)	ecoli	protein	bible
32	2.13	4.00	5.33
64	2.20	4.57	5.33
128	2.16	4.12	4.92
256	2.15	4.26	5.12
512	2.16	4.41	5.01
1024	2.16	4.33	5.01
1536	2.17	4.32	5.05
2048	2.16	4.32	5.03
4096	2.17	4.30	5.03

(A) The length of the longest substring of the pattern fitting in w bits.
(B) The size of the minimal 1-factorization of the pattern.
(C) The ratio between m and the size of the minimal 1-factorization of the pattern.

Concerning the BNDM-like algorithms, the experimental results show that in the case of long patterns both variants based on the 1-factorization encoding are considerably faster than their corresponding variants BNDM and BNDM*. In the first test suite, with $\sigma = 63$, the LBNDM algorithm turns out to be the fastest one, except for very long patterns, as the treshold on large alphabets is quite high. In the second test suite, with $\sigma = 20$, LBNDM is still competitive but, in the cases in which it beats the BNDM* algorithm, the difference is thin.

Likewise, the F-Shift-And variant is faster than the classical Shift-And algorithm in all cases, for $m \geqslant 64$.

6 Conclusions

We have presented an alternative technique, suitable for bit-parallelism, to represent the nondeterministic automaton and the nondeterministic suffix automaton of a given string. On the average, the new encoding allows to pack in a single machine word of w bits state configurations of (suffix) automata relative to strings of more than w characters long. When applied to the BNDM algorithm, and for long enough patterns, our encoding allows larger shifts in the case of the single word variant and a more compact encoding in the case of the multiple words variant, resulting in faster implementations.

Further compactness could be achieved with 2-factorizations (with the obvious meaning), or with hybrid forms of factorizations. Clearly, more involved factorizations will also result into more complex bit-parallel simulations and larger space complexity, thus requiring a careful tuning to identify the best degree of compactness for the application at hand.

References

1. Arnold, R., Bell, T.: A corpus for the evaluation of lossless compression algorithms. In: DCC 1997: Proceedings of the Conference on Data Compression, Washington, DC, USA, p. 201. IEEE Computer Society, Los Alamitos (1997), `http://corpus.canterbury.ac.nz/`
2. Baeza-Yates, R., Gonnet, G.H.: A new approach to text searching. Commun. ACM 35(10), 74–82 (1992)

3. Boyer, R.S., Moore, J.S.: A fast string searching algorithm. Communications of the ACM 20(10), 762–772 (1977)
4. Crochemore, M., Rytter, W.: Text algorithms. Oxford University Press, Oxford (1994)
5. Knuth, D.E., Morris Jr., J.H., Pratt, V.R.: Fast pattern matching in strings. SIAM Journal on Computing 6(1), 323–350 (1977)
6. Navarro, G., Raffinot, M.: A bit-parallel approach to suffix automata: Fast extended string matching. In: Farach-Colton, M. (ed.) CPM 1998. LNCS, vol. 1448, pp. 14–33. Springer, Heidelberg (1998)
7. Nevill-Manning, C.G., Witten, I.H.: Protein is incompressible. In: DCC 1999: Proceedings of the Conference on Data Compression, Washington, DC, USA, p. 257. IEEE Computer Society, Los Alamitos (1999), `http://data-compression.info/Corpora/ProteinCorpus/`
8. Peltola, H., Tarhio, J.: Alternative algorithms for bit-parallel string matching. In: Nascimento, M.A., de Moura, E.S., Oliveira, A.L. (eds.) SPIRE 2003. LNCS, vol. 2857, pp. 80–94. Springer, Heidelberg (2003)

Algorithms for Three Versions of the Shortest Common Superstring Problem

Maxime Crochemore[1,3], Marek Cygan[2], Costas Iliopoulos[1,4], Marcin Kubica[2], Jakub Radoszewski[2], Wojciech Rytter[2,5], and Tomasz Waleń[2]

[1] King's College London, London WC2R 2LS, UK
maxime.crochemore@kcl.ac.uk, csi@dcs.kcl.ac.uk
[2] Dept. of Mathematics, Computer Science and Mechanics, University of Warsaw, Warsaw, Poland
{cygan,kubica,jrad,rytter,walen}@mimuw.edu.pl
[3] Université Paris-Est, France
[4] Digital Ecosystems & Business Intelligence Institute, Curtin University of Technology, Perth WA 6845, Australia
[5] Dept. of Math. and Informatics, Copernicus University, Toruń, Poland

Abstract. The input to the Shortest Common Superstring (SCS) problem is a set S of k words of total length n. In the classical version the output is an explicit word $SCS(S)$ in which each $s \in S$ occurs at least once. In our paper we consider two versions with multiple occurrences, in which the input includes additional numbers (multiplicities), given in binary. Our output is the word $SCS(S)$ given implicitly in a compact form, since its real size could be exponential. We also consider a case when all input words are of length two, where our main algorithmic tool is a compact representation of Eulerian cycles in multigraphs. Due to exponential multiplicities of edges such cycles can be exponential and the compact representation is needed. Other tools used in our paper are a polynomial case of integer linear programming and a min-plus product of matrices.

1 Introduction

The algorithmic aspects of the SCS problem are thoroughly studied in theoretical as well as in practical computer science. In this paper we consider two variations of the SCS problem related to the number of occurrences of input words: Uniform Multiple Occurrences SCS (SUM-SCS) and Multiple Occurrences SCS Problem (MULTI-SCS). Our algorithms use several interesting combinatorial tools: Eulerian cycles in multigraphs, shortest paths via matrix multiplication and integer linear programming with a constant number of variables.

The SCS problem and its variations are studied in their own and also from the point of view of computational biologists. Recent advances, based either on sequencing by synthesis or on hybridisation and ligation, are producing millions of short reads overnight. An important problem with these technologies is how

A. Amir and L. Parida (Eds.): CPM 2010, LNCS 6129, pp. 299–309, 2010.

to efficiently and accurately map these short reads to a reference genome [12] that serves as a framework. The Genome Assembly Problem is as follows: given a large number of short DNA sequences (often called "fragments") generated by a sequencing machine like the ones mentioned above, put them back together to create a representation of the chromosome that sequences were taken from (usually a single organism). There are several software tools for this (called assemblers), to name a few: Celera Assembler [13] (mostly long reads), SSAKE [16], SHARCGS [5], SHRAP [14], Velvet [17]. When the problem of genome assembly arisen, computer scientists came up with the above abstraction of the SCS Problem, which they showed to be NP-hard [8], and developed several efficient approximation algorithms for it (see for example [1,2,3,7,15]). However, the SCS problem actually does not address the following issue of the biological Genome Assembly Problem — multiple occurrences of the fragments in the assembly. The shortest common superstring algorithms result in a shortest word but ignore repeated occurrences. In our paper we consider some variations of the problem in which we deal with multiple occurrences. First we consider a special case of SCS in which all fragments are of length 2. It happens that even this simple version is nontrivial in presence of multiplicities. Then a special case of constant number of words in S is considered. It is also nontrivial: we use sophisticated polynomial time algorithms for integer linear programs with constant number of variables.

Definitions of problems

Let $S = \{s_1, s_2, \ldots, s_k\}$ be the set of input words, $s_i \in \Sigma^*$. In all the problems defined below, we assume that S is a factor-free set, i.e. none of the words s_i is a factor of any other word from S. Let n denote the total length of all words in S. Let $\#occ(u, v)$ be the number of occurrences (as a factor) of the word u in the word v. We consider three problems:

SUM-SCS(k):
Given a positive integer m, find a shortest word u such that

$$\sum_{i=1}^{k} \#occ(s_i, u) \geq m \ .$$

MULTI-SCS(k):
Given a sequence of non-negative integers $m_1, m_2, \ldots, m_k$, find a shortest word u such that: $\#occ(s_i, u) \geq m_i$ for each $i = 1, 2, \ldots, k$.

MULTI-SCS$_2$(k):
A special case of the **MULTI-SCS(k)** problem where all input words s_i are of length 2.

We assume, for simplicity, that the binary representation of each of the numbers in the input consists of $O(n)$ bits, i.e. $m = O(2^n)$ in **SUM-SCS(k)** and $m_i = O(2^n)$ in **MULTI-SCS(k)**. Moreover, we assume that such numbers fit in a single memory cell in the RAM model, and thus operations on such numbers can be performed in constant time (if this is not the case, one would need to multiply the time complexities of the algorithms presented here by a polynomial

of the length of binary representation of numbers). Also, the total size of input data in each of the problems is $O(n)$.

By *finding* the SCS in each of the problems we mean computing its length and its compressed representation which is of size polynomial in n, that can be used to reconstruct the actual word in a straightforward manner in $O(\ell)$ time, where ℓ is the length of the word (this could be a context-free grammar, a regular expression etc).

2 Preliminaries

Let the overlap $ov(s,t)$ of two non-empty words, s and t, be the longest word y, such that $s = xy$ and $t = yz$ for some words $x \neq \varepsilon$ and z. We define $ov(s,t) = \varepsilon$ if $s = \varepsilon$ or $t = \varepsilon$. Also, let the prefix $pr(s,t)$ of s, w.r.t. t, be the prefix of s of length $|s| - |ov(s,t)|$ — therefore $s = pr(s,t)ov(s,t)$. For a given set $S = \{s_1, s_2, \ldots, s_k\}$ of words, the *prefix graph* of S is a directed graph with labeled and weighted edges defined as follows. The set of vertices is $\{0, 1, 2, \ldots, k, k+1\}$; vertices $1, 2, \ldots, k$ represent the words $s_1, s_2, \ldots, s_k$ and 0, $k+1$ are two additional vertices called *source* and *destination*, each of which corresponds to an empty word $s_0 = s_{k+1} = \varepsilon$. The edges are labeled with words, and their lengths (weights) are just the lengths of the labels. For all $0 \leq i, j \leq k+1$, $i \neq k+1$, $j \neq 0$, there is an edge (s_i, s_j) labeled with $pr(s_i, s_j)$. Note that, for a factor-free set S, the concatenation of labels of all edges in a path of the form $0 = v_1, v_2, v_3, \ldots, v_{p-1}, v_p = k+1$, i.e.

$$pr(s_{v_1}, s_{v_2})pr(s_{v_2}, s_{v_3}) \ldots pr(s_{v_{p-1}}, s_{v_p}) \ ,$$

represents a shortest word containing words $s_{v_2}, s_{v_3}, \ldots, s_{v_{p-1}}$ in that order. The prefix graph can easily be constructed in $O(k \cdot \sum_{i=1}^{k} |s_i|)$ time, using the prefix function from the Morris-Pratt algorithm [4]. However, this can also be done in the optimal time complexity $O(\sum_{i=1}^{k} |s_i| + k^2)$ — see [9].

Let A and B be matrices of size $(k+2) \times (k+2)$ containing non-negative numbers. The min-plus product $A \oplus B$ of these matrices is defined as:

$$(A \oplus B)[i,j] = \min\{A[i,q] + B[q,j] \ : \ q = 0, 1, \ldots, k+1\} \ .$$

We assume that the reader is familiar with a basic theory of formal languages and automata, see [10].

3 MULTI-SCS$_2(k)$ Problem

First, let us investigate a variant of the **MULTI-SCS**(k) problem in which all input words s_i are of length 2. Note that in such a case $n = 2k$. It is a folklore knowledge that **MULTI-SCS**$_2(k)$ can be solved in polynomial time when all multiplicities m_i are equal to one. We prove a generalization of this result for the **MULTI-SCS**$_2(k)$ problem:

Theorem 1. *The* **MULTI-SCS**$_2(k)$ *problem can be solved in* $O(n^2)$ *time. The length of the shortest common superstring can be computed in* $O(n)$ *time, and its compact representation of size* $O(n^2)$ *can be computed in* $O(n^2)$ *time. (The real size of the output could be exponential.)*

Proof. Let us construct a multigraph $G = (V, E)$, such that each vertex corresponds to a letter of the alphabet Σ, and each edge corresponds to some word s_i — $(u, v) \in E$ if the word uv is an element of S. Each word s_i has a given multiplicity m_i, therefore we equip each edge $e \in E$ corresponding to s_i with its multiplicity $c(e) = m_i$. Using this representation the graph G has size $O(k)$ ($|E| = k$, and $|V| = O(k)$ if we remove isolated vertices). We refer to such an encoding as to a *compact multigraph representation.*

Any solution for the **MULTI-SCS**$_2(k)$ problem can be viewed as a path in some supergraph $G' = (V, E \cup E')$ of G, passing through each edge $e \in E$ at least $c(e)$ times. We are interested in finding the shortest solution, consequently we can reduce the problem to finding the smallest cardinality multiset of edges E' such that the multigraph $(V, E \cup E')$ has an Eulerian path. To simplify the description, we find the smallest multiset E' for which the graph $(V, E \cup E')$ has an Eulerian cycle, and then reduce the cycle to a path (if $E' \neq \emptyset$).

MULTI-SCS$_2(k)$ can be solved using the following algorithm:

```
1: construct the multigraph G = (V, E)
2: find the smallest cardinality multiset of edges E' such that G' =
   (V, E ∪ E') has an Eulerian cycle
3: find an Eulerian cycle C in G'
4: if E' ≠ ∅ then
5:    return path P obtained from C by removing one edge from E'
6: else
7:    return C
```

As a consequence of the following Lemma 1, the smallest cardinality multiset E' can be computed in $O(|V| + |E|)$ time. The compact representation of an Eulerian cycle can be computed, by Lemma 2, in $O(|V| \cdot |E|)$ time. This gives us an $O(|V| \cdot |E|) = O(n^2)$ time algorithm for the **MULTI-SCS**$_2(k)$ problem. □

Lemma 1. *For a given compact representation of a directed multigraph* $G = (V, E)$*, there exists an* $O(|V| + |E|)$ *time algorithm for computing the smallest cardinality multiset of edges* E'*, such that the multigraph* $G' = (V, E \cup E')$ *has an Eulerian cycle.*

Proof. In the trivial case, when G already has an Eulerian cycle, we return $E' = \emptyset$. Let $C_1, C_2, \ldots, C_q$ ($C_i \subseteq V$) be connected components in the undirected version of the multigraph G. For each component we compute its demand D_i defined as follows:

$$D_i = \sum_{v \in C_i} \max(indeg(v) - outdeg(v), 0)$$

where $indeg(v)$ (resp. $outdeg(v)$) is the in (resp. out) degree of a vertex v. Let the demand of the vertex v be defined as $d(v) = |indeg(v) - outdeg(v)|$. Let

$V^+(C_i)$ (resp. $V^-(C_i)$) be the set of vertices $v \in V(C_i)$, such that $indeg(v) > outdeg(v)$ (resp. $indeg(v) < outdeg(v)$). We describe an algorithm that computes the smallest cardinality multiset E' of the size:

$$|E'| = \sum_{i=1}^{q} \max(1, D_i) \ .$$

First, let us observe that an edge $e = (u, v) \in E'$ can have one of the following contributions:

- if $u, v \in C_i$ then e can decrease the demand D_i by at most 1,
- if $u \in C_i$ and $v \in C_j$ ($i \neq j$, with $D_i > 0$ and $D_j > 0$) then e merges C_i, C_j into a single component with demand at least $D_i + D_j - 1$,
- if $u \in C_i$ and $v \in C_j$ ($i \neq j$, with $D_i = 0$ or $D_j = 0$) then e merges C_i, C_j into a single component with demand at least $D_i + D_j$.

To construct the optimal set E', we use the following algorithm (see Fig. 1):

```
 1: E' = ∅
 2: first we connect all components:
 3: while number of components > 1 do
 4:     let C_i, C_j be two different components
 5:     if D_i ≠ 0 then let u be a vertex from the set V^+(C_i), otherwise from V(C_i)
 6:     if D_j ≠ 0 then let v be a vertex from the set V^-(C_j), otherwise from V(C_j)
 7:     add to E' a single edge (u, v) (with c((u, v)) = 1)
 8: from now on we have only one component C in G, we reduce its demand to 0:
 9: while V^+(C) ≠ ∅ do
10:     let v^+ ∈ V^+(C), v^- ∈ V^-(C)
11:     add to E' an edge (v^+, v^-) with multiplicity min(d(v^+), d(v^-))
12: return E'
```

Observe that if a component C_i admits $D_i > 0$ it means that both sets $V^-(C_i)$ and $V^+(C_i)$ are nonempty. In the first phase we add exactly $q - 1$ edges to E'. In the second phase we reduce the total demand to 0, each iteration of the **while** loop reduces the demand of at least one vertex to 0. Hence we add at most $O(|V|)$ different edges in the second phase. If we store E' using a compact representation, its size is $O(|V| + |E|)$ and the above algorithm computes it in such time complexity. □

Lemma 2. *For a given compact representation of a directed Eulerian multigraph $G = (V, E)$, there exists an $O(|V| \cdot |E|)$ time algorithm for computing the representation of an Eulerian cycle in G.*

Proof. The Eulerian cycle can be of exponential size, therefore we construct its compressed representation. Such a representation is an expression of the form

$$\pi = w_1^{p_1} w_2^{p_2} \dots w_\ell^{p_\ell} \ ,$$

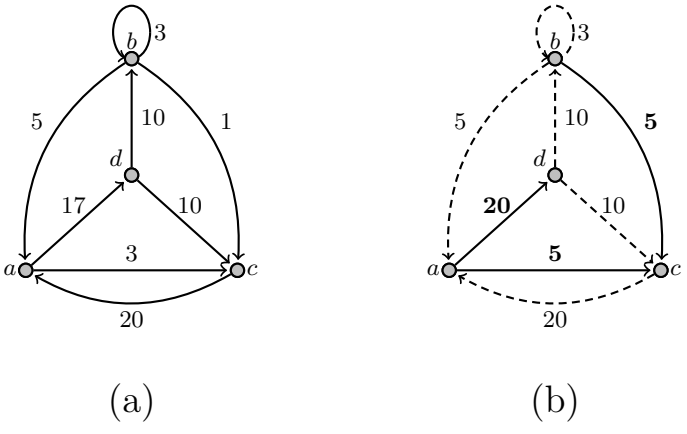


Fig. 1. (a) Multigraph $G = (V, E)$ for a set of words $S = \{ad, ac, ba, bb, bc, ca, db, dc\}$ with a sequence of multiplicities $(m_i)_{i=1}^{8} = (17, 3, 5, 3, 1, 20, 10, 10)$. It consists of a single component C with $V^+(C) = \{a, b\}$ and $V^-(C) = \{c, d\}$. (b) Eulerian multigraph $G' = (V, E \cup E')$ obtained from G by adding the following minimal multiset of edges: $E' = \{(a, d) \cdot 3,\ (a, c) \cdot 2,\ (b, c) \cdot 4\}$.

where w_i is a path in G, and p_i is a non-negative integer. We say that an occurrence of $v \in w_i$ is *free* if $p_i = 1$. The algorithm is a slight modification of the standard algorithm for computing Eulerian cycles in graphs, see Fig. 2 and 3.

Function EulerianCycle(G) {assume G is an Eulerian multigraph}
1: find any simple cycle C in G
2: let $c_{\min} = \min\{c(e) : e \in C\}$
3: let $\pi = C^1\ C^{c_{\min}-1}$ { so that each $v \in V(C)$ has a free occurrence }
4: **for all** $e \in C$ **do**
5: decrease $c(e)$ by $c_{\min}$, if $c(e) = 0$ remove e from $E(G)$
6: **for all** strongly connected components W_i of G **do**
7: let v_i be any common vertex of $V(C)$ and $V(W_i)$
8: let $\pi_i = \text{EulerianCycle}(W_i)$
9: insert cycle π_i to π after some free occurrence of vertex v_i
10: **return** π

We can find a simple cycle in the Eulerian graph G in $O(|V|)$ time by going forward until we get to an already visited vertex. Each cycle removes at least one edge from E, so the algorithm investigates at most $|E|$ cycles. A simple implementation of lines 6-9 yields $O(|V| \cdot |E|)$ time complexity per recursive step, however, with a careful approach (G not decomposed to W_i explicitly, π implemented as a doubly-linked list) one can obtain $O(|V|)$ time complexity of these lines. Thus the time complexity and the total size of the representation π is $O(|V| \cdot |E|)$. □

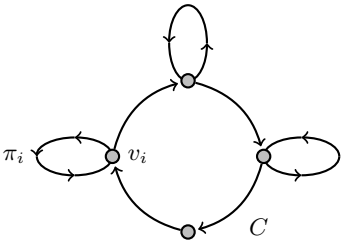


Fig. 2. Construction of an Eulerian cycle by algorithm EulerianCycle(G)

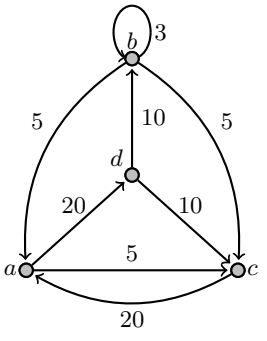


Fig. 3. Eulerian multigraph obtained for a set of words $S = \{ad, ac, ba, bb, bc, ca, db, dc\}$ with a sequence of multiplicities $(m_i)_{i=1}^{8} = (20, 5, 5, 3, 5, 20, 10, 10)$. The compressed representation of an Eulerian cycle can have the following form: $(a{\to}d{\to}b)(b{\to}b)^3(b{\to}a)(a{\to}d{\to}b{\to}a)^4(a{\to}c{\to}a)^5(a{\to}d{\to}b{\to}c{\to}a)^5(a{\to}d{\to}c{\to}a)^{10}$, which corresponds to a word $adb(b)^3a(dba)^4(ca)^5(dbca)^5(dca)^{10}$

4 MULTI-SCS(k) Problem for $k = O(1)$

Let us consider a prefix graph G of $S = \{s_1, s_2, \ldots, s_k\}$. In order to solve the general **MULTI-SCS(k)** problem, it suffices to find the shortest path π from 0 to $k+1$ in G that passes through each vertex $i \in V(G)$, for $1 \le i \le k$, at least m_i times. We assume that $k = O(1)$.

Let us treat G as a (deterministic) finite automaton A: 0 is its start state, $k+1$ is its accept state, and an edge from i to j in G is identified by a triple $(i, j, |pr(s_i, s_j)|)$ which represents its starting and ending vertex and its length. Let $\Gamma \subseteq \{0, \ldots, k{+}1\} \times \{0, \ldots, k{+}1\} \times (\mathbb{Z}_+ \cup \{0\})$ be the set of triples identifying all edges of G. Each path from 0 to $k+1$ corresponds to a word (from Γ^*) in the language accepted by A.

Let $\alpha(A)$ be a regular expression corresponding to the language accepted by A — its size is $O(1)$ and it can be computed in $O(1)$ time [10] (recall that $k = O(1)$).

Definition 1. *We call two words $u, v \in \Gamma^*$* commutatively equivalent *(notation: $u \approx v$) if for any $a \in \Gamma$, $\#occ(a, u) = \#occ(a, v)$. We call two regular languages*

L_1, L_2 commutatively equivalent *(notation: $L_1 \approx L_2$) if for each word $u \in L_1$ ($u \in L_2$) there exists a word $v \in L_2$ ($v \in L_1$) such that $u \approx v$.*

Lemma 3. *The regular expression $\alpha(A)$ can be transformed in $O(1)$ time into a regular expression $\beta(A)$ such that:*

$$\mathcal{L}(\beta(A)) \subseteq \mathcal{L}(\alpha(A)) \text{ and } \mathcal{L}(\beta(A)) \approx \mathcal{L}(\alpha(A)) \tag{1}$$

and $\beta(A)$ is in the following normal form:

$$\beta(A) = B_1 + B_2 + \ldots + B_k$$

where $B_i = C_{i,1}C_{i,2}\ldots C_{i,l_i}$ and each $C_{i,j}$ is:

- *either $a \in \Gamma$,*
- *or $(a_1 a_2 \ldots a_p)^*$, where $a_r \in \Gamma$.*

Proof. The proof contains an algorithm for computing $\beta(A)$ in $O(1)$ time.

In the first step we repetively use the following transformations in every possible part of $\alpha(A)$ until all Kleene's stars contain only concatenation of letters from Γ (all letters in the transformations denote regular expressions):

$$\big(\gamma(\delta + \sigma)\rho\big)^* \to (\gamma\delta\rho)^*(\gamma\sigma\rho)^* \tag{2}$$

$$(\gamma\delta^*\sigma)^* \to \big(\gamma\delta^*\sigma(\gamma\sigma)^*\big) + \varepsilon\ . \tag{3}$$

Since then, it suffices to repetively use the following transformation to obtain the required normal form:

$$\gamma(\delta + \sigma)\rho \to (\gamma\delta\rho) + (\gamma\sigma\rho)\ . \tag{4}$$

It is easy to check that each of the transformations (2)–(4) changes the regular expression into another regular expression such that the language defined by the latter is a commutatively equivalent sublanguage of the language defined by the former. □

Let us notice that, due to the conditions (1), from our point of view $\beta(A)$ may serve instead of $\alpha(A)$ — for each path generated by the expression $\alpha(A)$ there exists a path generated by $\beta(A)$ such that the multisets of edges visited in both paths are exactly the same (thus the paths are of the same length).

To compute the result for $\beta(A)$, we process each B_i (see Lemma 3) separately and return the minimum of values computed. When computing the result (the shortest path generated by it that visits each vertex an appropriate number of times) for a given B_i, the only choices we might have are in those $C_{i,j}$'s that are built using Kleene's star. For each of them we introduce a single integer variable x_j that is used to denote the number of times we take the given fragment of the expression in the path we are to construct. For a given set of values of variables x_j, it is easy to compute, for each vertex y of the graph, how many times it is visited in the word, representing a path, generated by B_i:

$$\#(y, B_i) = \sum_{j=1}^{l_i} \#(y, C_{i,j})$$

and what is the total length of the word:

$$len(B_i) = \sum_{j=1}^{l_i} len(C_{i,j}) \ .$$

If $C_{i,j} = a$, for $a = (v, w, \ell) \in \Gamma$, then

$$\#(y, C_{i,j}) = \delta_{wy} \qquad len(C_{i,j}) = \ell$$

and if $C_{i,j} = (a_1 a_2 \dots a_p)^*$, where $a_r \in \Gamma$ for $1 \le r \le p$, then

$$\#(y, C_{i,j}) = x_j \cdot \sum_{r=1}^{p} \#(y, a_r) \qquad len(C_{i,j}) = x_j \cdot \sum_{r=1}^{p} len(a_r) \ .$$

Here δ_{xy} denotes the Kronecker delta: $\delta_{x,x} = 1$, and $\delta_{x,y} = 0$ for $x \neq y$.

If values of the variables are not fixed, we can treat $\#(y, B_i)$ and $len(B_i)$ as (linear) expressions over those variables. Our goal is, therefore, to minimize the value of $len(B_i)$ under the following constraints on variables $x_j \in \mathbb{Z}$:

$$\begin{aligned} x_j &\ge 0 \\ \#(y, B_i) &\ge m_y \quad \text{for} \ \ y = 1, 2, \dots, k \ . \end{aligned}$$

But this is exactly an integer linear programming problem (see Example 1). For a fixed number of variables and constraints it can be solved in polynomial time in the length of the input, see Lenstra's paper [11], and even in linear time in terms of the maximum encoding length of a coefficient, see Eisenbrand's paper [6].

Example 1. Assume that $S = \{s_1, s_2\}$, $m_1 = 2010$, $m_2 = 30$. Note that the set of symbols in the regular expressions $\alpha(A)$ and $\beta(A)$ is $\Gamma \subseteq \{0, \dots, 3\} \times \{0, \dots, 3\} \times (\mathbb{Z}_+ \cup \{0\})$. Let

$$B_i = (0,1,0)(1,1,7)^*(1,2,3)\big((2,1,2)(1,1,7)(1,2,3)\big)^*(2,3,5)$$

be a part of the expression $\beta(A)$ for which we are to compute the shortest path satisfying the occurrence conditions.

Observe that we can interpret our task as a graph problem. We are given a directed graph having a form of a directed path (a backbone) with disjoint cycles attached, in which vertices are labeled with indices from the set $\{0, \dots, 3\}$ and edges are labeled with lengths, as in Fig. 3. In this graph we need to find the length of the shortest path from the vertex labeled 0 to the vertex labeled $k + 1 = 3$ visiting at least $m_1 = 2010$ 1-labeled vertices and at least $m_2 = 30$ 2-labeled vertices.

In the integer program we introduce two variables x_1, x_2 that uniquely determine a word generated by B_i:

$$(0,1,0)(1,1,7)^{x_1}(1,2,3)\big((2,1,2)(1,1,7)(1,2,3)\big)^{x_2}(2,3,5) \ .$$

The integer program for this example looks as follows:

$$\begin{aligned} x_1, x_2 &\ge 0 \\ 1 + x_1 + 2x_2 = \#(1, B_i) &\ge m_1 = 2010 \\ 1 + x_2 = \#(2, B_i) &\ge m_2 = 30 \end{aligned}$$

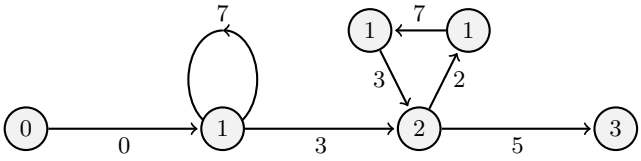


Fig. 4. Labeled graph corresponding to B_i from Example 1. The variables x_1 and x_2 from the integer program correspond to the number of times the loop $(1 \to 1)$ and the cycle $(2 \to 1 \to 1 \to 2)$ are traversed in the shortest path.

and we are minimizing the expression

$$len(B_i) = 0 + 7x_1 + 3 + 12x_2 + 5 \ .$$

To recompute the actual SCS, we choose the one B_i that attains the globally smallest value of $len(B_i)$. Note that solving the integer program gives us the values of variables x_j, from which we can restore the corresponding word v generated by the regular expression B_i simply by inserting the values of x_j instead of Kleene's stars, resulting in a polynomial representation of the shortest common superstring.

Theorem 2. MULTI-SCS(k) *can be solved in $O(poly(n))$ time for $k = O(1)$.*

5 SUM-SCS(k) Problem

Let G be the prefix graph of $S = \{s_1, s_2, \ldots, s_k\}$. We are looking for the shortest word containing m occurrences of words from S. Recall that such a word corresponds to the shortest path in G from the source to the destination, traversing $m + 1$ edges. Let M be the adjacency matrix of G. The length of the shortest path from the vertex 0 to the vertex $k + 1$ passing through $m + 1$ edges equals $M^{m+1}[0, k + 1]$, where M^{m+1} is the $(m + 1)^{th}$ power of M w.r.t. the min-plus product. M^{m+1} can be computed in $O(k^3 \log m)$ time by repeated squaring, i.e. using identities:

$$M^{2p} = (M^p)^2 \qquad\qquad M^{2p+1} = M \oplus M^{2p} \ .$$

Having computed the described matrices, we can also construct a representation of SCS of size $O(poly(n))$ by a context-free grammar. The set of terminals is Σ. For each of the matrices M^p that we compute, we create an auxiliary matrix K_p containing distinct non-terminals of the grammar. If M^p (for $p > 1$) is computed in the above algorithm using M^a and M^b then we add the following production from $K_p[i, j]$:

$$K_p[i, j] \Rightarrow K_a[i, q]K_b[q, j] \qquad \text{where} \quad M^p[i, j] = M^a[i, q] + M^b[q, j] \ .$$

The production from $K_1[i,j]$ is defined as:

$$K_1[i,j] \Rightarrow pr(s_i, s_j) \ .$$

The starting symbol of the grammar is $K_{m+1}[0, k+1]$.

Clearly, this representation uses $O(k^2 \log m)$ memory and the only word generated by this grammar is the requested SCS. Hence, we obtain the following theorem:

Theorem 3. *The* **SUM-SCS**(k) *problem can be solved in* $O(n+k^3 \log m)$ *time and* $O(n + k^2 \log m)$ *memory.*

References

1. Armen, C., Stein, C.: A 2 2/3-approximation algorithm for the shortest superstring problem. In: Hirschberg, D.S., Myers, G. (eds.) CPM 1996. LNCS, vol. 1075, pp. 87–101. Springer, Heidelberg (1996)
2. Blum, A., Jiang, T., Li, M., Tromp, J., Yannakakis, M.: Linear approximation of shortest superstrings. J. ACM 41(4), 630–647 (1994)
3. Breslauer, D., Jiang, T., Jiang, Z.: Rotations of periodic strings and short superstrings. Journal of Algorithms 24(2), 340–353 (1997)
4. Crochemore, M., Rytter, W.: Jewels of Stringology. World Scientific Publishing Company, Singapore (2002)
5. Dohm, J.C., Lottaz, C., Borodina, T., Himmelbauer, H.: SHARCGS, a fast and highly accurate short-read assembly algorithm for de novo genomic sequencing. Genome research 17(11), 1697–1706 (2007)
6. Eisenbrand, F.: Fast integer programming in fixed dimension. In: Di Battista, G., Zwick, U. (eds.) ESA 2003. LNCS, vol. 2832, pp. 196–207. Springer, Heidelberg (2003)
7. Gallant, J., Maier, D., Storer, J.A.: On finding minimal length superstrings. J. Comput. Syst. Sci. 20(1), 50–58 (1980)
8. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman & Co., New York (1979)
9. Gusfield, D., Landau, G.M., Schieber, B.: An efficient algorithm for the all pairs suffix-prefix problem. Inf. Process. Lett. 41(4), 181–185 (1992)
10. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages and Computation. Addison-Wesley, Reading (1979)
11. Lenstra Jr., H.W.: Integer programming with a fixed number of variables. Mathematics of Operations Research 8(4), 538–548 (1983)
12. Li, H., Ruan, J., Durbin, R.: Mapping short DNA sequencing reads and calling variants using mapping quality scores. Genome Research 18(11), 1851–1858 (2008)
13. Myers, E.W., et al.: A whole-genome assembly of drosophila. Science 287(5461), 2196–2204 (2000)
14. Sundquist, A., Ronaghi, M., Tang, H., Pevzner, P., Batzoglou, S.: Whole-genome sequencing and assembly with high-throughput, short-read technologies. PLoS ONE 2(5), e484 (2007)
15. Tarhio, J., Ukkonen, E.: A greedy approximation algorithm for constructing shortest common superstrings. Theor. Comput. Sci. 57(1), 131–145 (1988)
16. Warren, R.L., Sutton, G.G., Jones, S.J., Holt, R.A.: Assembling millions of short DNA sequences using SSAKE. Bioinformatics 23(4), 500–501 (2007)
17. Zerbino, D.R., Birney, E.: Velvet: algorithms for de novo short read assembly using de Bruijn graphs. Genome research 18(5), 821–829 (2008)

Finding Optimal Alignment and Consensus of Circular Strings

Taehyung Lee[1,⋆], Joong Chae Na[2,⋆⋆],
Heejin Park[3,⋆⋆⋆,†], Kunsoo Park[1,⋆], and Jeong Seop Sim[4,‡]

[1] Seoul National University, Seoul 151-742, South Korea
[2] Sejong University, Seoul 143-747, South Korea
[3] Hanyang University, Seoul 133-791, South Korea
[4] Inha University, Incheon 402-751, South Korea

Abstract. We consider the problem of finding the optimal alignment and consensus (string) of circular strings. Circular strings are different from linear strings in that the first (leftmost) symbol of a circular string is wrapped around next to the last (rightmost) symbol. In nature, for example, bacterial and mitochondrial DNAs typically form circular strings. The consensus string problem is finding a representative string (consensus) of a given set of strings, and it has been studied on linear strings extensively. However, only a few efforts have been made for the consensus problem for circular strings, even though circular strings are biologically important. In this paper, we introduce the consensus problem for circular strings and present novel algorithms to find the optimal alignment and consensus of circular strings under the Hamming distance metric. They are $O(n^2 \log n)$-time algorithms for three circular strings and an $O(n^3 \log n)$-time algorithm for four circular strings. Our algorithms are $O(n/\log n)$ times faster than the naïve algorithm directly using the solutions for the linear consensus problems, which takes $O(n^3)$ time for three circular strings and $O(n^4)$ time for four circular strings. We achieved this speedup by adopting a convolution and a system of linear equations into our algorithms to reflect the characteristics of circular strings that we found.

⋆ This work was supported by NAP of Korea Research Council of Fundamental Science & Technology. The ICT at Seoul National University provides research facilities for this study.

⋆⋆ This work was supported by the Korea Research Foundation(KRF) grant funded by the Korea government(MEST) (No. 2009-0069977).

⋆⋆⋆ This work was supported by 21C Frontier Functional Proteomics Project from Korean Ministry of Education, Science and Technology (FPR08-A1-020).

† Corresponding author, E-mail: hjpark@hanyang.ac.kr

‡ This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (2009-0090441).

A. Amir and L. Parida (Eds.): CPM 2010, LNCS 6129, pp. 310–322, 2010.

1 Introduction

A *circular* (or *cyclic*) string is the string that is constructed by linking the beginning and end of a (linear) string together, which can be often found in nature. Gusfield emphasized "Bacterial and mitochondrial DNA is typically circular, both in its genomic DNA and in plasmids, even some true eukaryotes contain plasmid DNA. Consequently, tools for handling circular strings may someday be of use in those organisms." (see [1], page 12)

Finding a representative string of a given set $\mathbb{S} = \{S_1, \ldots, S_m\}$ of m strings of equal length, called a *consensus string* (or *closest string* or *center string*), is a fundamental problem in multiple sequence alignment, which is closely related to the motif recognition problem. Among the conditions that a string should satisfy to be accepted as a consensus, the two most important conditions are

1. to minimize the sum of (Hamming) distances from the strings in $\mathbb{S}$ to the consensus, and
2. to minimize the longest distance (or radius) from the strings in $\mathbb{S}$ to the consensus.

In this paper we consider four different types of consensus problems, CS, CR, CSR, and BSR: Problem CS is finding the optimal consensus minimizing the distance sum, Problem CR is finding the optimal consensus minimizing the radius, Problem CSR is finding the optimal consensus minimizing both distance sum and radius if one exists, and finally Problem BSR is finding a consensus whose distance sum and radius are smaller than given thresholds.

There has been substantial research to solve the problems for linear strings. Problem CS is easy to solve. We can find a string that minimizes the distance sum by selecting the symbol occurring most often in each position of the strings in $\mathbb{S}$. However, Problem CR is hard in general. Given a parameter r, the problem of asking the existence of a string X such that $\max_{1 \le i \le m} d(X, S_i) \le r$ is NP-complete for general m, even when the symbols of the strings are drawn from a binary alphabet [2]. Thus, attention has been restricted to approximation solutions [3,4,5,6,7,8] and fixed-parameter solutions [8,9,10,11]. Furthermore, there have been some algorithms for a small constant m. Gramm et al. [9] proposed a direct combinatorial algorithm for Problem CR for three strings. Sze et al. [12] showed a condition for the existence of a string whose radius is less than or equal to r. Boucher et al. [13] proposed an algorithm for finding a string X such that $\max_{1 \le i \le 4} d(X, S_i) \le r$ for four binary strings. Problems CSR and BSR were considered by Amir et al [14]. They considered the problems for three strings. However, there have been only a few results on multiple alignment of circular strings, even though circular strings are biologically important. Our algorithms differ from the existing multiple alignment algorithms for circular strings [15,16], which use the sum-of-pairs score and some general purpose multiple sequence alignment techniques, such as `clustalW` [17].

In this paper, we introduce the consensus problem for circular strings and present novel algorithms to find the optimal alignment and consensus of circular strings. The consensus problem for circular strings is different from the consensus

problem for linear strings because every alignment of the circular strings should be considered to find the optimal alignment and a consensus string, which is not necessary in the case of linear strings. Our contributions are as follows:

- We present an algorithm to solve Problem CS for three circular strings in $O(n^2 \log n)$ time. This algorithm uses convolution to compute the minimum distance sum for each alignment and selects the mininum among them.
- Problems CR, CSR, and BSR for three circular strings can be solved in $O(n^2 \log n)$ time. The crux of our algorithms is computing the minimum radius, which requires both convolution and solving a system of linear equations. Since those algorithms are similar and due to the page limit, we only present the algorithm for Problem CSR.
- We present an algorithm to solve Problem CS for four circular strings in $O(n^3 \log n)$ time. For four circular strings, it requires both convolution and solving a system of linear equations to compute the minimum distance sum.

Our algorithms are $O(n/\log n)$ times faster than the naïve algorithm directly using the solutions for the linear consensus problems, which takes $O(n^3)$ time for three circular strings and $O(n^4)$ time for four circular strings.

2 Preliminaries

In this section, we introduce formal definition of consensus problems for circular strings. We also give a brief overview of a preliminary result on consensus problems for linear strings, and introduce a discrete convolution, which will be extensively used to solve the problems for circular strings.

2.1 Problem Definition

A (linear) string s of length n is a sequence of n symbols from a constant-sized alphabet Σ. Generally, s is represented by $s[0]s[1]\cdots s[n-1]$, where each $s[i]$, $0 \leq i < n$, denotes the ith symbol of s. A circular string S of length n is similar to a linear string of length n except that the first symbol $S[0]$ is wrapped around next to the last symbol $S[n-1]$. From a circular string S of length n, n instances of linear strings can be derived where each instance $S(r)$ is a linear string $S[r]S[(r+1) \mod n]\cdots S[(r+n-1) \mod n]$. We call r the *index* of the instance. For instance, if $S = \texttt{ababc}$, $S(0) = \texttt{ababc}$, $S(1) = \texttt{babca}$, $S(2) = \texttt{abcab}$, and so forth. For two strings x and y, we denote the concatenation of x and y by xy and a tandem repeat xx by x^2.

Let $\mathbb{S} = \{S_1, S_2, \ldots, S_m\}$ be a set of m circular strings of equal length n. We define an *alignment* of $\mathbb{S}$ as a juxtaposition of their instances. For m integer indices $0 \leq \rho_1, \rho_2, \ldots, \rho_m < n$, the alignment $\rho = (\rho_1, \rho_2, \ldots, \rho_m)$ is a juxtaposition of instances $S_1(\rho_1), S_2(\rho_2), \ldots, S_m(\rho_m)$. Because we only consider the Hamming distance, all alignments $(\rho_1 + k \mod n, \rho_2 + k \mod n, \ldots, \rho_m + k \mod n)$ for $0 \leq k < n$ are essentially the same. This implies that we can naturally *fix* one of the indices of instances, and hence we assume that S_1 is fixed, i.e., $\rho_1 = 0$. Therefore, there exist n^{m-1} distinct alignments instead of n^m.

For two linear strings x and y, let $d(x, y)$ denote the Hamming distance between x and y, which is the number of positions i where $x[i] \neq y[i]$. Given an alignment $\rho = (\rho_1, \rho_2, \ldots, \rho_m)$ of $\mathbb{S} = \{S_1, S_2, \ldots, S_m\}$ and a string X, the *distance sum of X for ρ*, denoted by $E(\rho, X)$, is defined as $\sum_{1 \leq i \leq m} d(X, S_i(\rho_i))$. We also define the *radius of X for ρ*, denoted by $R(\rho, X)$, as $\max_{1 \leq i \leq m} d(X, S_i(\rho_i))$. For each alignment ρ, we define $E_{\min}(\rho)$ as the smallest distance sum of any string X' for ρ, i.e., $E_{\min}(\rho) = \min_{X'} E(\rho, X')$. Similarly, let $R_{\min}(\rho)$ be the smallest radius of any string X' for ρ, i.e., $R_{\min}(\rho) = \min_{X'} R(\rho, X')$. For $E_{\min}(\rho)$ and $R_{\min}(\rho)$, we will omit the alignment notation ρ if no confusion arises. For any alignment ρ' and strings X', we define $E_{\text{opt}} = \min_{\rho', X'} E(\rho', X')$ and $R_{\text{opt}} = \min_{\rho', X'} R(\rho', X')$.

Now we formally define optimal consensus problem and bounded consensus problem on circular strings as follows.

Problem 1. Optimal consensus
Given a set $\mathbb{S} = \{S_1, \ldots, S_m\}$ of m circular strings of length n, find an optimal alignment ρ and a string X (if any) that satisfy:

Problem CS $E(\rho, X) = E_{\text{opt}}$.
Problem CR $R(\rho, X) = R_{\text{opt}}$.
Problem CSR $E(\rho, X) = E_{\text{opt}}$ and $R(\rho, X) = R_{\text{opt}}$.

If such an optimal alignment and consensus string exist, the string can be accepted as an optimal consensus string of $\mathbb{S}$. However, sometimes such ρ and X do not exist, and then, an alignment and a string satisfying weaker conditions may be sought for as follows.

Problem 2. Bounded consensus
Given a set $\mathbb{S} = \{S_1, \ldots, S_m\}$ of m circular strings of length n and two integers $s > 0$ and $r > 0$, find an alignment ρ and a string X (if any) that satisfy:

Problem BS $E(\rho, X) \leq s$.
Problem BR $R(\rho, X) \leq r$.
Problem BSR $E(\rho, X) \leq s$ and $R(\rho, X) \leq r$.

2.2 Problem CSR for Three Linear Strings

For three linear strings s_1, s_2 and s_3 of equal length n, a consensus string s minimizing both (Hamming) distance sum and radius can be found efficiently [14]. First, each aligned position i is classified into five types as follows:

Type 0 $s_1[i] = s_2[i] = s_3[i]$ (all matches)
Type 1 $s_1[i] \neq s_2[i] = s_3[i]$ ($s_1[i]$ is the minority)
Type 2 $s_2[i] \neq s_1[i] = s_3[i]$ ($s_2[i]$ is the minority)
Type 3 $s_3[i] \neq s_1[i] = s_2[i]$ ($s_3[i]$ is the minority)
Type 4 $s_1[i] \neq s_2[i]$, $s_1[i] \neq s_3[i]$, $s_2[i] \neq s_3[i]$ (all mismatches)

Then, we count the number of positions that belong to type i as c_i, for each $i = 0, \ldots, 4$. The crux of [14] is that those counters are used to determine the minimum possible distance sum $E_{\min}$ and radius $R_{\min}$. The authors showed that $E_{\min} = c_1 + c_2 + c_3 + 2c_4$ and $R_{\min} = \max(L_1, L_2)$ where $L_1 = \lceil \max_{i \neq j} d(s_i, s_j)/2 \rceil$ and $L_2 = \lceil E_{\min}/3 \rceil$.

Once the number of each type has been counted, the algorithm in [14] finds a consensus string s with $E_{\min}$ and $R_{\min}$ if it exists. Since it scans the whole strings once and computes the above counters, the algorithm runs in $O(n)$ time.

Lemma 1 (Amir et al. [14]). *For three linear strings of length n, if we are given counters c_1 to c_4 defined as above,*

1. *$E_{\min}$ and $R_{\min}$ are computed in $O(1)$ time.*
2. *The existence of a consensus minimizing both $E_{\min}$ and $R_{\min}$ are determined in $O(1)$ time.*
3. *We can construct such a consensus in $O(n)$ time.*

2.3 Convolution Method for Counting Matches/Mismatches

A discrete convolution is defined as follows.

Definition 1. *Let t and p be arrays of integers, whose lengths are n and m, respectively. The* discrete convolution *of t and p, denoted by $t \otimes p$, is defined as the array of all inner products of p and a sub-array of t, where:*

$$(t \otimes p)[i] = \sum_{j=0}^{m-1} t[i+j]p[j] \quad \text{for } i = 0, \ldots, n-m \ .$$

It is obvious that n elements in the array of $t \otimes p$ can be computed in $O(nm)$ time. However, the convolution can be computed in $O(n \log m)$ time by using the fast Fourier transform (FFT). The reader may refer, for example, to [18].

The discrete convolution has been widely used in various string matching problems, where we match a pattern and a text at every text position [19,20,21]. In this paper, we use convolutions to count the number of matches or mismatches between every pair of instances of two circular strings. For this purpose, we need some definitions first. Given a string x of length n, we define a bit mask $B_{x,\sigma}$ and an inverse bit mask $\overline{B}_{x,\sigma}$ for each symbol $\sigma \in \Sigma$ and $i = 0, \ldots, n-1$,

$$B_{x,\sigma}[i] = \begin{cases} 1, & \text{if } x[i] = \sigma \\ 0, & \text{if } x[i] \neq \sigma \end{cases} \quad \text{and} \quad \overline{B}_{x,\sigma}[i] = \begin{cases} 1, & \text{if } x[i] \neq \sigma \\ 0, & \text{if } x[i] = \sigma \end{cases} \ .$$

For two strings x and y of equal length n and $\sigma \in \Sigma$, the inner product of $B_{x,\sigma}$ and $B_{y,\sigma}$, $\sum_{i=0}^{n-1} B_{x,\sigma}[i]B_{y,\sigma}[i]$, equals the number of positions i where $x[i] = y[i] = \sigma$. Thus, the sum of all the inner products over all σ, $\sum_{\sigma} \sum_{i=0}^{n-1} B_{x,\sigma}[i]B_{y,\sigma}[i]$ gives the number of matches between x and y. Likewise, we can compute the number of

mismatches between x and y, i.e., Hamming distance $d(x, y)$, by summing up all the inner products of $B_{x,\sigma}$ and $\overline{B}_{y,\sigma}$ over all σ.

We can further generalize this method to compute distance between two strings with *don't care* (or *wildcard*) symbols. Assume that a don't care symbol $\$ \notin \Sigma$ matches any symbol $\sigma \in \Sigma \cup \{\$\}$. Then, we can compute $d(x, y)$ by simply setting $B_{x,\sigma}[i] = 0$ if $x[i] = \$$ and $\overline{B}_{y,\sigma}[i] = 0$ if $y[i] = \$$, for $i = 0, \ldots, n-1$.

We now consider convolution of such bit masks. For two circular strings X and Y, convolution $B_{X(0)^2,\sigma} \otimes B_{Y(0),\sigma}$ is an array where the number in each position $i = 0, \ldots, n-1$ is the inner product of $B_{X(i),\sigma}$ and $B_{Y(0),\sigma}$ (or equivalently, $B_{X(0),\sigma}$ and $B_{Y(n-i),\sigma}$). Thus, if we compute such convolutions for each $\sigma \in \Sigma$ and add the results, we get the total number of matches for every alignment of two circular strings. Since discrete convolution $B_{X^2(0),\sigma} \otimes B_{Y(0),\sigma}$ can be computed in $O(n \log n)$ time by using FFT, the total time for counting all matches is $O(|\Sigma| n \log n)$.

Lemma 2. *Given two circular strings X and Y of equal length n over a constant-sized alphabet, the numbers of matches or mismatches in all n alignments can be computed in $O(n \log n)$ time, even with the presence of don't care symbols.*

3 Algorithms

In this section, we first describe an $O(n^2 \log n)$-time algorithm to solve Problem CS for three circular strings. Also, we present an $O(n^2 \log n)$-time algorithm that solves Problem CSR for three circular strings. Then, we show how to solve Problem CS for four circular strings in $O(n^3 \log n)$ time.

3.1 Problem CS for Three Circular Strings

Assume that we are given a set $\mathbb{S} = \{S_1, S_2, S_3\}$ of three circular strings of length n. One obvious way to find a consensus minimizing the distance sum of $\mathbb{S}$ is to compute $E_{\min}$ separately for each of n^2 alignments of $\mathbb{S}$, and to construct a consensus from an alignment with the smallest $E_{\min}$ among all n^2 values. Clearly, this naïve approach takes $O(n^3)$ time, since it requires $O(n)$ time for each alignment.

In contrast, our algorithm solves Problem CS for $\mathbb{S}$ in overall $O(n^2 \log n)$ time. The crux of the algorithm lies in the use of FFT, which enables us to compute n values of $E_{\min}$'s for every n alignments together in $O(n \log n)$ time, and hence to achieve a speedup factor of $O(n/\log n)$ over the naïve approach.

Let δ and γ be integer indices ranging from 0 to $n-1$. Our algorithm consists of two stages: (a) We first superpose two instances of S_1 and S_2 into one string with displacement of δ, and compute convolutions to overlay an instance of S_3 at every position γ of the superposed string. By doing this, we equivalently evaluate $E_{\min}$ for all alignments of $\mathbb{S}$. (b) Among all n^2 alignments, we find the best alignment with the smallest $E_{\min}$, and construct a consensus of the alignment by simply selecting the majority symbol at each aligned position. By definition, the corresponding circular string becomes the consensus circular string of $\mathbb{S}$.

We now describe the algorithm in detail. Consider an n-by-n table E, where we store the value of $E_{\min}(\rho)$ for each alignment $\rho = (0, \delta, \gamma)$ in entry $\mathsf{E}[\delta, \gamma]$. To compute a row of n entries $\mathsf{E}[\delta, 0 : n-1]$ altogether, we define the δth superposition Z_δ of S_1 and S_2 as follows:

Definition 2 (The δth superposition). *For $i = 0, \ldots, n-1$, we define $Z_\delta \in (\Sigma \times \Sigma)^*$ as $Z_\delta[i] = (\sigma_1, \sigma_2)$, where $\sigma_1 = S_1(0)[i]$ and $\sigma_2 = S_2(\delta)[i]$.*

We also define a bit mask of Z_δ^2, the concatenation of two Z_δ's as follows: For each $\sigma \in \Sigma$ and $i = 0, \ldots, 2n-1$,

$$B_{Z_\delta^2,\sigma}[i] = \begin{cases} 1, & \text{if } Z_\delta[i \mod n] = (\sigma, *) \text{ or } (*, \sigma), \\ 0, & \text{otherwise.} \end{cases}$$

The following lemma shows correctness of computation of a single row of $\mathsf{E}[\delta, 0 : n-1]$ derived from convolutions.

Lemma 3. *Given Z_δ and S_3, the following equation holds for $\gamma = 0, \ldots, n-1$:*

$$\mathsf{E}[\delta, \gamma] = \sum_{\sigma \in \Sigma} (B_{Z_\delta^2,\sigma} \otimes \overline{B}_{S_3(0),\sigma})[n - \gamma] \ . \tag{1}$$

Proof. By definition of discrete convolution, the right-hand side of (1) is

$$\sum_{\sigma \in \Sigma} (B_{Z_\delta^2,\sigma} \otimes \overline{B}_{S_3(0),\sigma})[n - \gamma] = \sum_{\sigma \in \Sigma} \sum_{j=0}^{n-1} B_{Z_\delta^2,\sigma}[n - \gamma + j] \overline{B}_{S_3(0),\sigma}[j] \ . \tag{2}$$

Using the relations $S_3(0)[j] = S_3(\gamma)[(j - \gamma) \mod n]$ and $B_{Z_\delta^2}[n + j] = B_{Z_\delta^2}[j]$ gives

$$\sum_{\sigma \in \Sigma} \sum_{j=0}^{n-1} B_{Z_\delta^2,\sigma}[n - \gamma + j] \overline{B}_{S_3(0),\sigma}[j] = \sum_{\sigma \in \Sigma} \sum_{j=0}^{n-1} B_{Z_\delta^2,\sigma}[n - \gamma + j] \overline{B}_{S_3(\gamma),\sigma}[(j - \gamma) \mod n] \tag{3}$$

$$= \sum_{\sigma \in \Sigma} \sum_{j=0}^{n-1} B_{Z_\delta^2,\sigma}[n + j] \overline{B}_{S_3(\gamma),\sigma}[j \mod n] \tag{4}$$

$$= \sum_{\sigma \in \Sigma} \sum_{j=0}^{n-1} B_{Z_\delta^2,\sigma}[j] \overline{B}_{S_3(\gamma),\sigma}[j] \ . \tag{5}$$

It remains to show that the summation of all the inner products over all σ in (5) yields $\mathsf{E}[\delta, \gamma]$. First, we have

$$\sum_{\sigma \in \Sigma} \sum_{j=0}^{n-1} B_{Z_\delta^2,\sigma}[j] \overline{B}_{S_3(\gamma),\sigma}[j] = \sum_{j=0}^{n-1} \sum_{\sigma \in \Sigma} B_{Z_\delta^2,\sigma}[j] \overline{B}_{S_3(\gamma),\sigma}[j] \ . \tag{6}$$

Algorithm 1. Algorithm for Problem CS for three circular strings

1: **for** $\delta = 0$ to $n-1$ **do**
2: Compute superposition Z_δ and bit mask $B_{Z_\delta^2,\sigma}$ for all $\sigma \in \Sigma$. // $O(n)$ time
3: Compute $B_{Z_\delta^2,\sigma} \otimes \overline{B}_{S_3(0),\sigma}$ for all $\sigma \in \Sigma$ and add element by element the resulting arrays. // $O(n \log n)$ time
4: Take a reverse of the resulting array to compute $\mathsf{E}[\delta, 0 : n-1]$. // $O(n)$ time
5: **end for**
6: Find δ^* and γ^* where $\mathsf{E}[\delta^*, \gamma^*]$ is the minimum. // $O(n^2)$ time
7: Compute a consensus string of $S_1(0)$, $S_2(\delta^*)$, and $S_3(\gamma^*)$ // $O(n)$ time

Let $s_1 = S_1(0)$, $s_2 = S_2(\delta)$, and $s_3 = S_3(\gamma)$. For each aligned position j and $\sigma \in \Sigma$, it is straightforward to show that

$$B_{Z_\delta^2,\sigma}[j]\overline{B}_{S_3(\gamma),\sigma}[j] = \begin{cases} 1, & \text{if } (s_1[j] = \sigma \text{ or } s_2[j] = \sigma) \text{ and } s_3[j] \neq \sigma, \\ 0, & \text{otherwise.} \end{cases}$$

Then, it follows that the sum $\sum_{\sigma \in \Sigma} B_{Z_\delta^2,\sigma}[j]\overline{B}_{S_3(\gamma),\sigma}[j]$ becomes zero for each position of type 0, one for each position of types 1 to 3, and two for each position of type 4. Therefore, adding up these sums over all j's gives $c_1 + c_2 + c_3 + 2c_4$, which equals $E_{\min}$ of three strings s_1, s_2,and s_3, that is in turn $\mathsf{E}[\delta, \gamma]$. □

Algorithm 1 shows how we solve Problem CS for three circular strings. Since it requires at most $O(n \log n)$ time per iteration (see line 3), Algorithm 1 takes $O(n^2 \log n)$ time in total. In terms of space complexity, instead of maintaining entire table E, we only keep the best alignment $(0, \delta^*, \gamma^*)$ with the smallest $E_{\min}$ seen so far to accommodate in $O(n)$ space. Therefore, we get the following theorem.

Theorem 1. *Problem CS for three circular strings can be solved in $O(n^2 \log n)$ time and $O(n)$ space.*

Remark. If the size of alphabet Σ is not constant, the running time in Theorem 1 is multiplied by $|\Sigma|$, which is also applied to Theorems 2 and 3.

3.2 Problem CSR for Three Circular Strings

We now proceed to Problem CSR for three circular strings. Recall that Lemma 1 implies that we can determine the existence of an optimal consensus in $O(1)$ time if we know all counters c_1 to c_4. Algorithm 1 directly computes $E_{\min}$ for each alignment, but it does not compute any of counters, c_1 to c_4, which are required for applying Lemma 1 to our problem. In this subsection, we propose an algorithm that explicitly computes all counters for each alignment, which in turn enables us to solve Problem CSR for three circular strings. The key idea is that we can count c_i's for every alignment efficiently using the convolution and the system of linear equations. Then, we can find an optimal alignment by applying the algorithm in [14] to each alignment.

Lemma 4. *For three linear strings s_1, s_2, and s_3, given c_0 and $d(s_i, s_j)$'s for all $i \neq j \in \{1, 2, 3\}$, the following system of linear equations holds for c_1, c_2, c_3, and c_4.*

$$c_1 + c_2 + c_3 + c_4 = n - c_0,$$
$$c_1 + c_2 + c_4 = d(s_1, s_2), \quad c_1 + c_3 + c_4 = d(s_1, s_3), \quad c_2 + c_3 + c_4 = d(s_2, s_3) \ .$$

Proof. Since each position belongs to exactly one of five types, $c_0 + c_1 + c_2 + c_3 + c_4 = n$. By definition, $d(s_1, s_2)$ is the number of positions i where $s_1[i] \neq s_2[i]$, which occurs in one of the three distinct cases: (i) $s_1[i]$ is the minority (Type 1), (ii) $s_2[i]$ is the minority (Type 2), or (iii) all three aligned symbols are distinct (Type 4). It implies that $c_1 + c_2 + c_4 = d(s_1, s_2)$, and similar arguments can be applied to the other cases for $d(s_1, s_3)$ and $d(s_2, s_3)$. □

The solution of the system of linear equations in Lemma 4 leads to the following corollary.

Corollary 1. *Once c_0 and the pairwise Hamming distances $d(s_i, s_j)$ for all $i < j \in \{1, 2, 3\}$ have been computed, we can compute c_1 to c_4 as follows:*

$$c_1 = n - d(s_2, s_3) - c_0, \quad c_2 = n - d(s_1, s_3) - c_0, \quad c_3 = n - d(s_1, s_2) - c_0,$$
$$c_4 = d(s_1, s_2) + d(s_1, s_3) + d(s_2, s_3) - 2c_0 - 2n \ .$$

The above corollary can be easily extended to the case of three circular strings as follows: We store each counter c_i for all $i = 0, \ldots, 4$ into an n-by-n table $\mathsf{C}_i[0 : n-1, 0 : n-1]$, in which each entry $\mathsf{C}_i[\delta, \gamma]$ contains counter c_i for an alignment $(0, \delta, \gamma)$. Suppose that we have computed table C_0 and three arrays of size n, D_{12}, D_{23}, and D_{13}, which contains pairwise Hamming distances $\mathsf{D}_{12}[\delta] = d(S_1(0), S_2(\delta))$, $\mathsf{D}_{23}[\delta] = d(S_2(0), S_3(\delta))$, and $\mathsf{D}_{13}[\delta] = d(S_1(0), S_3(\delta))$ for $\delta = 0, \ldots, n-1$. Then, we have the following lemma.

Lemma 5. *For $0 \leq \delta < n$ and $0 \leq \gamma < n$,*

$$\begin{aligned}
\mathsf{C}_1[\delta, \gamma] &= n - \mathsf{D}_{23}[(\gamma - \delta) \mod n] - \mathsf{C}_0[\delta, \gamma], \\
\mathsf{C}_2[\delta, \gamma] &= n - \mathsf{D}_{13}[\gamma] - \mathsf{C}_0[\delta, \gamma], \\
\mathsf{C}_3[\delta, \gamma] &= n - \mathsf{D}_{12}[\delta] - \mathsf{C}_0[\delta, \gamma], \\
\mathsf{C}_4[\delta, \gamma] &= \mathsf{D}_{12}[\delta] + \mathsf{D}_{23}[(\gamma - \delta) \mod n] + \mathsf{D}_{13}[\gamma] - 2\mathsf{C}_0[\delta, \gamma] - 2n \ .
\end{aligned}$$

The basic idea of our algorithm is to compute C_0 and D_{ij}'s for all n^2 possible alignments efficiently. By Lemma 2, all values of D_{ij}'s can be computed in time $O(n \log n)$ by using FFT. Thus, if we compute $\mathsf{C}_0[\delta, \gamma]$ for all $0 \leq \delta, \gamma < n$, we can get $\mathsf{C}_1[\delta, \gamma]$ to $\mathsf{C}_4[\delta, \gamma]$ simultaneously by following the above lemma. For this purpose, we define the δth intersection I_δ of S_1 and S_2 as follows: Given an integer index $0 \leq \delta \leq n - 1$,

Definition 3 (The δth intersection). *For $i = 0, \ldots, n-1$, we define $I_\delta \in (\Sigma \cup \{\#\})^*$ as*

$$I_\delta[i] = \begin{cases} S_1(0)[i], & \text{if } S_1(0)[i] = S_2(\delta)[i], \\ \#, & \text{if } S_1(0)[i] \neq S_2(\delta)[i], \end{cases}$$

Algorithm 2. Algorithm for Problem CSR for three circular strings

1: Compute $\mathsf{D}_{12}[0:n-1]$, $\mathsf{D}_{23}[0:n-1]$, $\mathsf{D}_{13}[0:n-1]$ using FFT // $O(n \log n)$ time
2: **for** $\delta = 0$ to $n-1$ **do**
3: Compute intersection I_δ of $S_1(0)$ and $S_2(\delta)$ // $O(n)$ time
4: Compute the number of matches between I_δ and $S_3(\gamma)$ for all $0 \leq \gamma < n$ using FFT // $O(n \log n)$ time
5: Take a reverse of the resulting array to compute $\mathsf{C}_0[\delta, 0:n-1]$ // $O(n)$ time
6: **end for**
7: **for all** (δ, γ) **do**
8: Compute all $\mathsf{C}_i[\delta, \gamma]$'s by using Lemma 5, and determine $E_{\min}$ and $R_{\min}$ for alignment $(0, \delta, \gamma)$.
9: **end for**
10: Find the optimal alignment $(0, \delta, \gamma)$ minimizing $E_{\min}$ and $R_{\min}$ of $S_1(0)$, $S_2(\delta)$, and $S_3(\gamma)$

where $\# \notin \Sigma$ denotes a mismatch symbol, which does not match any symbol in Σ.

Then, the number of matches between I_δ and $S_3(\gamma)$ equals $\mathsf{C}_0[\delta, \gamma]$. The first loop in Algorithm 2 computes $\mathsf{C}_0[\delta, \gamma]$ for all $0 \leq \gamma < n$ together in $O(n \log n)$ time for given δ by counting the number of matches between I_δ and $S_3(\gamma)$. This is done by computing convolutions between $B_{I_\delta^2, \sigma}$ and $B_{S_3(0), \sigma}$ for $\sigma \in \Sigma$. Hence, we get the following theorem.

Theorem 2. *Problem CSR for three circular strings can be solved in $O(n^2 \log n)$ time and $O(n)$ space.*

3.3 Problem CS for Four Circular Strings

We now describe how to compute a consensus minimizing distance sum of $\mathbb{S} = \{S_1, S_2, S_3, S_4\}$.

Basically, we take an approach similar to that for three circular strings, however we need to account for a more number of types of aligned positions in the case of four strings. Given four linear strings s_1, s_2, s_3, and s_4, we classify each aligned position i into five types. (We assume that j_1, j_2, j_3, and j_4 is a permutation of $\{1, 2, 3, 4\}$.)

Type A $s_1[i] = s_2[i] = s_3[i] = s_4[i]$ (All matches)
Type B $s_{j_1}[i] = s_{j_2}[i] = s_{j_3}[i] \neq s_{j_4}[i]$ (One of symbols is the minority, while the others are the same)
Type C $s_{j_1}[i] = s_{j_2}[i], s_{j_3}[i] = s_{j_4}[i], s_{j_1}[i] \neq s_{j_3}[i]$ (Two distinct pairs of the same symbols)
Type D $s_{j_1}[i] = s_{j_2}[i], s_{j_1}[i] \neq s_{j_3}[i], s_{j_1}[i] \neq s_{j_4}[i], s_{j_3}[i] \neq s_{j_4}[i]$ (Two of symbols are the same, while the others are distinct)
Type E $s_j[i] \neq s_{j'}[i]$ for all $j \neq j' \in \{1, 2, 3, 4\}$ (All mismatches)

Table 1. Types and counters for aligned positions of four strings

	Type A	Type B			Type C			Type D						Type E
s_1	♣	♣	♢	♢	♣	♢	♢	♣	♣	♣	♢	♢	♢	♣
s_2	♣	♢	♣	♢	♢	♣	♢	♣	♢	♢	♣	♣	♡	♢
s_3	♣	♢	♢	♣	♢	♢	♣	♢	♣	♡	♣	♡	♣	♡
s_4	♣	♢	♢	♣	♣	♣	♣	♡	♡	♣	♡	♣	♣	♠
Counter	a	b_1	b_2	b_3	b_4	c_1	c_2	c_3	d_1	d_2	d_3	d_4	d_5	d_6	e

Table 1 depicts types and counters we consider. Four symbols ♣, ♢, ♡, and ♠ represent distinct symbols in each aligned position, and the same symbols in a single column represents matches among corresponding strings in that position. We count each type of position in all n positions as in Table 1. Note that we divide Types B, C, and D further with respect to combinations of distinct symbols.

For linear strings, we can construct a consensus string with the minimum distance sum $E_{\min} = \sum_i b_i + 2(\sum_i c_i + \sum_i d_i) + 3e$. This is easily done in $O(n)$ time by choosing the majority symbol in each aligned position, while we scan the whole strings. For circular strings, a naïve algorithm takes $O(n^4)$ time since there are n^3 possible alignments of four circular strings. However, we can do better in $O(n^3 \log n)$ time as in the case of three circular strings.

We basically compute all fifteen counters by setting and solving a system of linear equations for them. It is quite straightforward to derive twelve equations from the numbers of pairwise, triple-wise, or quadruple-wise matches among four strings. Still, we require three more equations to have the solution for fifteen counters and those equations are derived from distance between two pairs of strings defined as follows.

For $1 \leq i < j \leq 4$, we define a string s_{ij} from a pair of strings s_i and s_j, in which, for $k = 0, \ldots, n-1$, $s_{ij}[k] = s_i[k]$ if $s_i[k] = s_j[k]$; otherwise, $s_{ij}[k] = \$$, where $\$$ is a don't care symbol. Then, it is easily shown that Hamming distance $d(s_{12}, s_{34})$ equals c_3. Likewise, $c_1 = d(s_{14}, s_{23})$ and $c_2 = d(s_{13}, s_{24})$. Now we finally have the following system of linear equations for all fifteen counters.

Lemma 6. *The following system of linear equations holds for the counters of four strings.*

$$
\begin{gathered}
a + b_3 + b_4 + c_3 + d_1 = M_{12},\ a + b_2 + b_4 + c_2 + d_2 = M_{13},\\
a + b_2 + b_3 + c_1 + d_3 = M_{14},\ a + b_1 + b_4 + c_1 + d_4 = M_{23},\\
a + b_1 + b_3 + c_2 + d_5 = M_{24},\ a + b_1 + b_2 + c_3 + d_6 = M_{34},\\
a + b_4 = M_{123},\quad a + b_3 = M_{124},\quad a + b_2 = M_{134},\quad a + b_1 = M_{234},\\
a = M_{1234},\\
c_3 = d(s_{12}, s_{34}),\quad c_2 = d(s_{13}, s_{24}),\quad c_1 = d(s_{14}, s_{23}),\\
a + b_1 + b_2 + b_3 + b_4 + c_1 + c_2 + c_3 + d_1 + d_2 + d_3 + d_4 + d_5 + d_6 + e = n,
\end{gathered}
$$

where M_{ij}, M_{ijk}, and M_{ijkl} denote the numbers of pairwise, triple-wise, and quadruple-wise matches, respectively, among S_i, S_j, S_k and S_l.

All six M_{ij}'s and four M_{ijk}'s for every alignment can be computed in $O(n^2)$ and $O(n^3)$ time, respectively. All M_{1234}'s for every alignment can be computed in $O(n^3 \log n)$ time by using the intersection and the convolution in the similar way as explained in Sect. 3.2. To compute $d(s_{ij}, s_{kl})$'s for each alignment, we construct s_{ij} for each alignment of $S_i(0)$ and $S_j(\delta)$, and we also compute all Hamming distances between n possible s_{ij}'s and n possible s_{kl}'s in $O(n^3 \log n)$ time by using FFTs. Then, using the solutions of the equations in Lemma 6, we compute all counters and $E_{\min}$ in $O(1)$ time for each of n^3 alignments. Finally, we can find an optimal alignment and consensus in $O(n)$ time.

Theorem 3. *Problem CS for four circular strings can be solved in $O(n^3 \log n)$ time and $O(n)$ space.*

References

1. Gusfield, D.: Algorithms on Strings, Tree, and Sequences. Cambridge University Press, Cambridge (1997)
2. Frances, M., Litman, A.: On covering problems of codes. Theory of Computing Systems 30(2), 113–119 (1997)
3. Ben-Dor, A., Lancia, G., Perone, J., Ravi, R.: Banishing bias from consensus sequences. In: Hein, J., Apostolico, A. (eds.) CPM 1997. LNCS, vol. 1264, pp.247–261. Springer, Heidelberg (1997)
4. Gasieniec, L., Jansson, J., Lingas, A.: Approximation algorithms for Hamming clustering problems. Journal of Discrete Algorithms 2(2), 289–301 (2004)
5. Lanctot, K., Li, M., Ma, B., Wang, S., Zhang, L.: Distinguishing string selection problems. In: Proceedings of the 10th ACM-SIAM Symposium on Discrete Algorithms, pp. 633–642 (1999)
6. Li, M., Ma, B., Wang, L.: Finding similar regions in many strings. In: Proceedings of the 31st Annual ACM Symposium on Theory of Computing, pp. 473–482 (1999)
7. Li, M., Ma, B., Wang, L.: On the closest string and substring problems. Journal of the ACM 49(2), 157–171 (2002)
8. Ma, B., Sun, X.: More efficient algorithms for closest string and substring problems. In: Vingron, M., Wong, L. (eds.) RECOMB 2008. LNCS (LNBI), vol. 4955, pp. 396–409. Springer, Heidelberg (2008)
9. Gramm, J., Niedermeier, R., Rossmanith, P.: Exact solutions for closest string and related problems. In: Eades, P., Takaoka, T. (eds.) ISAAC 2001. LNCS, vol. 2223, pp. 441–453. Springer, Heidelberg (2001)
10. Gramm, J., Niedermeier, R., Rossmanith, P.: Fixed-parameter algorithms for closest string and related problems. Algorithmica 37(1), 25–42 (2003)
11. Stojanovic, N., Berman, P., Gumucio, D., Hardison, R., Miller, W.: A linear-time algorithm for the 1-mismatch problem. In: Rau-Chaplin, A., Dehne, F., Sack, J.-R., Tamassia, R. (eds.) WADS 1997. LNCS, vol. 1272, pp. 126–135. Springer, Heidelberg (1997)
12. Sze, S., Lu, S., Chen, J.: Integrating sample-driven and pattern-driven approaches in motif finding. In: Jonassen, I., Kim, J. (eds.) WABI 2004. LNCS (LNBI), vol. 3240, pp. 438–449. Springer, Heidelberg (2004)
13. Boucher, C., Brown, D., Durocher, S.: On the structure of small motif recognition instances. In: Amir, A., Turpin, A., Moffat, A. (eds.) SPIRE 2008. LNCS, vol. 5280, pp. 269–281. Springer, Heidelberg (2008)

14. Amir, A., Landau, G.M., Na, J.C., Park, H., Park, K., Sim, J.S.: Consensus optimizing both distance sum and radius. In: Karlgren, J., Tarhio, J., Hyyrö, H. (eds.) SPIRE 2009. LNCS, vol. 5721, pp. 234–242. Springer, Heidelberg (2009)
15. Mosig, A., Hofacker, I., Stadler, P.: Comparative analysis of cyclic sequences: Viroids and other small circular RNAs. Lecture Notes in Informatics, vol. P-83, pp. 93–102 (2006)
16. Fernandes, F., Pereira, L., Freitas, A.: CSA: An efficient algorithm to improve circular DNA multiple alignment. BMC Bioinformatics 10(1), 230 (2009)
17. Thompson, J., Higgins, D., Gibson, T.: CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. Nucleic Acids Research 22, 4673–4680 (1994)
18. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 2nd edn. The MIT Press, Cambridge (2001)
19. Fischer, M.J., Paterson, M.S.: String matching and other products. In: Karp, R.M. (ed.) Complexity of Computation. SIAM-AMS Proceedings, pp. 113–125 (1974)
20. Abrahamson, K.: Generalized string matching. SIAM J. Comput. 16(6), 1039–1051 (1987)
21. Amir, A., Lewenstein, M., Porat, E.: Faster algorithms for string matching with k mismatches. In: SODA 2000: Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms, Philadelphia, PA, USA, pp. 794–803. Society for Industrial and Applied Mathematics (2000)

Optimizing Restriction Site Placement for Synthetic Genomes

Pablo Montes[1,⋆], Heraldo Memelli[1], Charles Ward[1,⋆⋆], Joondong Kim[2,⋆⋆⋆], Joseph S.B. Mitchell[2,⋆⋆⋆], and Steven Skiena[1,⋆⋆]

[1] Department of Computer Science
Stony Brook University
Stony Brook, NY 11794
{pmontes,hmemelli,charles,skiena}@cs.sunysb.edu
[2] Department of Applied Mathematics and Statistics
Stony Brook University
Stony Brook, NY 11794
{jdkim,jsbm}@ams.sunysb.edu

Abstract. Restriction enzymes are the workhorses of molecular biology. We introduce a new problem that arises in the course of our project to design virus variants to serve as potential vaccines: we wish to modify virus-length genomes to introduce large numbers of unique restriction enzyme recognition sites while preserving wild-type function by substitution of synonymous codons. We show that the resulting problem is NP-Complete, give an exponential-time algorithm, and propose effective heuristics, which we show give excellent results for five sample viral genomes. Our resulting modified genomes have several times more unique restriction sites and reduce the maximum gap between adjacent sites by three to nine-fold.

Keywords: Synthetic biology, restriction enyzme placement, genome refactoring.

1 Introduction

An exciting new field of *synthetic biology* is emerging with the goal of designing novel living organisms at the genetic level. DNA sequencing technology can be thought of as reading DNA molecules, so as to describe them as strings on $\{ACGT\}$ for computational analysis. DNA *synthesis* is the inverse operation, where one can take any such string and construct DNA molecules to specification with exactly the given sequence. Indeed, commercial vendors such as GeneArt

⋆ On leave from and supported in part by Politécnico Grancolombiano, Bogotá, Colombia.

⋆⋆ Supported in part by NIH Grant 5R01AI07521903, NSF Grant DBI-0444815, and IC Postdoctoral Fellowship HM1582-07-BAA-0005.

⋆⋆⋆ Partially supported by grants from the National Science Foundation (CCF-0729019), Metron Aviation, and NASA Ames.

A. Amir and L. Parida (Eds.): CPM 2010, LNCS 6129, pp. 323–337, 2010.

(`http://www.geneart.com`) and Blue Heron (`http://www.blueheronbio.com`) today charge under 60 cents per base to synthesize virus-length sequences, and prices are rapidly dropping [1,2]. The advent of cheap synthesis will have many exciting new applications throughout the life sciences: the need to design new sequences to specification leads to a variety of new algorithmic problems on sequences.

In this paper, we introduce a new problem that arises in the course of our project to design virus variants to serve as potential vaccines [3,4]. In particular, *restriction enzymes* are laboratory reagents that cut DNA at specific patterns. For example, the enzyme EcoRI cuts at the pattern $GAATTC$. Each enzyme cuts at a particular pattern, and over 3000 restriction enzymes have been studied in detail, with more than 600 of these being available commercially [5].

Each occurrence of a pattern within a given DNA target sequence is called a *restriction enzyme recognition site* or *restriction site*. Unique restriction sites within a given target are particularly prized, as they cut the sequence unambiguously in exactly one place. Many techniques for manipulating DNA make use of unique restriction sites [6,7]. In particular, *subcloning* is an important method of inserting a new sequence between two different unique restriction sites.

Thus a genomic sequence which contains unique restriction sites at regular intervals will be easy to manipulate in the laboratory. Traditionally, DNA sequences manipulated in laboratories were from living organisms, so the experimenter had no choice but to work with what they were given. But low-cost, large-scale DNA synthesis changes this equation.

Refactoring [8] is a software engineering term for redesigning a program to improve its internal structure for better ease of maintenance while leaving its external behavior unchanged. Genome synthesis technology enables us to refactor biological organisms: we seek to restructure the genome of an organism into a sequence which is functionally equivalent (meaning behaves the same in its natural environment) while being easier to manipulate.

The redundancy of the genetic code (64 three-base codons coding for 20 distinct amino acids) gives us the freedom to insert new restriction sites at certain places and remove them from others without changing the protein coded for by a given gene. Identifying the locations of both current and potential sites can be done using conventional pattern matching algorithms. Much more challenging is the problem of finding well-spaced unique placements for many different enzymes to facilitate laboratory manipulation of synthesized sequences. Our contributions in this paper are:

- *Problem Definition* – We abstract a new optimization problem on sequences to model this sequence design task: the *Unique Restriction Site Placement Problem* (URSPP). We show this problem is NP-Complete and give approximability results.
- *Algorithm Design* – We present a series of algorithms and heuristics for the Unique Restriction Site Placement Problem. In particular we give an $O(n^2 2^r)$-time dynamic programming algorithm for URSPP, which is practical for designs with small number of enzymes. We also give an efficient

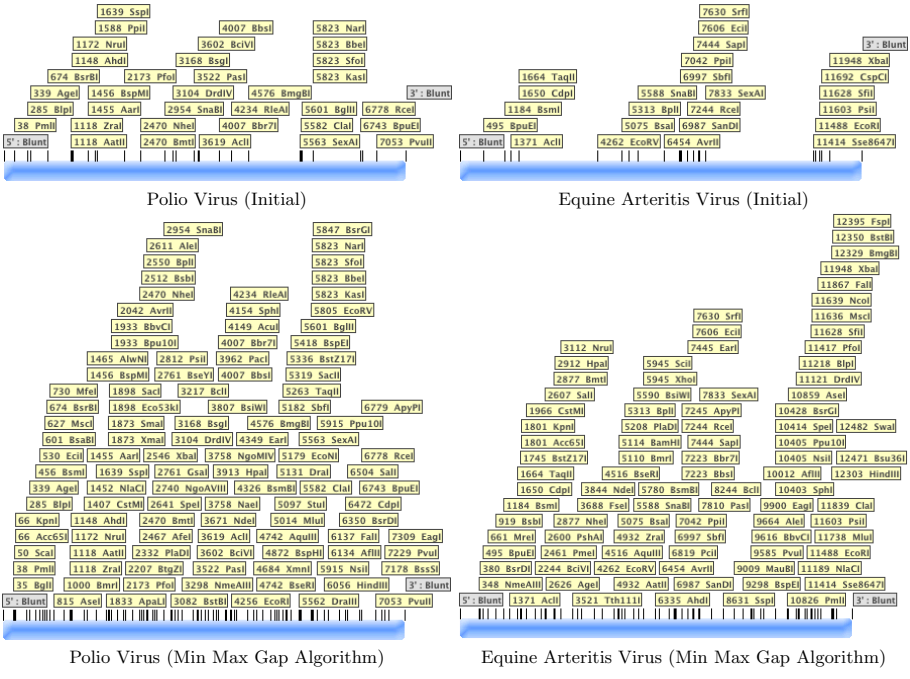


Fig. 1. Visualization of restriction enzymes for Polio Virus and Equine Arteritis Virus. Images were created using Serial Cloner 2.0 [9].

greedy heuristic for site placement, and a heuristic based on weighted bipartite matching which is polynomial in both n and r, both of which construct good designs in practice.

- *Sequence Design Tool* – Our design algorithms have been integrated with the Aho-Corasick pattern matching algorithm to yield a sequence design tool we anticipate will be popular within the synthetic biology community. In particular, we have developed this tool as part of a project underway to design a candidate vaccine for a particular agricultural pathogen.
- *Experimental Results for Synthetic Viruses* – The URSPP problem abstraction to some extent obscures the practical aspects of sequence design. The critical issue is how regularly unique restriction sites can be inserted into the genomes of representative viruses. We perform a series of experiments to demonstrate that impressive numbers of regularly-spaced, unique restriction sites can be engineered into viral genomes.

  Indeed, our system produces genomes with three to four-fold more unique restriction enzymes than a baseline algorithm (details given in the results section) and reduces the maximum gap size between restriction sites three to nine-fold. Figure 1 shows example results for Polio Virus and Equine Arteritis Virus.

This paper is organized as follows. In Section 2 we briefly review related work on genome refactoring and sequence design. In Section 3 we discuss our algorithmic approach to the problem. Finally, in Section 4 we give the results for our refactored viral genomes.

2 Related Work

Synthetic biology is an exciting new field of growing importance. The synthesis of virus-length DNA sequences, a difficult task just a decade ago [10], is now a relatively inexpensive commercialized service. This enables a tremendous number of applications, notably the manipulation of viral genomes to produce attenuated viruses for vaccine production [3,4]. This work is in support of genome refactoring efforts related to this project.

Broadly, the field of genome refactoring seeks to expand our understanding of genetics through the construction of an engineering toolkit to easily modify genomes. Chan et al. [11], for example, refactored the bacteriophage T7 so "that is easier to study, understand, and extend." A number of different tools for genome refactoring exist: GeneJAX [12] is a JavaScript web application CAD tool for genome refactoring. SiteFind is a tool which seeks to introduce a restriction enzyme as part of a point mutation using site-directed mutagenesis [13]. However, SiteFind considers the much more restricted problem of introducing a single restriction site into a short (< 400b) sequence specifically to serve as a marker for successful mutagenesis, in contrast with our efforts to place hundreds of sites in several kilobase genomes.

GeneDesign is a tool which aids the automation of design of synthetic genes [14]. One of its functionalities is the silent insertion of restriction sites. The user can manually choose the enzymes and the sites from the possible places where they can be inserted, or the program can automatically do the insertion. The latter is similar to our tool in trying to automate the creation of restriction sites in the sequence, but the process is done quite differently. GeneDesign only attempts to insert restriction sites of enzymes that do not appear anywhere in the sequence. It follows a simple heuristic to try to space the introduced consecutive sites at an

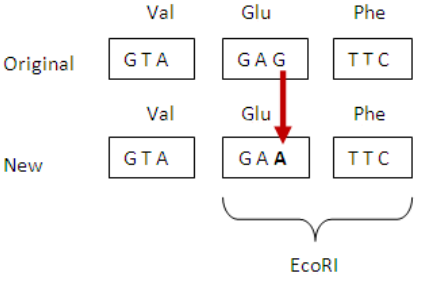


Fig. 2. Introduction of a restriction site by silent mutation

interval specified by the user, without any attempt or guarantee to optimize the process.

Other relevant work includes Skiena [15], which gives an algorithm for optimally removing restriction sites from a given coding sequence. The problem here differs substantially, in that (1) we seek to remove all but one restriction sites per cutter, and (2) we seek to introduce cut sites of unrepresented enzymes in the most advantageous manner.

3 Methodology

3.1 Problem Statement

The primary goal of our system is to take a viral plasmid sequence and, through minimal sequence editing, produce a new plasmid which contains a large number of evenly spaced unique restriction sites. In order to accomplish this, of course, we create and remove restriction sites in the sequence. The primary restrictions on our freedom to edit the sequence are:

- The amino–acid sequence of all genes must be preserved. Each amino–acid in a gene is encoded by a triplet of nucleotides called a codon; as there are more triplets than amino–acids, there are between one and six codons which encode each amino–acid. Codons which encode the same amino–acid are termed synonymous. Thus, in gene-encoding regions we may only make nucleotide changes which change a codon into a synonymous codon.

  Figure 2 shows an example of this concept. Here a single nucleotide change introduces the EcoRI restriction site, without modifying the amino–acid sequence (*GAG* and *GAA* both code for the amino–acid glutamic acid).
- Certain regions of the sequence may not be editable at all. Examples of such regions include known or suspected functional RNA secondary structures. We also lock overlapping regions of multiple open reading frames; very few such regions admit useful synonymous codon changes to multiple ORFs simultaneously.

The problem of finding the optimal placement of the restriction sites is interesting and difficult at the same time. Its difficulty arises from the fact that there are many aspects that the program must keep track off at the same time, and many combinatorial possibilities that have to deal with: the order of considering the enzymes, the decision on which occurrence of a site to keep, the position of inserting enzymes. All these should be done while maintaining the amino–acid sequence and trying to minimize the gaps between consecutive sites.

We define the decision problem version of the *Unique Restriction Site Placement Problem* (URSPP) as follows:

- **Input:** a set of m subsets S_i of integers, each in the range $[1, \ldots, n]$, an integer K.
- **Output:** Does there exist a single element s_i in all S_i such that the maximum gap between adjacent elements of $\{0, n+1, s_1, \ldots, s_m\}$ is at most K?

Here, each subset consists of the existing or potential recognition sites for a specific restriction enzyme. The decision problem corresponds to choosing a single site for each restriction enzyme in such a way that guarantees that adjacent unique restriction sites are no more than K bases apart. The optimization variant of the problem simply seeks to minimize K, the largest gap between adjacent restriction enzymes.

In the following sections, we show that this problem is NP-Complete, and we give an exponential time and space dynamic programming algorithm for this problem. Due to the impracticality of running this algorithm, we also give suboptimal heuristics which do not give optimal results but still give good results which should prove very useful for our biological purpose.

3.2 NP-Completeness and Approximability

Theorem 1. *The decision version of URSPP is NP-complete. Further, the optimization version of URSPP cannot be approximated within factor 3/2 unless P=NP.*

Proof. The decision problem is clearly in NP, as one can readily verify if a specified selection of s_i's has gap at most k.

In order to prove NP-hardness, we use a reduction from Set Cover. Our reduction is illustrated in Figure 3.

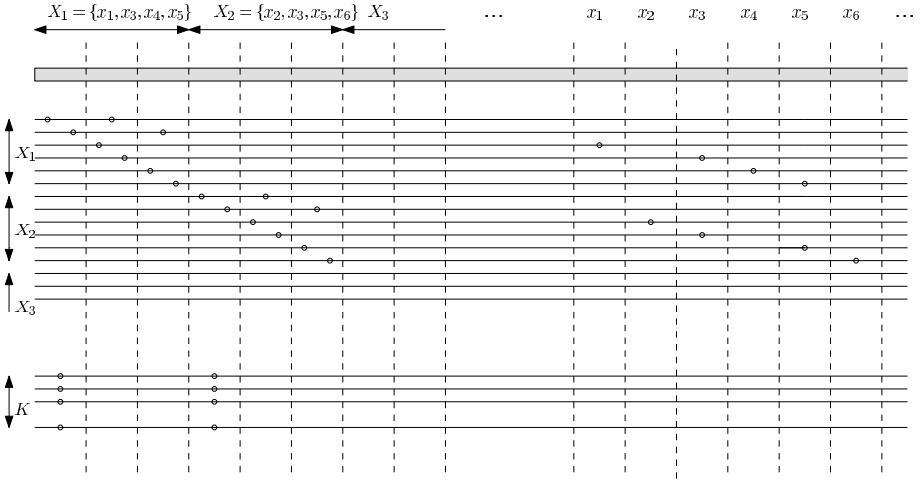


Fig. 3. Illustration of the reduction from Set Cover to URSPP. The shaded bar across the top represents the interval $[0, n = 2k(3M + N)]$. The vertical dashed lines correspond to the singleton sets of values at positions $2k, 4k, 6k, \ldots$. The rows below the shaded bar show the values in the sets S_i associated with $X_1, X_2, X_3, \ldots$, in the order $P_1, Q_1, A_1, B_1, C_1, D_1, P_2, Q_2, A_2, \ldots$. The bottom set of rows correspond to the K copies of the sets $\{k, 7k, 13k, \ldots, (M-1)6k+k\}$. (Not shown are rows corresponding to the singleton sets at positions $2k, 4k, 6k, \ldots$.)

Consider an instance of Set Cover, with universe set $U = \{x_1, x_2, \ldots, x_N\}$ and a collection $\mathcal{C} = \{X_1, X_2, \ldots, X_M\}$ of subsets $X_i \subseteq U$. Also given in the instance is an integer K, with $1 \leq K \leq M$. The problem is to decide if there exists a subset $\mathcal{C}' \subseteq \mathcal{C}$, with $|\mathcal{C}'| \leq K$, such that $\mathcal{C}'$ forms a cover of U: $\bigcup_{X \in \mathcal{C}'} X = U$. We can assume that each $X_i \in \mathcal{C}$ has four elements ($|X_i| = 4$); this special version of Set Cover is also known to be NP-complete [16,17].

Given an instance of Set Cover, we construct an instance of URSPP as follows. First let k be an even positive integer, and we let $n = 2k(3M + N)$.

We specify singleton sets $S_1 = \{2k\}$, $S_2 = \{4k\}$, ..., $S_{3M+N-1} = \{2k(3M + N - 1)\}$. Since each of these sets S_i are singletons, the single points $s_i \in S_i$ will be selected in any solution of URSPP. The resulting set of s_i's, together with the elements 0 and $n = 2k(3M + N)$, specify a set of $3M + N$ *intervals* each of length $2k$: $0, 2k, 4k, 6k, \ldots, 2k(3M + N - 1), 2k(3M + N)$. We refer to the first $3M$ intervals as *set-intervals* (they will be associated with the sets X_i) and the last N intervals as *element-intervals* (they will be associated with the elements $x_i \in U$).

Next, for each of the M sets X_i of the Set Cover instance, we specify 6 sets S_i, with each set having some of its elements at values within 3 of the first $3M$ set-intervals, and some values within the last N element-intervals. We use the first three set-intervals for X_1, the next three set-intervals for X_2, etc. Specifically, $X_1 = \{x_a, x_b, x_c, x_d\}$ corresponds to the sets $P_1 = \{k/2, 3k\}$, $Q_1 = \{3k/2, 5k\}$, $A_1 = \{5k/2, \ell_a\}$, $B_1 = \{7k/2, \ell_b\}$, $C_1 = \{9k/2, \ell_c\}$, $D_1 = \{11k/2, \ell_d\}$, where $\ell_i = 3M \cdot 2k + (i - 1)2k + k$ is the number (integer) at the midpoint of the ith element-interval. These sets are set up to allow "propagation" of a choice to include set X_1 in the set cover: If we select an element $s_i = k$ (using the "selection-sets" described next), thereby splitting the first set-interval into two intervals of size k, then we are free to select the right element, $3k$, in set P_1, which splits the second set-interval into two intervals of size k, and the right element, $5k$, in set Q_1; these choices result in the second and third set-intervals being split into two intervals of size k, freeing up the selections in sets A_1, B_1, C_1, and D_1 to be the right elements, $\ell_a, \ell_b, \ell_c, \ell_d$, each of which splits the element-intervals corresponding to x_a, x_b, x_c, x_d, effectively "covering" these elements. If we do not select an element $s_i = k$, then in order to have gaps of length at most k, we must select the *left* choices ($k/2$ and $3k/2$) in sets P_1 and Q_1, which then implies that we must also make the left choices in sets A_1, B_1, C_1, D_1, implying that we do not select splitting values in element-intervals corresponding to x_a, x_b, x_c, x_d, thereby not "covering" these elements.

Finally, we specify K sets, each exactly equal to the same set $\{k, 7k, 13k, \ldots, (M - 1)6k + k\}$. We call these the *selection-sets*. They each have one element in the middle of the first (of the 3) set-intervals associated with each set X_i. Selecting element $(i - 1)6k + k$ from one of these selection-sets corresponds to deciding to use set X_i in the collection $\mathcal{C}'$ of sets that should cover U. Since there are K selection-sets, we are allowed to use up to K sets X_i.

In total, then, our instance of URSPP has $m = (3M + N - 1) + 6M + K$ sets S_i.

Claim. The SET COVER instance has a solution if and only if the URSPP instance has a solution. In other words, there exists a set cover of size K if and only if there exists a selection of s_i's for URSPP with maximum gap k.

Proof. Assume the SET COVER instance has a solution, $\mathcal{C}'$, with $|\mathcal{C}'| = K$. Then, for the K selection-sets, we select one s_i corresponding to each $X \in \mathcal{C}'$. For each of these K selected sets, X_i, the corresponding sets P_i and Q_i are free to use the right choices, thereby freeing up sets A_i, B_i, C_i, D_i also to use right choices, effectively "covering" the 4 element-intervals corresponding to the elements of X_i. Since $\mathcal{C}'$ is a covering, we know that all N element-intervals are covered, resulting in all element-intervals being split into subintervals of size k. For sets X_i *not* in $\mathcal{C}'$, the corresponding sets P_i and Q_i use the left choices (in order that the first set-interval for X_i not have a gap larger than k), and the sets A_i, B_i, C_i, D_i also use the left choices (in order that the second and third set-intervals for X_i not have a gap larger than k). All gaps are therefore at most k, so the instance of URSPP has a solution.

Now assume that the instance of URSPP has a solution, with maximum gap k. Then, every element-interval must be split, by making right choices in several sets (A_i, B_i, C_i, D_i) associated with some of the sets X_i. If such a choice is made for even one set associated with X_i, then, in order to avoid a gap greater than k (of size at least $3k/2$) in either the second or third set-interval associated with X_i, at least one of the sets P_i or Q_i must also be a right choice. This then implies that there must be a selection-set that chooses to use the element that splits the first set-interval associated with X_i; otherwise that interval will have a gap of size at least $3k/2$. Thus, in order that URSPP has a solution, there must be a way to make selections in the selection-sets in order that the selected sets X_i form a cover of U. Thus, the instance of SET COVER has a solution.

In fact, our argument above shows that there exists a set cover of size K if and only if there exists a selection of s_i's for URSPP with maximum gap less than $3k/2$, since, in any suboptimal solution of the URSPP instance, the gap size is at least $3k/2$. This shows the claimed hardness of approximation.

On the positive side, we are able to give an approximation algorithm for the URSPP optimization problem:

Theorem 2. *The URSPP optimization problem has a polynomial-time 2-approximation.*

Proof. We show that in polynomial time we can run an algorithm, for a given positive integer k, that will report "success" or "failure". If it reports "success", it will provide a set of selections $s_i \in S_i$, one point per S_i, such that the maximum gap is at most $2k-1$. If it reports "failure", then we guarantee that it is impossible to make selections $s_i \in S_i$ such that all gaps are of size at most k. By running this algorithm for each choice of k, we obtain the claimed approximation algorithm.

The algorithm is simply a bipartite matching algorithm (see [18]). We consider the bipartite graph whose "red" nodes are the sets S_i and whose "blue" nodes

are the k-element integer sets $\{jk+1, jk+2, \ldots, jk+k\}$, for $j = 1, 2, \ldots$. There is an edge in the bipartite graph from red node S_i to blue node $\{jk+1, jk+2, \ldots, jk+k\}$ if and only if S_i contains an integer element in the set $\{jk+1, jk+2, \ldots, jk+k\}$. If there exists a matching for which every set $\{jk+1, jk+2, \ldots, jk+k\}$ is matched to a red node, then we report "success"; the corresponding elements from the S_i's have the property that no two consecutive selections s_i are separated by more than $2k+1$, since each interval $\{jk+1, jk+2, \ldots, jk+k\}$ has an s_i. On the other hand, if no matching exists for which every blue node is matched, then it is impossible to select one element s_i per set S_i with each set $\{jk+1, jk+2, \ldots, jk+k\}$ having an element s_i in it; this implies that one cannot achieve a selection with all gaps of size at most k.

We note that it is an interesting open problem to close the gap between the upper and lower bounds on the approximation factor (2 versus 3/2).

3.3 Dynamic Programming Algorithm

Consider a list of events, where each event can be one of (1) a location where a restriction enzyme is currently cutting or (2) a place where an unused restriction enzyme can be inserted, sorted according to their position along the DNA sequence. Let i be the number of events, S be a set of unused restriction enzymes, and j be the index of the last event placed in the sequence. Similarly, $\textsc{Position}(i)$ returns the position in the sequence where the event i occurs, $\textsc{Enzyme}(i)$ returns the actual enzyme that can be inserted at the position given by event i, and $\textsc{isCurrent}(i)$ returns $\textsc{True}$ if event i is a location where a restriction enzyme is currently cutting and $\textsc{False}$ otherwise.

Then the length of the minimum maximum gap possible can be found by the following recurrence relation

$$C[i,S,j] = \begin{cases} \textsc{Position}(j) & i < 0 \\ \max\{C[i-1,S,i], \textsc{Position}(j) - \textsc{Position}(i)\} & \textsc{isCurrent}(i) \\ \min\{\max\{C[i-1, S \setminus \{\textsc{Enzyme}(i)\}, i], & \\ \qquad \textsc{Position}(j) - \textsc{Position}(i)\}, C[i-1,S,j]\} & \text{otherwise} \end{cases}$$

Intuitively, current restriction sites should be kept and they have to be taken into consideration to find the length of the maximum gap. Now, for each place where an unused enzyme can be inserted we have two options: either we place the enzyme in that position or we do not. For each of these options we find the best placement among the remaining enzymes and the remaining events.

This algorithm runs in time $O(n^2 2^r)$ where r is the number of unused enzymes and n is the total number of events. Given the exponential dependence on the number of unused enzymes, the algorithm not only takes an exponential amount of time, but also requires an exponential amount of memory.

Our approach to overcoming this problem was to run the dynamic programming algorithm in blocks. First we run the algorithm to find the optimal placement of a feasibly small set of X enzymes first, then for the following X, and

so on until we have covered all enzymes. Moreover, as discussed below, we only insert enzymes with no initial recognition site, and before this, we apply a heuristic for deleting multiple restriction sites. This approach, thus, will not always give the exact optimal solution, but we will end with a good approximation. In Section 4 we give results for this approach using two orderings of enzymes: most possible insertion points first and fewest possible insertion points first. From our results, it was not clear that either ordering performed consistently better, however.

3.4 Practical Considerations

Restriction Map Construction. In our particular problem, we have a fixed set of patterns, known in advance (the restriction sites), and variable texts (the DNA sequence being analyzed). This fact justifies preprocessing the set of patterns so as to speed search. Particularly, we want to efficiently search for all occurrences of all of the restriction sites within a given DNA sequence in order to build the restriction map. Furthermore, whenever we make a base change we want to efficiently check that we have not created an occurrence for another restriction enzyme.

In order to accomplish these two objectives, we use the *Aho-Corasick algorithm* [19]. This is a dictionary-matching algorithm to efficiently find all occurrences of a finite set of patterns P in a given text, which works by constructing a deterministic finite automaton using a trie of the patterns in P.

Using this algorithm, we compute the main data structure used in the system, the *restriction map* (as commonly used in restriction enzyme manipulation tools; for in example in [20]). This data structure keeps track of the list of restriction enzymes, each with its name and its recognition site. Additionally, for every restriction enzyme we store a list of occurrences (start/end locations) in the DNA sequence that we are processing.

Deleting Recognition Sites. When deleting a restriction site, we do so by changing a single base whenever possible. If that base is within a gene (which is true most of the time for the sequences we are interested in), then the program makes a change that maintains the amino–acid sequence of the gene. We have some degree of flexibility in choosing the base change, even inside genes; however, we currently pick a synonymous codon arbitrarily.

It is worth noting that a sufficiently large number of synonymous base changes could substantially disrupt the codon bias of a gene. The codon bias of a genome is the statistical over-representation of certain codons over other synonymous codons. Genes which have codon bias significantly different from those of the host system (for example, poliovirus replicating in human cells) are known to express poorly. Thus, altering the codon bias of a viral genome could significantly affect the phenotype of the virus [21]. Although we do not believe that the number of codon changes we make is large enough to significantly alter the codon bias for a relatively large genome, such an issue could, in principle, be dealt with by a somewhat more clever policy to handle restriction site deletion.

Inserting Recognition Sites. When adding restriction sites, we again seek to change a minimal number of bases. However, we cannot just modify the bases of the sequence to create a restriction site for a given enzyme anywhere we want: we cannot modify locked regions, we have to make sure that we are maintaining the amino–acid sequence of genes, and we need to ensure that by creating a recognition site for a given enzyme we are not accidentally creating a recognition site for another enzyme, etc. A simple $O(nm)$ algorithm was implemented to find all the possible places where a given restriction enzyme can be inserted, where n is the length of the sequence and m is the length of the recognition site.

3.5 Program

Our heuristic approach for this problem is:

- First, eliminate all but one restriction site for each enzyme which appear in the genome initially.
- Second, insert new restriction sites for enzymes which do not appear in the genome initially.

Recognition Site Deletion Phase. The first phase following preprocessing seeks to create unique restriction sites from those sites which already appear in the genome. We are trying to have as many unique enzymes as possible in the sequence while trying to minimize the amount of work (in terms of total base changes), and we attempt to do this with a randomized greedy approach. We sort the enzymes by number of existing restriction sites, and we immediately lock the enzymes that have only one occurrence (since they are already unique and thus can be used without any base changes).

Then, for those enzymes that have 2 occurrences, we delete one of them, and keep the other. In increasing order of n, for enzymes with n occurrences we delete $n-1$, and we randomly keep only one. Although it is possible to construct pathological cases, in practice the randomization of which occurrence to keep makes the final distribution of sites quite uniform throughout the sequence. This part of the algorithm also discards enzymes that cannot be used because they have 2 or more occurrences within locked regions.

Insertion of Unused Enzymes. After modifying our sequence in order to delete restriction sites so that a subset of restriction enzymes have unique occurrences, we are left with *gaps*: sequences of contiguous bases between these unique occurrences in which there are no restriction sites. Our objective is to use the restriction enzymes that do not currently appear in the genome by creating recognition sites for them in order to reduce the size of these gaps. We do this by finding the ideal places for insertions (in order to minimize the maximum gap) and the actual possible places where the enzymes can be inserted. From this, we consider three approaches to insert the enzymes as close as possible to the optimal insertion points:

- using our exponential time dynamic programming algorithm in X enzyme phases, as discussed in Section 3.3;
- a greedy heuristic;
- and a maximal bipartite matching method.

Determining the Ideal Insertion Points. Consider the following problem. We are given a set of n gaps $G = \{g_1, g_2, \ldots, g_n\}$, where a gap g_i has length ℓ_i, and a set of m separators $S = \{s_1, s_2, \ldots, s_m\}$. A separator can be placed anywhere within a gap in order to split it in two. If a separator is placed at position p within a gap of length ℓ, the resulting two gaps will have size p and $\ell - p$.

Our task is to place the separators within the gaps so that the maximum length of the resulting gaps is minimized. The overall effect is that all the resulting gaps at the end of the process will be of approximately the same length (and restriction sites will be evenly distributed among the entire sequence).

Our algorithm to compute these ideal insertion points works as follows. We say each gap, g_i, is composed of k_i segments. Initially $k_i = 1$ for $1 \leq i \leq n$. Every time we insert a separator s_j within a gap g_i we increment k_i by one. Note that at this point we are not making a commitment in terms of the exact position in which s_j should be placed, we are just saying that s_j should be placed somewhere within g_i.

As long as there are unused separators, the next available separator should be placed in the gap g_i whose ratio between its length and the number of segments it is composed of (l_i/k_i) is highest.

FindPreferredInsertLocations(G, m)

1 Without lost of generality, assume $\ell_1/k_1 > \ell_2/k_2 > \cdots > \ell_n/k_n$
2 **while** there are separators available
3 **do** Place the current separator in the first gap of the list
4 Move the first gap of the list to its new position so that the list remains sorted in decreasing order of the ratio between its length and the number of segments it is composed of

The exact location where separators should be placed within a given gap can be found by evenly dividing the length of the gap by the number of segments composing the gap. Specifically, if a given gap g_i begins at position s_i of the sequence, $k_i - 1$ separators should be placed at positions $s_i + 1 \times \ell i/k_i$, $s_i + 2 \times \ell i/k_i$, $\ldots$, $s_i + (k_i - 1) \times \ell i/k_i$.

The above procedure is used to compute the ideal place where unused enzymes should be inserted in order to minimize the gaps, where G is a set of n gaps and m is the total number of unused enzymes whose recognition site can be created in at least one possible location. G is computed based on the current state of the restriction map after we have modified the sequence in order to create unique occurrences for a subset of restriction enzymes.

A Greedy Approach to Insertion. We implemented a simple greedy algorithm to try to insert the unused restriction enzymes on or close to the ideal

locations. At each step the algorithm tries to insert the restriction enzyme with the least number of potential insertion points, as close to a ideal insertion location as possible. Both the selected enzyme and the selected insert location are removed from their corresponding lists and the algorithm iterates until all restriction enzymes are inserted.

Weighted Bipartite Matching. An alternate approach to this problem is that after we have found both the list of ideal places where enzymes should be inserted and the list of places where each unused enzyme can actually be inserted, we formulate the problem of deciding which enzyme should be inserted in which location as a weighted bipartite matching problem.

Let $G = (X \cup Y, E)$ be a weighted bipartite graph where X is a set of unused restriction enzymes and Y is a list of ideal places where enzymes should be inserted. For each $x \in X$ we have an edge $e \in E$ from x to every $y \in Y$, where the weight of e is given by the squared distance in the sequence between y and the location where x can be inserted that is closest to y. We then compute the minimum weight perfect matching by using the Hungarian Algorithm [22]. This gives us, for each unused enzyme the location where we should create the recognition site for it.

4 Results

We tested our program on several viral sequences acquired from the Website of the National Center for Biotechnology Information (`http://www.ncbi.nlm.nih.gov`): Equine Arteritis Virus, Polio Virus, Enterobacteria Phage λ, Measles Virus, and Rubella Virus. For these results

Table 1. Results for varying viruses under different insertion heuristics, including the baseline algorithm, and the result of only removing duplicate sites, with no site insertion ("After Removal")

Virus	Metric	Initial	Baseline	After Removal	Greedy	Weighted Bipartite	Min Max Gap (Fewest First)	Min Max Gap (Most First)
Equine Arteritis Virus	# of base changes	N/A	90	158	188	196	186	190
	# of unique enzymes	24	29	80	90	91	88	92
	Max. gap length	3575	1866	949	671	714	498	413
λ Phage	# of base changes	N/A	149	371	383	384	384	384
	# of unique enzymes	18	28	77	82	82	82	82
	Max. gap length	10085	6288	3091	2954	2954	2954	2954
Measles Virus	# of base changes	N/A	247	358	397	399	395	395
	# of unique enzymes	15	42	83	90	89	89	88
	Max. gap length	4317	1087	1921	1118	1118	804	977
Polio Virus	# of base changes	N/A	141	120	207	216	194	189
	# of unique enzymes	35	40	81	104	105	104	110
	Max. gap length	982	537	685	459	240	269	269
Rubella Virus	# of base changes	N/A	197	174	219	218	204	215
	# of unique enzymes	32	40	84	99	99	94	97
	Max. gap length	990	772	658	351	351	314	314

We use a set of 162 restriction enzymes from the REBase restriction enzyme database [5] with recognition sites at least 6 bases in length (as shorter recognition sites appear very frequently).

In the tables below we give the number of nucleotides changed, total number of unique restriction sites, and maximum gap length for each of our insertion methods, as well as for the removal alone.

We give as a baseline for comparison one final heuristic. In this heuristic, we first compute a gap-length g which would generate evenly spaced positions throughout the genome, based on the total number of restriction enzymes which either appear or can be created in the genome. We then attempt to introduce a unique restriction site (either by insertion or by deletion) as close to position g as possible, say at position p. The heuristic then moves to position $p + g$ and repeats. This algorithm is similar to that used by GeneDesign [14].

5 Conclusion

We consider the problem of manipulating virus-length genomes to insert large numbers of unique restriction sites, while preserving wild-type phenotype. We give an abstraction of this problem, show that it is NP-Complete, give a $2-$approximation algorithm, and give a dynamic programming algorithm which solves it in exponential time and space. We also give several practical heuristics, which create genomes with several times more unique restriction sites, reducing the largest gap between adjacent sites by three to nine-fold.

Acknowledgments

We would like to thank Estie Arkin, George Hart, and other members of the Stony Brook Algorithms Reading Group for their contributions to this paper.

References

1. Bugl, H., Danner, J.P., Molinari, R.J., Mulligan, J.T., Park, H.O., Reichert, B., Roth, D.A., Wagner, R., Budowle, B., Scripp, R.M., Smith, J.A.L., Steele, S.J., Church, G., Endy, D.: Dna synthesis and biological security. Nature Biotechnology 25, 627–629 (2007)
2. Czar, M.J., Anderson, J.C., Bader, J.S., Peccoud, J.: Gene synthesis demystified. Trends in Biotechnology 27(2), 63–72 (2009)
3. Coleman, J.R., Papamichail, D., Skiena, S., Futcher, B., Wimmer, E., Mueller, S.: Virus attenuation by genome-scale changes in codon pair bias. Science 320(5884), 1784–1787 (2008)
4. Wimmer, E., Mueller, S., Tumpey, T., Taubenberger, J.: Synthetic viruses: a new opportunity to understand and prevent viral disease. Nature Biotech. 27(12), 1163–1172 (2009)
5. Roberts, R., Vincze, T., Posfai, J., Macelis, D.: Rebase–a database for dna restriction and modification: enzymes, genes and genomes. Nucl. Acids Res. 38, D234–D236 (2010)

6. González-Ballester, D., de Montaigu, A., Galván, A., Fernández, E.: Restriction enzyme site-directed amplification PCR: a tool to identify regions flanking a marker DNA. Anal. Biochem. 340(2), 330–335
7. Roberts, R.: How restriction enzymes became the workhorses of molecular biology. Proc. Natl. Acad. Sci. 102, 5905–5908 (2005)
8. Mens, T., Tourwe, T.: A survey of software refactoring. IEEE Transactions on Software Engineering 30(2), 126–139 (2004)
9. Serial Cloner, `http://serialbasics.free.fr/Serial_Cloner.html`
10. Cello, J., Paul, A., Wimmer, E.: Chemical synthesis of poliovirus cdna: generation of infectious virus in the absence of natural template. Science 297(5583), 1016–1018 (2002)
11. Chan, L., Kosuri, S., Endy, D.: Refactoring bacteriophage t7. Mol. Syst. Biol. 1 (2005)
12. Anand, I., Kosuri, S., Endy, D.: Genejax: A prototype cad tool in support of genome refactoring (2006)
13. Evans, P., Liu, C.: Sitefind: A software tool for introducing a restriction site as a marker for successful site-directed mutagenesis. BMC Mol. Biol. 6(22)
14. Richardson, S.M., Wheelan, S.J., Yarrington, R.M., Boeke, J.D.: Genedesign: Rapid, automated design of multikilobase synthetic genes. Genome Res. 16(4), 550–556 (2006)
15. Skiena, S.: Designing better phages. Bioinformatics 17, S253–S261 (2001)
16. Papadimitriou, C., Yannakakis, M.: Optimization, approximation, and complexity classes. In: STOC 1988: Proceedings of the twentieth annual ACM symposium on Theory of computing, pp. 229–234. ACM, New York (1988)
17. Duh, R.C., Frer, M.: Approximation of k-set cover by semi-local optimization. In: Proc. 29th STOC, pp. 256–264. ACM, New York (1997)
18. Hopcroft, J.E., Karp, R.M.: An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. SIAM Journal on Computing 2(4), 225–231 (1973)
19. Aho, A.V., Corasick, M.J.: Efficient string matching: An aid to bibliographic search. Communications of the ACM 18(6), 333–340 (1975)
20. Vincze, T., Posfai, J., Roberts, R.J.: NEBcutter: a program to cleave DNA with restriction enzymes. Nucl. Acids Res. 31(13), 3688–3691 (2003)
21. Ermolaeva, M.: Synonymous codon usage in bacteria. Curr. Issues Mol. Biol. 3(4), 91–97 (2001)
22. Kuhn, H.W.: The hungarian method for the assignment problem. Naval Research Logistics Quarterly 2, 83–97 (1955)

Extension and Faster Implementation of the GRP Transform for Lossless Compression

Hidetoshi Yokoo

Department of Computer Science, Gunma University
Kiryu 376-8515, Japan
yokoo@cs.gunma-u.ac.jp

Abstract. The GRP transform, or the *generalized radix permutation* transform was proposed as a parametric generalization of the BWT of the block-sorting data compression algorithm. This paper develops its extension that can be applied with any combination of parameters. By using the technique developed for linear time/space implementation of the sort transform, we propose an efficient implementation for the inverse transformation of the GRP transform. It works for arbitrary parameter values, and can convert the transformed string to the original string in time linear in the string length.

1 Introduction

The GRP transform, or the *generalized radix permutation* transform, was proposed by Inagaki, Tomizawa, and the present author in [3] as a parametric generalization of the BWT of the block-sorting data compression algorithm [1],[2]. The BWT and its variations [6],[7],[8] can be derived from the GRP transform as its special cases. The GRP transform has two parameters: the *block length* ℓ and the *context order* d, to which we can assign appropriate values so that we can also realize new transforms. In this sense, the GRP transform is a proper extension of those existing transforms. Preliminary experiments [4] show that some files are more efficiently compressed by an appropriate combination of parameter values with a tuned second-step encoder than by the original BWT.

In spite of its generality, the GRP transform is given concrete procedures only in the case where its parameters satisfy $n = b\ell$ and $0 \leq d \leq \ell$ for the length n of the string to be compressed and for an integer b. It is conceptually possible to remove these restrictions and to allow n, ℓ and d to be any natural numbers. However, it is not obvious whether we can run such a general version in an efficient way. In our previous paper [3], we have shown that the GRP transform can be performed in $O(n + bd) = O(n + nd/\ell)$ time. This implies that the transform runs in time linear in the string length, as long as the parameter d stays in the range of $0 \leq d \leq \ell$. However, it may require quadratic time when we wish to increase d beyond the range.

In the GRP transform, the inverse transformation is more complicated than the forward transformation. We observe a quite similar situation [5], [6] in the

A. Amir and L. Parida (Eds.): CPM 2010, LNCS 6129, pp. 338–347, 2010.

Sort Transform (ST) [7], which has been proposed as a finite-order variant of the BWT. The ST, which is also a special case of the GRP transform, was originally developed to speed up the BWT. This aim was achieved in its forward transformation with a trade-off of a more demanding and complicated inverse transformation. Nong and Zhang [5] have addressed the problem of developing an efficient inverse ST transform, and gave a linear time/space algorithm with Chan in their recent paper [6].

In this paper, we show that the method developed by Nong, Zhang, and Chan [6] can be applied to the GRP transform so that its inverse transformation can be performed in linear time for any combination of the parameters. In order to show this and make clear the relation between the GRP transform and the BWT, we adopt a completely different description on the GRP transform than that given in our previous paper [3]. For example, in [3], the original data string to be compressed is arranged as a column vector of a matrix. In the present paper, on the other hand, we adopt the convention of the BWT, in which the original string is arranged as a row vector of a matrix. The BWT gathers those symbols that occur in the same or similar contexts in a source string, where the contexts are *backward* ones in the sense that the string is traversed from right to left. In this respect, too, we follow the BWT. The introduction of the conventions of the BWT and the ST to the description here makes it easy to understand that its inverse transformation runs in linear time.

The rest of the paper is organized as follows: Section 2 gives an extended version of the GRP transform. We extend the transform to allow arbitrary parameters. Our main emphasis is on the development of an efficient inverse transformation. For this, we apply the technique by Nong, Zhang, and Chan [5], [6] to our transform. Its details and other remarks on complexity issues will be given separately in Section 3.

2 GRP Transform with Arbitrary Parameters

2.1 Preliminaries

We consider lossless compression of strings over an ordered alphabet $\mathcal{A}$ of a finite size $|\mathcal{A}|$. Elements of the alphabet are called *symbols*. The alphabetically largest symbol in the alphabet, denoted by \$, is a *sentinel*, which appears exactly once at the end of a data string. We represent a string of length n by

$$x[1..n] = x_1 x_2 \cdots x_{n-1}\$, \tag{1}$$

where the ith symbol is denoted by $x[i]$, i.e., $x[i] = x_i$ for $1 \leq i \leq n-1$ and $x[n] = \$$. Similarly, a two-dimensional $m \times n$ matrix M of symbols is denoted by $M[1..m][1..n]$. The purpose of introducing sentinels is twofold. While the first one is the same as that adopted in the explanation of the BWT, the second one is specific to the present paper. We append some extra sentinels to the input string during the transformation in order to make its length an integer multiple of a parameter.

The GRP transform, which is an extension of our previous one [3], converts an input string (1) into another string $y[1..n] \in \mathcal{A}^n$. The transform has two parameters: the *block length* ℓ and the (*context*) *order* d, where ℓ is an integer in $1 \leq \ell \leq n$ and d is an integer in $0 \leq d \leq n$. The values of these parameters and the string length n are shared by the forward and inverse transformations so that the original string (1) can be uniquely recovered from the transformed string $y[1..n]$.

In the forward transformation, the input string (1) is divided into blocks of the same length ℓ. We call an integer $b = \lceil n/\ell \rceil$ the *number of blocks* of the string. We first consider a string

$$x'[1..b\ell] = x_1 x_2 \cdots x_{n-1} \$ \cdots \$, \tag{2}$$

which is a concatenation of $x[1..n]$ and extra $b\ell - n$ sentinels. Let $x'[0] = \$$. Then, the forward transformation begins with a $b \times b\ell$ matrix $M = M[1..b][1..b\ell]$ whose (i, j) element is initialized as

$$M[i][j] = x'[(i-1)\ell + j \bmod b\ell], \quad 1 \leq i \leq b,\ 1 \leq j \leq b\ell. \tag{3}$$

The leftmost d columns and the rightmost ℓ columns of M are called the *reference part* and the *output part*, respectively.

Example: For $n = 11$, $d = 4$, $\ell = 3$, consider the string:

$$x[1..11] = \texttt{bacacabaca\$}. \tag{4}$$

Then, $b = \lceil 11/3 \rceil = 4$, $b\ell = 12$, and $x'[1..12] = \texttt{bacacabaca\$\$}$. The initial configuration of M is given below. Note that the string $x'[1..b\ell]$ appears as the first row, and each row is followed by its ℓ-symbol left cyclic shift. Thus, the concatenation of the rows of the left ℓ symbols forms the string $x'[1..b\ell]$. If we concatenate the rows of any consecutive ℓ columns in an appropriate order, we can recover $x'[1..b\ell]$.

$$M = M[1..4][1..12] = \begin{bmatrix} \underbrace{\texttt{b a c a}}_{\text{reference part}} & \texttt{c a b a c a} & \texttt{\$ \$} & \underbrace{\texttt{b a c}}_{\text{output part}} \\ \texttt{a c a b} & \texttt{a c a \$} & \texttt{\$ b} & \texttt{a c a} \\ \texttt{b a c a} & \texttt{\$ \$ b} & \texttt{a c a} & \texttt{c a a} \\ \texttt{a \$ \$ b} & \texttt{a c a c} & \texttt{a b} & \texttt{a c} \end{bmatrix}.$$

The forward and inverse transformations can be described in terms of operations on the matrix M. The most basic operation is the sorting of the row vectors. Sorting is performed in a stable manner by using the entire row or its part as a sorting key.

In our previous paper [3], we defined the GRP transform on a matrix consisting only of the reference and output parts because other elements have nothing to do with the transform. In fact, the matrix representation is not essential to the transform. We simply adopt the above representation so that we can intuitively understand the relation of the transform with the BWT and the ST.

2.2 Forward Transformation

The forward transformation comprises sorting of the row vectors of M and output of its column vectors.

1. /∗ Initialization ∗/
   Convert the input string into a matrix $M = M[1..b][1..b\ell]$ using (3);
2. Use the symbols of the reference part (first d columns) of M as a sort key, and sort the rows of M lexicographically and stably;
   /∗ The matrix M is said to be in state "A" immediately after Step 2. ∗/
3. **for** $j := b\ell$ **downto** $b\ell - \ell + 1$ **do**
   (a) Output the symbols of the jth column of the current M according to:

$$y'[(b\ell - j)b + i] := M[i][j] \text{ for } 1 \leq i \leq b;$$

   (b) **if** $j = b\ell - \ell + 1$ **then** break;
   (c) Sort the row vectors of M in a stable manner by the symbols of the jth column;
   end for

When there are more than one sentinels in $x'[1..b\ell]$, the second and other succeeding sentinels are outputted from the last row. Therefore, we do not have to include these sentinels except for the first one in the transformed string. In this case, the number of symbols outputted from the forward transformation is equal to n. We represent the output of the corresponding transformation by $y[1..n] = y_1 y_2 \ldots y_n$. The output drawn directly from the above procedure has been denoted by $y'[1..b\ell]$. Both $y[1..n]$ and $y'[1..b\ell]$ are easily converted to one another by sharing the parameters d, ℓ, and n.

Example: For the string in (4) and $d = 4$, $\ell = 3$, we have

$$y'[1..12] = \texttt{cc\$acaa\$bbaa}, \tag{5}$$

$$y[1..11] = \texttt{cc\$acaabbaa}. \tag{6}$$

2.3 Relation with Existing Transforms

Before proceeding to a description of the inverse transformation, we reveal close relations between the GRP transform and other established transforms.

First, consider the case of $\ell = 1$ and $d = n$. In this case, it is easy to see that $y[1..n]$ $(= y'[1..b\ell])$ is exactly the same as one obtained when we apply the BWT to $x[1..n]$. In this sense, the GRP transform can be regarded as a proper extension of the BWT. Similarly, we can consider the case where we limit the order d to an integer $k < n$ with $\ell = 1$. This is known as the k-order variant of the BWT, or sometimes called the Sort Transform (ST) [5],[6],[7].

Now, consider an application of the ST to a string over $\mathcal{A}^\ell$. For example, suppose that we have a string `CACB` over $\mathcal{A}^3$, where `A` = `aca`, `B` = `a$$`, and

C = bac. The ST with any $k \geq 1$ transforms this string into CCBA. If we apply the GRP transform with $d = 1$ and $\ell = 1$ to the same string CACB, we have

$$M = \begin{bmatrix} \mathtt{C} & \mathtt{A} & \mathtt{C} & \mathtt{B} \\ \mathtt{A} & \mathtt{C} & \mathtt{B} & \mathtt{C} \\ \mathtt{C} & \mathtt{B} & \mathtt{C} & \mathtt{A} \\ \mathtt{B} & \mathtt{C} & \mathtt{A} & \mathtt{C} \end{bmatrix} \longrightarrow M_{\mathrm{A}} = \begin{bmatrix} \mathtt{A} & \mathtt{C} & \mathtt{B} & \mathtt{C} \\ \mathtt{B} & \mathtt{C} & \mathtt{A} & \mathtt{C} \\ \mathtt{C} & \mathtt{A} & \mathtt{C} & \mathtt{B} \\ \mathtt{C} & \mathtt{B} & \mathtt{C} & \mathtt{A} \end{bmatrix} = \begin{bmatrix} \mathtt{a} & \mathtt{c} & \mathtt{a} & \mathtt{b} & \mathtt{a} & \mathtt{c} & \mathtt{a} & \$ & \$ & \mathtt{b} & \mathtt{a} & \mathtt{c} \\ \mathtt{a} & \$ & \$ & \mathtt{b} & \mathtt{a} & \mathtt{c} & \mathtt{a} & \mathtt{c} & \mathtt{a} & \mathtt{b} & \mathtt{a} & \mathtt{c} \\ \mathtt{b} & \mathtt{a} & \mathtt{c} & \mathtt{a} & \mathtt{c} & \mathtt{a} & \mathtt{b} & \mathtt{a} & \mathtt{c} & \mathtt{a} & \$ & \$ \\ \mathtt{b} & \mathtt{a} & \mathtt{c} & \mathtt{a} & \$ & \$ & \mathtt{b} & \mathtt{a} & \mathtt{c} & \mathtt{a} & \mathtt{c} & \mathtt{a} \end{bmatrix},$$

where M_{A} represents the matrix M in state A. With a simple observation, we can see that, in the case of $d = kl$, the application of the k-order ST to an $|\mathcal{A}^{\ell}|$-ary string is essentially equivalent to obtaining the matrix M in state A in the GRP transform with parameters ℓ and d for the corresponding $|\mathcal{A}|$-ary string. The main difference between the k-order ST for an $|\mathcal{A}^{\ell}|$-ary string and the GRP transform for the equivalent $|\mathcal{A}|$-ary string lies in Step 3 of the forward transformation of the GRP transform, in which $|\mathcal{A}|$-ary symbols are outputted symbolwise and sorted repeatedly according to their right neighbor columns.

In the sequel, we represent a matrix M in state A by M_{A}, which plays a similar role to a sorted matrix in the BWT and the ST.

2.4 Inverse Transformation

The inverse transformation of the GRP transform consists mainly of two parts: reconstruction of the output part of M_{A} and restoring the original string.

Let $L_{\ell}[1..b][1..\ell]$ denote the output part of M_{A}. That is, $L_{\ell}[i][j] = M_{\mathrm{A}}[i][(b-1)\ell+j]$ for $1 \leq i \leq b$ and $1 \leq j \leq \ell$. It can be reconstructed from the transformed string $y'[1..b\ell]$ in the following way.

/* Reconstruction of the output part of M_{A} */

1. Allocate a $b \times \ell$ matrix $L_{\ell}[1..b][1..\ell]$ of symbols, and set

$$L_{\ell}[i][1] := y'[i + (\ell - 1)b] \text{ for } 1 \leq i \leq b;$$

2. **for** $j := 2$ **to** ℓ **do**
   (a) Sort the symbols in $y'[1 + (\ell - j)b..b + (\ell - j)b]$ alphabetically, and put the result into $L_{\ell}[1..b][j]$;
   (b) Rearrange the rows of $L_{\ell}[1..b][1..j]$ in a stable manner so that its jth column corresponds to $y'[1 + (\ell - j)b..b + (\ell - j)b]$;
   end for

The validity of the above procedure was given in [3] as Lemma 1. The output part $L_{\ell}[1..b][1..\ell]$ plays a similar role to the last column of the sorted matrix of BWT. In BWT, the symbols in the last column are then sorted in order to obtain the first column of the same matrix.

In the GRP transform, on the other hand, we can use $L_{\ell}[1..b][1..\ell]$ to obtain the first d or ℓ columns of M_{A}, depending on the value of d. Actually, however, we do not recover explicitly the left columns of M_{A}. Instead, we keep only two mappings between the left and right ends of M_{A}. To do so, we perform stable sort

on the set of row vectors of $L_\ell[1..b][1..\ell]$ in lexicographic order of their prefixes of length $\min\{d, \ell\}$. As a result of sorting, if $L_\ell[i][1..\ell]$ is the jth one of the sorted list of the row vectors, then define two column vectors: $P[1..b]$ and $Q[1..b]$ by

$$P[i] = j \text{ and } Q[j] = i.$$

When $d = 0$, they are defined by $P[i] = n$ for $i = 1$ and $P[i] = i - 1$ otherwise, and $Q[j] = 1$ for $j = n$ and $Q[j] = j + 1$ otherwise.

In order to continue our description of the inverse transform, we borrow some notions from [6], in which Nong, Zhang, and Chan developed a linear time implementation of the inverse ST. We first introduce a binary vector $D[1..b] \in \{0, 1\}^b$ such that

$$D[i] = \begin{cases} 0 & \text{for } i \geq 2 \text{ and } M_{\mathrm{A}}[i][1..d] = M_{\mathrm{A}}[i-1][1..d], \\ 1 & \text{for } i = 1 \text{ or } M_{\mathrm{A}}[i][1..d] \neq M_{\mathrm{A}}[i-1][1..d]. \end{cases} \tag{7}$$

Further, we introduce a counter vector $C_d[1..b]$ and an index vector $T_d[1..b]$. If $D[i] = 0$, then $C_d[i]$ is also defined to be zero. Otherwise, $C_d[i]$ stores the number of the same prefixes of the row vectors in M_{A} as $M_{\mathrm{A}}[i][1..d]$. The index vector $T_d[1..b]$ along with $C_d[1..b]$ is computed in the following way, provided that $D[1..b]$ is given. The computation of $D[1..b]$ will be deferred to the next section.

/* Computation of T_d and C_d */
1. Set $C_d[1..b]$ to be a zero vector;
2. **for** $i := 1$ **to** b **do**
   (a) **if** $D[i] = 1$ then set $j := i$;
   (b) Set $T_d[Q[i]] := j$;
   (c) Set $C_d[j] := C_d[j] + 1$;
   end for

We are now ready to summarize the inverse transformation. We can restore the original string $x'[1..b\ell]$ using the following procedure.

/* Restoring the original string */
1. Set $j := (b - 1)\ell + 1$;
2. Set $i :=$ such an index that $L_\ell[i][1..\ell]$ includes the sentinel \$;
3. **while** $j > 0$ **do**
   (a) Set $x'[j..j + \ell - 1] := L_\ell[i][1..\ell]$;
   (b) Set $i := T_d[i]$;
   (c) Set $C_d[i] := C_d[i] - 1$;
   (d) Set $i := i + C_d[i]$;
   (e) Set $j := j - \ell$;
   end while

Example: First, the transformed string in (6) is converted to its equivalent form (5) by using $n = 11$ and $\ell = 3$. Then, $L_\ell[1..b][1..\ell]$ and other auxiliary quantities

are obtained as shown in the left table. Finally, the original string $x'[1..12]$ is restored as shown right below.

i	$L_3[i][1..3]$	$Q[i]$	$D[i]$	$T_4[i]$	$C_4[i]$
1	bac	4	1	3	1
2	bac	3	1	3	1
3	a$$	1	1	2	2
4	aca	2	0	1	0

j	$L_3[i][1..3]$, $i=3$
10	$x'[10..12]$ = a$$, $i=2$
7	$x'[7..9]$ = bac, $i=4$
4	$x'[4..6]$ = aca, $i=1$
1	$x'[1..3]$ = bac

3 Details of the Inverse Transformation

3.1 Computation of Vector *D*

The inverse transformation above is based on the framework of Nong and Zhang [5]. A key issue in the framework lies in the computation of D, which they called the *context switch* vector. The same authors gave a more efficient solution to the issue in their recent paper [6]. We here generalize their recent technique in order to apply it to the computation of the vector $D[1..b]$ in our case.

As shown in (7), the vector $D[1..b]$ is defined on the matrix in state A, which is not available when we are about to compute $D[1..b]$. However, the left d columns of $M_{\mathrm{A}}[1..d][1..b\ell]$ can be retrieved only by $L_\ell[1..b][1..\ell]$ and $Q[1..b]$ in the following way.

/* Reconstruction of the ith row of the reference part of M_{A} */
1. Set $j := 1$ and $k := Q[i]$;
2. **while** $j \le d$ **do**
   (a) Set $M_{\mathrm{A}}[i][j..j+\ell-1] := L_\ell[k][1..\ell]$; /* valid only up to d columns */
   (b) Set $k := Q[k]$; $j := j + \ell$;
   end while

When we wish to have a single value $D[i]$ for a specific i, we may perform the above procedure for $i-1$ and i, and compare the results symbol by symbol. However, in order to obtain the set $D[1..b]$ of those values more efficiently, we rather use a new quantity *height* and the notion of *cycles*. Thus, the above reconstruction procedure is no longer used in actual transformation, but should be remarked as a procedural representation of *Property 6* in [6] with being generalized to our transform. Property 6 in [6] says that a limited (constant) order context of the ST can be retrieved by the combination of mapping Q and symbols in the last column.

The vector $height[1..b]$ represents the lengths of the longest common prefixes (LCPs) between adjacent rows in M_{A}. Let $height[i]$ denote the length of the LCP between $M_{\mathrm{A}}[i-1][1..b\ell]$ and $M_{\mathrm{A}}[i][1..b\ell]$. Obviously, $\{D[i]=0\}$ is equivalent to $\{height[i] \ge d\}$ for $2 \le i \le b$. The following theorem is a generalization of Theorem 1 in [6] to the GRP transform with parameters d and ℓ. The original theorem in [6] corresponds to the case of $\ell = 1$.

Theorem 1. *height*$[Q[i]] \geq$ *height*$[i] - \ell$ *for* $2 \leq i \leq b$.

Both in the above procedure and in the theorem, the indexes to the row vectors are retrieved one after another by the use of mapping Q. Starting from an arbitrary integer i in [1, b], we eventually return to the same integer after successive applications of Q. Thus, it is natural to define a set of indexes

$$\alpha(i) = \{i, Q[i], Q^2[i], \ldots, Q^{b-1}[i]\}, \tag{8}$$

where $Q^k[i]$ represents $Q[i]$ for $k = 1$ and $Q[Q^{k-1}[i]]$ for $k \geq 2$. We call $\alpha(i)$ a *cycle*, which we may regard as either a set or a linear list of indexes (integers) depending on the context. Obviously, any two cycles are disjoint. Although our definition of cycles is slightly different from that in [6], where a cycle is defined as a list of *symbols*, we can apply almost the same discussions as those in [6] to the computation of D in the inverse GRP transformation. The most characteristic difference arises when we apply Theorem 1 to the computation of *height*s along a cycle. A specific procedure follows.

/* Computation of the heights for the indexes in a cycle $\alpha(i)$ */
1. Set $j := i$ and $h := 0$;
2. **do**
   (a) **while** $h < d$ **do**
   if $j = 1$ or *Diff*(j, h) **then** break **else** h++;
   end while
   (b) Set *height*$[j] := h$;
   (c) Set $h := \max\{h - \ell,\ 0\}$;
   (d) Set $j := Q[j]$;
   while $j \neq i$

In the above procedure, *Diff*(j, h) is a boolean predicate that indicates whether the hth symbols ($1 \leq h \leq d$) of the $j - 1$th and jth row vectors of M_{A} are different. If we can perform this comparison in $O(1)$ time, the time complexity of the above procedure becomes $O(d + \ell \cdot |\alpha(i)|)$ for the cardinality $|\alpha(i)|$ of cycle $\alpha(i)$. This comes from the fact that the total number of times we increment h in Step 2 (a) never exceeds the sum of d and the total number of decrements of h in Step 2 (c). Since the sum of the cardinalities $|\alpha(i)|$ of all the cycles is equal to $b = \lceil n/\ell \rceil$, the essential issues to be addressed when we wish to compute the vector D in linear time are the development of $O(1)$-implementation of *Diff*(j, h) and the exclusion of d from the complexity of the above procedure for all cycles. Actually, these two issues are the main focus of Nong, Zhang, and Chan [6].

We can apply their method [6] almost as it is in order to compute $D[1..b]$ in time linear in the string length. First, to implement *Diff*(j, h), note that the hth symbol of the jth row of M_{A} is given by

$$M_{\mathrm{A}}[j][h] = L_\ell[Q^{\lceil h/\ell \rceil}[j]][(h-1) \bmod \ell + 1] \quad \text{for } 1 \leq j \leq b,\ 1 \leq h \leq d. \tag{9}$$

This can be validated by the reconstruction procedure of the reference part of M_{A}, which was given in the beginning of this subsection. In (9), $Q^{\lceil h/\ell \rceil}[j]$ can be

computed in a constant time by the use of a suitable data structure [6]. Thus, we can compute $\mathit{Diff}(j,h)$ in $O(1)$-time from $L_\ell[1..b][1..\ell]$.

As for the exclusion of d from the complexity $\sum_{\text{cycles}} O(d+\ell\,|\alpha(i)|)$, we can completely follow the technique in [6]. As a result, we can compute the vector $D[1..b]$ in $O(\ell b) = O(n)$ time without depending on the value of d.

3.2 Summary of Complexity Issues

We summarize the complexity of the inverse transformation.

The inverse transformation begins with the reconstruction of the output part $L_\ell[1..b][1..\ell]$ of M_{A}. Assuming that sorting of symbols can be performed in linear time using bucket sorting, we can reconstruct the output part in time linear in the string length n. Then, we proceed to the computation of the mappings $P[1..b]$ and $Q[1..b]$. These mappings are obtained by the sorting of the set of row vectors of $L_\ell[1..b][1..\ell]$. Since the number of the row vectors is b and the length of the key is at most ℓ symbols, the mappings are produced by radix sort in $O(b\ell) = O(n)$ time.

The computation of vectors $T_d[1..b]$ and $C_d[1..b]$ can be done obviously in time linear in b when we already have the vector D, which we can obtain in $O(n)$ time, as mentioned above. The last operation for restoring the original string is to simply copy the set of row vectors of $L_\ell[1..b][1..\ell]$ in a designated order. Thus, we can now have the following theorem.

Theorem 2. *The inverse transformation of the proposed GRP transform can restore the original string of length n in $O(n)$ time.*

Although we have not discussed the space complexity so far, it is obvious that the space requirement for the inverse transformation is also $O(n)$ because $L_\ell[1..b][1..\ell]$ requires $O(n)$ space while other auxiliary vectors P, Q, D, T_d, and C_d require only $O(b)$ space.

To conclude this subsection, we must give a brief comment on the complexity of the forward transformation. Its time complexity is obviously $O(b(d+\ell)) = O(bd+n)$ when we implement it by using simple radix sort. Although it is desirable to exclude the order d as in the case with the inverse transformation, $O(bd+n)$ is not so demanding in practice. If we use the same radix sort to implement the k-order ST, the time complexity of the k-order ST will be $O(kn)$. By choosing appropriate parameters in the GRP transform, we can make its time complexity of $O(bd+n)$ significantly smaller than $O(kn)$ of the k-order ST. When the GRP transform corresponds to the BWT, we can use various techniques developed for faster construction of a suffix array [1]. However, it does not seem plausible to be able to generalize those techniques to the GRP transform. On the other hand, the space requirement of the forward transformation is simply $O(n)$. An array for the input string with a permutation vector representing the row order of M will suffice to implement the matrix. We can access its elements via the relation (3).

4 Conclusion

We have proposed an extension of the GRP transform and its efficient implementation for the inverse transformation. The GRP transform is a proper generalization of the BWT and their typical variations. We can also say that, in an algorithmic viewpoint, the proposed inverse transformation is a generalization of the Nong–Zhang–Chan implementation of the inverse ST [6]. The generality of the GRP transform will result also in the generality of second-step encoders, which are used after the transform for actual compression. We can extend the Move-to-Front heuristics, an example of the second-step encoders incorporated into the block-sorting compression algorithm, so that it accommodates the characteristics of the output of the GRP transform. We will present its details with compression experiments on another occasion.

Acknowledgement

We thank Kazumasa Inagaki and Yoshihiro Tomizawa for their contribution to our previous work.

References

1. Adjeroh, D., Bell, T., Mukherjee, A.: The Burrows-Wheeler Transform: Data Compression, Suffix Arrays, and Pattern Matching. Springer, Heidelberg (2008)
2. Burrows, M., Wheeler, D.J.: A block-sorting lossless data compression algorithm. SRC Research Report 124, Digital Systems Research Center, Palo Alto (1994)
3. Inagaki, K., Tomizawa, Y., Yokoo, H.: Novel and generalized sort-based transform for lossless data compression. In: Karlgren, J., Tarhio, J., Hyyrö, H. (eds.) SPIRE 2009. LNCS, vol. 5721, pp. 102–113. Springer, Heidelberg (2009)
4. Inagaki, K., Tomizawa, Y., Yokoo, H.: Data compression experiment with the GRP transform (in Japanese). In: 32nd Sympo. on Inform. Theory and its Applications, SITA 2009, Yamaguchi, Japan, pp. 330–335 (2009)
5. Nong, G., Zhang, S.: Efficient algorithms for the inverse sort transform. IEEE Trans. Computers 56(11), 1564–1574 (2007)
6. Nong, G., Zhang, S., Chan, W.H.: Computing inverse ST in linear complexity. In: Ferragina, P., Landau, G.M. (eds.) CPM 2008. LNCS, vol. 5029, pp. 178–190. Springer, Heidelberg (2008)
7. Schindler, M.: A fast block-sorting algorithm for lossless data compression. In: DCC 1997, Proc. Data Compression Conf, Snowbird, UT, p. 469 (1997)
8. Vo, B.D., Manku, G.S.: RadixZip: Linear time compression of token streams. In: Very Large Data Bases: Proc. 33rd Intern. Conf. on Very Large Data Bases, Vienna, pp. 1162–1172 (2007)

Parallel and Distributed Compressed Indexes*

Luís M.S. Russo[1], Gonzalo Navarro[2], and Arlindo L. Oliveira[3]

[1] CITI, Departamento de Informática, Faculdade de Ciências e Tecnologia, FCT, Universidade Nova de Lisboa, 2829-516 Caparica, Portugal
lsr@di.fct.unl.pt
[2] Dept. of Computer Science, University of Chile
gnavarro@dcc.uchile.cl
[3] INESC-ID, R. Alves Redol 9, 1000 Lisboa, Portugal
aml@algos.inesc-id.pt

Abstract. We study parallel and distributed compressed indexes. Compressed indexes are a new and functional way to index text strings. They exploit the compressibility of the text, so that their size is a function of the compressed text size. Moreover, they support a considerable amount of functions, more than many classical indexes. We make use of this extended functionality to obtain, in a shared-memory parallel machine, near-optimal speedups for solving several stringology problems. We also show how to distribute compressed indexes across several machines.

1 Introduction and Related Work

Suffix trees are extremely important for a large number of string processing problems, in particular in bioinformatics, where large DNA and protein sequences are analyzed. This partnership has produced several important results, but it has also exposed the main shortcoming of suffix trees. Their large space requirements, plus their need to operate in main memory to be useful in practice, renders them inapplicable in the cases where they would be most useful, that is, on large texts.

The space problem is so important that it has originated a plethora of research, ranging from space-engineered suffix tree implementations [1] to novel data structures to simulate them, most notably suffix arrays [2]. Some of those space-reduced variants give away some functionality. For example suffix arrays miss the important suffix link navigational operation. Yet, all these classical approaches require $O(n \log n)$ bits, while the indexed string requires only $n \log \sigma$ bits[1], being n the size of the string and σ the size of the alphabet. For example the human genome can be represented in 700 Megabytes, while even a space-efficient suffix tree on it requires at least 40 Gigabytes [3], and the reduced-functionality suffix array requires more than 10 Gigabytes. This problem is particularly evident in DNA because $\log \sigma = 2$ is much smaller than $\log n$.

* Funded in part by Millennium Institute for Cell Dynamics and Biotechnology (ICDB), Grant ICM P05-001-F, Mideplan, and Fondecyt grant 1-080019, Chile (second author).

[1] In this paper log stands for $\log_2$.

A. Amir and L. Parida (Eds.): CPM 2010, LNCS 6129, pp. 348–360, 2010.

These representations are also much larger than the size of the *compressed* string. Recent approaches [4] combining data compression and succinct data structures have achieved spectacular results on what we will call generically *compressed suffix arrays (CSAs)*. These require space close to that of the compressed string and support efficient indexed searches. For example the most compact member of the so-called *FM-Index family* [5], which we will simply call *FMI*, is a CSA that requires $nH_k + o(n \log \sigma)$ bits of space and counts the number of occurrences of a pattern of length m in time $O(m(1 + \frac{\log \sigma}{\log \log n}))$. Here nH_k denotes the k-th order empirical entropy of the string [6], a lower bound on the space achieved by any compressor using k-th order modeling. Within that space the FMI represents the text as well, which can thus be dropped.

It turns out that it is possible to add a few extra structures to CSAs and support all the operations provided by suffix trees. Sadakane was the first to present such a *compressed suffix tree (CST)* [3], adding $6n$ bits to the size of the CSA. This $\Theta(n)$ extra-bits space barrier was recently broken by the so-called *fully-compressed suffix tree (FCST)* [7] and by another entropy-bounded CST [8]. The former is particularly interesting as it achieves $nH_k + o(n \log \sigma)$ bits of space, asymptotically the same as the FMI, its underlying CSA.

Distributing CSAs have been studied, yet focusing only on pattern matching. For example, Mäkinen et al. [9] achieved optimal speedup in the amortized sense, that is, when many queries arrive in batch.

In this paper we study parallel and distributed algorithms for several stringology problems (with well-known applications to bioinformatics) based on compressed suffix arrays and trees. This is not just applying known parallel algorithms to compressed representations, as the latter have usually richer functionality than classical ones, and thus offer unique opportunities for parallelization and distributed representations. In Section 4 we present parallel shared-memory algorithms to solve problems like pattern matching, computing matching statistics, longest common substrings, and maximal repeats. We obtain near-optimal speedups, by using sophisticated operations supported by these compressed indexes, such as generalized branching. Optimal speedups for some of those problems (and for others, like all-pairs suffix-prefix matching) have been obtained on classical suffix trees as well [10]. Here we show that one can obtain similar results on those problems and others, over a compressed representation that handles much larger texts in main memory. In Section 5 we further mitigate the space problem by introducing distributed compressed indexes. We show how CSAs and CSTs can be split across q machines, at some price in extra space and reasonable slowdown. The practical effect is that a much larger main memory is available, and compression helps reducing the number of machines across which the index needs to be distributed.

2 Basic Concepts

Fig. 1 illustrates the concepts in this section. We denote by T a **string**; by Σ the **alphabet** of size σ; by $T[i]$ the symbol at position $(i \bmod n)$ (so the first symbol of T is $T[0]$); by $T.T'$ the **concatenation**; by $T = T[..i-1].T[i..j].T[j+1..]$ respectively a **prefix**, a **subtring**, and a **suffix** of T.

The **path-label** of a node v, in a tree with edges labeled with strings over Σ, is the concatenation of the edge-labels from the root down to v. We refer indifferently to nodes and to their path-labels, also denoted by v. A **point** in $\mathcal{T}$ corresponds to any substring of T; this can be a node in $\mathcal{T}$ or a position within an edge label. The i-th letter of the path-label is denoted as $\textsc{Letter}(v, i) = v[i]$. The **string-depth** of a node v, denoted $\textsc{SDep}(v)$, is the length of its path-label, whereas the tree depth in number of edges is denoted $\textsc{TDep}(v)$. $\textsc{SLAQ}(v, d)$ is the highest ancestor of node v with $\textsc{SDep} \geq d$, and $\textsc{TLAQ}(v, d)$ is its ancestor of tree depth d. $\textsc{Parent}(v)$ is the parent node of v, whereas $\textsc{Child}(v, X)$ is the node that results of descending from v by the edge whose label starts with symbol X, if it exists. $\textsc{FChild}(v)$ is the first child of v, and $\textsc{NSib}(v)$ the next child of the same parent. $\textsc{Ancestor}(v, v')$ tells whether v is an ancestor of v', and $\textsc{LCA}(v, v')$ is the **lowest common ancestor** of v and v'.

The **suffix tree** of T is the deterministic compact labeled tree for which the path-labels of the leaves are the suffixes of $T\$$, where $\$$ is a terminator symbol not belonging to Σ. We will assume n is the length of $T\$$. The **generalized suffix tree** of T and T' is the suffix tree of $T\$T'\#$ where $\#$ is a new terminator symbol. For a detailed explanation see Gusfield's book [11]. The **suffix-link** of a node $v \neq \textsc{Root}$ of a suffix tree, denoted $\textsc{SLink}(v)$, is a pointer to node $v[1..]$. Note that $\textsc{SDep}(v)$ of a leaf v identifies the suffix of $T\$$ starting at position $n - \textsc{SDep}(v) = \textsc{Locate}(v)$. For example $T[\textsc{Locate}(ab\$)..] = T[7 - 3..] = T[4..] = ab\$$. The **suffix array** $A[0, n-1]$ stores the $\textsc{Locate}$ values of the leaves in lexicographical order. The *suffix tree nodes can be identified with suffix array intervals*: each node corresponds to the *range* of leaves that descend from v. The node b corresponds to the interval $[3, 6]$. Hence the node v will be represented by the interval $[v_l, v_r]$. Leaves are also represented by their left-to-right index (starting at 0). For example by $v_l - 1$ we refer to the leaf immediately before v_l, *i.e.* $[v_l - 1, v_l - 1]$. With this representation we can $\textsc{Count}$ in constant time the number of leaves that descend from v. The number of leaves below b is $4 = 6 - 3 + 1$. This is precisely the number of times that the string b occurs in the indexed string T. We can also compute $\textsc{Ancestor}$ in $O(1)$ time: $\textsc{Ancestor}(v, v') \Leftrightarrow v_l \leq v'_l \leq v'_r \leq v_r$. Operation $\textsc{Rank}_a(T, i)$ over a string T counts the number of times that the letter a occurs in T up to position i. Likewise, $\textsc{Select}_a(T, i)$ gives the position of the i-th occurence of a in T.

2.1 Parallel Computation Models

The parallel algorithms in this paper are studied under the Parallel Random Access Model (PRAM), which considers a set of independent sequential RAM processors, which have a private (or local) memory and a global shared memory. We assume that the RAMs can execute arithmetic and bitwise operations in constant time. We use the CREW model, where several processors can read the same shared memory cell simultaneously, but not write it. We call p the number of processors, and study the time used by the slowest processor.

For uniformity, distributed processing is studied by dividing the processors into q sites, which have a (fast) local memory and communicate using a (slow)

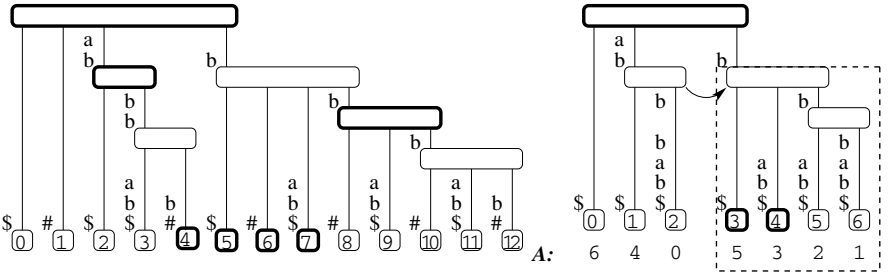


Fig. 1. Suffix tree $\mathcal{T}$ of string *abbbab* (right), with the leaves numbered. The arrow shows the SLink between node *ab* and *b*. Below it we show the suffix array. The portion of the tree corresponding to node *b* and respective leaves interval is highlighted with a dashed rectangle. The sampled nodes have bold outlines. We also show the generalized suffix tree for *abbbab* and *abbbb* (left), using the \$ and # terminators respectively.

```
((0)(1)((2)((3)(4)))((5)(6)(7)((8)(9)(10)(11)(12))))

    ( 0  1 ( 2 ( 3 (4))) (5)(6)(7)( 8  9  10  11  12 ) )
B : 1 0  0 1 0 1 0 10111 1011011011 0  0   0   0   0 1 1
B0: 1 0    1 0 1 0 1 111 1011 11011    0       0     1 1
B1: 1    0 1   1   10111 1 11011 11 0      0       0 1 1
```

Fig. 2. A parentheses representation of the generalized suffix tree in Fig. 1 (left), followed by the parentheses representation of the respective sampled tree $S_{\mathcal{C}}$. We also show B bitmap for the LSA operation of the sequential FCST and the B_i bitmaps for the LSA_i operations of the distributed FCST.

shared memory. The number of accesses to this slower memory are accounted for separately, and they can be identified with the amount of communication carried out on shared-nothing distributed models. We measure the local and total memory space required by the algorithms.

3 Overview of Sequential Fully-Compressed Suffix Trees

In this section we briefly explain the local FCST [7]. It consists of a compressed suffix array, a sampled tree S, and mappings between these structures.

Compressed suffix arrays (CSAs) are compact and functional representations of suffix arrays [4]. Apart from the basic functionality of retrieving $A[i] = \text{Locate}(i)$ (within a time complexity that we will call Φ), state-of-the-art CSAs support operation $\text{SLink}(v)$ *for leaves* v. This is called $\psi(v)$ in the literature: $A[\psi(v)] = A[v]+1$, and thus $\text{SLink}(v) = \psi(v)$, let its time complexity be Ψ. The iterated version of ψ, denoted ψ^i, can usually be computed faster than $O(i\Psi)$ with CSAs, since $\psi^i(v) = A^{-1}[A[v]+i]$. We assume the CSA also computes A^{-1} within $O(\Phi)$ time. CSAs might also support operation $\text{WeinerLink}(v, X)$ [12],

which for a node v gives the suffix tree node with path-label $X.v[0..]$. This is called the LF mapping in CSAs, and is a kind of inverse of ψ. Let its time complexity be $O(\tau)$. We extend LF to strings, $\mathrm{LF}(X.Y, v) = \mathrm{LF}(X, \mathrm{LF}(Y, v))$. For example, consider the interval $[3, 6]$ that represents the leaves whose path-labels start by b. In this case we have that $\mathrm{LF}(a, [3, 6]) = [1, 2]$, *i.e.* by using the LF mapping with a we obtain the interval of leaves whose path-labels start by ab.

CSAs also implement $\textsc{Letter}(v, i)$ for leaves v. $\textsc{Letter}(v, 0) = T[A[v]]$ is $v[0]$, the first letter of the path-label of leaf v. CSAs implement $v[0]$ in $O(1)$ time, and $\textsc{Letter}(v, i) = \textsc{Letter}(\textsc{SLink}^i(v), 0)$ in $O(\Phi)$ time. CSAs are usually self-indexes, meaning that they replace the text: they can extract any substring $T[i..i+\ell-1]$ in time $O(\Phi+\ell\Psi)$ time, since $T[i..i+\ell-1] = \textsc{Letter}(A^{-1}[i], 0..\ell-1)$.

We will use a CSA called the FMI [5], which requires $nH_k + o(n \log \sigma)$ bits, for any $k \leq \alpha \log_\sigma n$ and constant $0 < \alpha < 1$. It achieves $\Psi = \tau = O(1 + \frac{\log \sigma}{\log \log n})$ and $\Phi = O(\log n \log \log n)$.[2] The instantiation in Table 1 refers to the FMI.

The δ-sampled tree exploits the property that suffix trees are self-similar, $\textsc{SLink}(\mathrm{LCA}(v, v')) = \mathrm{LCA}(\textsc{SLink}(v), \textsc{SLink}(v'))$. A *$\delta$-sampled tree* S, from a suffix tree $\mathcal{T}$ of $\Theta(n)$ nodes, chooses $O(n/\delta)$ nodes such that, for each node v, node $\textsc{SLink}^i(v)$ is sampled for some $i < \delta$. Such a sampling can be obtained by choosing nodes with $\textsc{SDep}(v) = 0 \pmod{\delta/2}$ such that there is another node v' for which $v = \textsc{SLink}^{\delta/2}(v')$. Then the following equation holds, where $\mathrm{LCSA}(v, v')$ is the *lowest common sampled ancestor* of v and v':

$$\textsc{SDep}(\mathrm{LCA}(v, v')) = \max_{0 \leq i < \delta} \{i + \textsc{SDep}(\mathrm{LCSA}(\textsc{SLink}^i(v), \textsc{SLink}^i(v')))\} \quad (1)$$

From this relation the kernel operations are computed as follows. The i in LCA is the one that maximizes the computation in Eq. (1).

- $\textsc{SDep}(v) = \textsc{SDep}(\mathrm{LCA}(v, v)) = \max_{0 \leq i < d}\{i + \textsc{SDep}(\mathrm{LCSA}(\psi^i(v_l), \psi^i(v_r)))\}$,
- $\mathrm{LCA}(v, v') = \mathrm{LF}(v[0..i-1], \mathrm{LCSA}(\psi^i(\min\{v_l, v'_l\}), \psi^i(\max\{v_r, v'_r\})))$,
- $\textsc{SLink}(v) = \mathrm{LCA}(\psi(v_l), \psi(v_r))$.

These operations plus $\textsc{Parent}(v)$, which is easily computed on top of LCA, take time $O((\Psi + t)\delta)$. The exception is SDep, which takes $O(\Psi\delta)$.

Note that we have to solve LCSA. This requires mapping nodes to their lowest sampled ancestors in S, an operation called LSA we explain next. In addition, each sampled node v must store its $[v_l, v_r]$ interval, its $\textsc{Parent}_S$ and $\textsc{TDep}_{\mathcal{T}}$, and also compute LCA_S queries in constant time. All this takes $O((n/\delta) \log n)$ bits. The rest is handled by the CSA.

Computing lowest sampled ancestors. Given a CSA interval $[v_l, v_r]$ representing node v of $\mathcal{T}$, the *lowest sampled ancestor* $\mathrm{LSA}(v)$ gives the lowest sampled tree node containing v. With LSA we can compute $\mathrm{LCSA}(v, v') = \mathrm{LCA}_S(\mathrm{LSA}(v), \mathrm{LSA}(v'))$.

[2] $\psi(i)$ can be computed as $select_{T[A[i]]}(T^{bwt}, T[A[i]])$ using the wavelet tree [13]. The cost for Φ assumes a sampling step of $\log n \log \log n$, which adds $o(n)$ extra bits.

Table 1. Comparing local and distributed FCST representations. The operations are defined in Section 2. Time complexities, but not space, are big-O expressions. The dominant terms in the distributed times count slow-memory accesses. We give the generalized performance and an instantiation using $\delta = \log n \log\log n$, assuming $\sigma = O(\text{polylog}(n))$, and using the FMI [5] as the CSA.

	Local	Distributed
Space in bits	$\lvert CSA\rvert + O((n/\delta)\log n)$ $= nH_k + o(n\log\sigma)$	$\lvert CSA\rvert + O((n/\delta)\log n + n\log(1+q/\delta))$ $= nH_k + o(n\log\sigma) + O(n\log q)$
SDep	$\Psi\delta = \log n\log\log n$	$(\log q + \Psi + \tau)\delta$ $= \log q\log n\log\log n$
Count	$1 = 1$	$\log q = \log q$
Ancestor	$1 = 1$	$1 = 1$
Parent/ FChild/ NSib/ SLink/ LCA	$(\Psi+\tau)\delta = \log n\log\log n$	$(\log q + \Psi + \tau)\delta$ $= \log q\log n\log\log n$
SLink^i	$\Phi + (\Psi+\tau)\delta$ $= \log n\log\log n$	$\Phi + (\log q + \Psi + \tau)\delta$ $= \log q\log n\log\log n$
Letter(v,i)	$\Phi = \log n\log\log n$	$\Phi = \log n\log\log n$
Child	$\Phi\log\delta + (\Psi+\tau)\delta + \log(n/\delta)$ $= \log n(\log\log n)^2$	$\Phi\log\delta + (\Psi+\tau)\delta + \log(n/\delta)$ $= \log n(\log\log n)^2$
TDep	$(\Psi+\tau)\delta^2$ $= (\log n\log\log n)^2$	$(\log q + \Psi + \tau)\delta^2$ $= \log q\,(\log n\log\log n)^2$
TLAQ	$\log n + (\Psi+\tau)\delta^2$ $= (\log n\log\log n)^2$	$\log n + (\log q + \Psi + \tau)\delta^2$ $= \log q\,(\log n\log\log n)^2$
SLAQ	$\log n + (\Psi+\tau)\delta$ $= \log n\log\log n$	$\log n + (\log q + \Psi + \tau)\delta$ $= \log q\log n\log\log n$
WeinerLink	$\tau = 1$	$\tau = 1$

The key component for these operations is a bitmap B that is obtained by writing a 1 whenever we find a '(' or a ')' in the parentheses representation of the sampled tree S and 0 whenever we find a leaf of $\mathcal{T}$, see Fig. 2. Then the LSA is computed via Rank/Select on B. As B contains $m = O(n/\delta)$ ones and n zeros, it can be stored in $m\log(n/m) + O(m + n\log\log n/\log n) = O((n/\delta)\log\delta) + o(n)$ bits [14]. We now present a summary of the FCST representation.

Theorem 1. *Using a compressed suffix array (CSA) that supports ψ, ψ^i, $T[A[v]]$ and* LF *in times $O(\Psi)$, $O(\Phi)$, $O(1)$, and $O(\tau)$, respectively, it is possible to represent a suffix tree with the properties given in Table 1 (column "local").*

4 Parallel Compressed Indexes

In this section we study the situation where the index resides in main memory and we want to speed up its main search operations. We start by studying exact matching, matching statistics and longest common substrings over CSAs and FCSTs, and finish with maximal repeats (only over FCSTs).

The CSA's basic operations, such as ψ, LF, $A[i]$ and $A^{-1}[i]$, seem to be intrinsically sequential. However using *generalized branching* we can speed up several algorithms. The generalized branching $\textsc{Child}(v_1, v_2)$, for suffix tree points v_1 and v_2, is the point with path label $v_1.v_2$ if it exists. This operation was first considered by Huynh et al. [15], who achieved $O(\Phi \log n)$ time over a CSA. Russo et al. [16] achieved $O((\Psi+\tau)\delta + \Phi \log \delta + \log(n/\delta))$ time on the FCST. The procedure binary searches the interval of v_1 for the subinterval where $\psi^{\textsc{SDep}(v_1)} \in v_2$. With the information stored at sampled nodes, only an interval of size δ is binary searched this way; the other $O(\log(n/\delta))$ steps require constant-time accesses.

These binary searches can be accelerated using p processors. Instead of dividing the interval into two pieces we divide it into p and assign each comparison to a different processor. The time complexity becomes $\Pi(p) = O((\Psi+\tau)\delta + \Phi \log_p \delta + \log_p(n/\delta))$. The CSA-based algorithm also improves to $\Pi(p) = O(\Phi \log_p n)$.

Pattern matching. Assume we want to search for a pattern P of size m. We divide P into p parts and assign one to each processor. Each piece is searched for, like in the FMI, with the LF operation. This requires $O(m\tau/p)$ time. We then join the respective intervals with a binary tree of generalized Child operations. Assume for simplicity that m/p is a power of 2. We show a flow-graph of this procedure in Fig. 3. We first concatenate the $p/2$ pairs of leaves, using 2 processors for the generalized branching operation, using time $\Pi(2)$. We then concatenate the $p/4$ pairs of nodes of height 1, using 4 processors for the generalized branching, using time $\Pi(4)$. We continue until merging the two halves of P using p processors. The overall time of the hierarchical concatenation process is $O((\Psi+\tau)\delta \log p + (\Phi \log \delta + \log(n/\delta)) \log\log p)$ on the FCST and $O(\Phi \log n \log\log p)$ on the bare FMI. Using the same instantiation as in Table 1 this result becomes $O(m/p + \log n \log\log n(\log p + \log\log n \log\log p))$ time on the FCST and $O(m/p + \log^2 n \log\log n \log\log p)$ on the FMI, both in $nH_k + o(n \log \sigma)$ bits of index space. The speedup is linear in p, except for the polylogarithmic additive term. On the other hand, there is no point in using more than m processors; the optimum is achieved using less than m.

Matching statistics. The matching statistics $m(i)$ indicate the size of the longest prefix of $P[i..]$ that is a substring of T. Consider for example the string $P = abbbbabb$, using the running example suffix tree $\mathcal{T}$, right of Fig. 1. The corresponding matching statistics are $4, 3, 5, 4, 3, 3, 2, 1$. To compute these values we will again resort to the generalized Child operation. As before the idea is to first compute a generalized branch tree, which is the tree in the flow-graph of Fig. 3. This tree contains the intervals over CSA that correspond to strings $P[2^j k..2^j(k+1)-1]$, where j will indicate the level in the tree and k the position in the level. The levels and the positions start at 0. Notice that constructing this tree can be done exactly as for pattern matching, except that we do not stop the tree at pieces of length m/p but continue up to length 1. Since the subtrees for pieces of size m/p must be handled sequentially by one processor, they require additional time $O(m\tau/p + (m/p)\Pi(2))$. Note each node of this branching tree stores the suffix tree point (or suffix array interval plus length) that corresponds to its substring, if it exists, and Null otherwise.

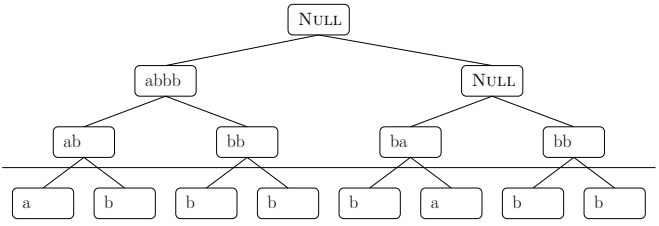


Fig. 3. The flow-graph for parallel exact matching and matching statistics is a tree of generalized Child operations for pattern $P = abbbbabb$ and $p = 4$ processors. Matching statistics use all the operations in the tree, whereas exact matching performs only the operations above the line and the search below the line is computed with LF operations.

After building the tree we traverse it m times, once for each $P[i..]$. For each such i we find $m(i)$ by traversing the tree path that covers $P[i..i + m(i) - 1]$ with the maximal nodes. This describes a path that ascends and then descends, touching $O(\log m)$ nodes of the branching tree. We start at the ith leaf x of the branching tree, which corresponds to the letter $P[i]$, with $v = P[i]$ the current point of $\mathcal{T}$, and move up to the parent z of x. If x is the right child of z, we do nothing more than $x \leftarrow z$. If x is the left child of z, we do a generalized $\textsc{Child}(v, u)$ operation, where u is the point of $\mathcal{T}$ that is stored in y, the right child of z. If the resulting point is not Null we set v to this new point, $v \leftarrow \textsc{Child}(v, u)$, and continue moving up, $x \leftarrow z$. Otherwise we start moving down on y, $x \leftarrow y$. While we are moving down we compute the generalized Child operation between v and the point u in the left child y of x. If the resulting point is still Null we move to the left child of x, $x \leftarrow y$. Otherwise we set v to this new point, $v \leftarrow \textsc{Child}(v, u)$, and move to the right child z of x, $x \leftarrow z$. The value $m(i)$ is obtained by initializing it at zero and adding 2^j to it each time we update v at level j of the branching tree.

For example, assume we want to compute $m(2)$, *i.e.* we want to determine the longest prefix of $bbbabb$ that is a substring of T. We start on the third leaf of the tree in Fig. 3 and set $v = b$. Then move up. Since we are moving from a left child we compute $\textsc{Child}(b, b)$ and obtain $v = bb$. Again we move up but this time we are moving from a right child so we do nothing else. We move up again. Since the interval on the right sibling is Null the Child operation also returns Null. Therefore we start descending in the right sibling. We now consider the left child of that sibling, and compute $\textsc{Child}(v, ba) = \textsc{Child}(bb, ba) = bbba$. Since this node is not Null we set v to it and move to the node labeled bb. Considering its left child we compute $\textsc{Child}(bbba, b) = bbbab$. Since it is not Null we set v to it and move to rightmost leaf. Finally we check whether $\textsc{Child}(bbbab, b) \neq \textsc{Null}$ since this is that case we know that we should consider the rightmost leaf as part of the common substring. This means that $m(2) = 5 = 2^0 + 2^0 + 2^1 + 2^0 = 7 - 2$.

Traversing the tree takes $O(\Pi(2) \log m)$ time per traversal, thus with p processors the time is $O((m/p)\Pi(2) \log m)$. By considering that only $p \leq m$ processors are useful, we have that the traversal time dominates the branching

tree construction time. Using the instantiation in Table 1 this result becomes $O((m/p)\log m \log n(\log\log n)^2)$ on the FCST and $O((m/p)\log m \log^2 n \log\log n)$ on the FMI. The total space is $O(m\log m) + nH_k + o(n\log\sigma)$ bits. This time the linear speedup is multiplied by a polylogarithmic factor, since the sequential algorithm can be made $O(m)$ time.

Longest common substring. We can compute the longest common substring between P and T by taking the maximum matching statistic $m(i)$ in additional negligible $O(m/p + \log p)$ time.

Maximal repeats. For this problem we need a FCST, and cannot simulate it with a CSA as before. A maximal repeat in T is a substring S, of size ℓ, that occurs in at least two positions i and i', *i.e.* $S = T[i..i+\ell-1] = T[i'..i'+\ell-1]$, and cannot be extended either way, *i.e.* $T[i-1] \neq T[i'-1]$ and $T[i+\ell] \neq T[i'+\ell]$. The solution for this problem consists in identifying the deepest internal nodes v of $\mathcal{T}$ that are left-diverse, *i.e.* the nodes for which there exist letters $X \neq Y$ such that $X.v$ and $Y.v$ are substrings of T [11]. Assume $v = [v_l, v_r]$. Then FMIs allow one to access $\textsc{Letter}(v_l, -1) = T[v_l - 1]$ in $O(\tau)$ time[3]. Hence node v is left-diverse iff $\textsc{Count}(\textsc{LF}(\textsc{Letter}(v_l, -1), v)) \neq \textsc{Count}(v)$. This can be verified in $O(\tau)$ time, moreover this verification can be performed independently for every internal node of $\mathcal{T}$. At each step the algorithm chooses p nodes from $\mathcal{T}$ and performs this verification. A simple way to choose all internal nodes (albeit with repetitions, which does not affect the asymptotic time of this algorithm) is to compute $\text{LCA}([i,i],[i+1,i+1])$ for all $0 \leq i < n-1$. Hence this procedure requires $O((n/p)(\Psi+\tau)\delta)$ time, plus negligible $O(n/p + \log p)$ time to find the longest candidates. Using the same instantiation as in Table 1 the result becomes $O((n/p)\log n \log\log n)$ time within optimal $nH_k + o(n\log\sigma)$ overall bits. This is an optimal speedup, if we consider the polylogarithmic penalty of using a FCST.

The speedups in this section are similar to the results obtained for "classical" uncompressed suffix trees by Clifford [10], which do not speed up exact matching because they do not use a generalized Child operation. Clifford speeds up the longest common substring problem and the maximal repeats, among others.

5 Distributed Compressed Indexes

In this section we study distributed CSAs and FCSTs, mainly to obtain support for large string databases. In this case we assume we have a collection $\mathcal{C}$ of q texts of total length n, distributed across q machines. Hence distributed FCSTs are always generalized suffix trees, and likewise for CSAs. In fact, the local text of each machine could also be a collection of smaller texts, and the whole database could be a single string arbitrarily partitioned into q segments: CSAs and CSTs treat both cases similarly. The only difference is whether the SLink of the last symbol of a text sends one to the next text or it stays within that text, but either variant can be handled with minimal changes. We choose the latter option.

[3] This is an access to the Burrows-Wheeler transform.

Various data layouts have been considered for distributing classical suffix trees and arrays [17,9,10]. One can distribute the texts and leave each machine index its own text, or distribute a single global index into lexicographical intervals, or opt for other combinations. In this paper we consider reducing the time of a single query, in contrast to previous work [17] where the focus is on speeding up batches of queries by distributing them across machines. Our approach is essentially that of indexing each text piece in its own machine, yet we end up distributing some global information across machines, similarly to the idea storing local indexes with global identifiers [17].

First we study the case where the local CSAs are used to simulate a global CSA, which can then be used directly in the FCST representation. This solution turns out to require extra space due to the need of storing some redundant information. Then we introduce a new technique to combine FCSTs that removes some of those redundant storage requirements.

Distributed Compressed Suffix Arrays. Assume we have a collection $\mathcal{C} = \{T_j\}_0^{q-1}$, and the respective local CSAs. We denote their operations with a subscript j, *i.e.* as A_j, A_j^{-1}, ψ_j and LF_j. The generalized CSA that results from this collection is denoted $A_{\mathcal{C}}$. Assume we store the accumulated text sizes, $AccT[i] = \sum_{j=0}^{i-1} |T_j|$, which need just $O(q \log n)$ bits.

We define the sequence $Id_{\mathcal{C}}$ of suffix indexes of $\mathcal{C}$, where $Id_{\mathcal{C}}[i] = j$ if the suffix in $A_{\mathcal{C}}[i]$ belongs to text T_j. Consider T as T_0 and T' as T_1 in our running example. The respective generalized suffix tree is shown in the left of Fig. 1. The Id sequence for this example is obtained by reading the leaves, and replacing \$ by 0 and # by 1. The resulting sequence is $Id = 0100101010101$.

If we process Id for Rank and Select queries we can obtain the operations of $A_{\mathcal{C}}$ from the operations of the A_j's. To compute Locate we use the equation $A_{\mathcal{C}}[v] = A_{Id[v]}[\mathrm{Rank}_{Id[v]}(v-1)] + AccT[Id[v]]$. For example for $A_{\mathcal{C}}[4]$ we have that $A_1[\mathrm{Rank}_1(4-1)] + AccT[1] = A_1[1] + 7 = 7$. To compute $A_{\mathcal{C}}^{-1}$ we use a similar relation, $A_{\mathcal{C}}^{-1}[i] = \mathrm{Select}_j(A_j^{-1}[i - AccT[j]] + 1)$, where j is such that $AccT[j] \leq i < AccT[j+1]$. Likewise $\psi_{\mathcal{C}}$ is computed as $\psi_{\mathcal{C}}[v] = \mathrm{Select}_{Id[v]}(\psi_{Id[v]}[\mathrm{Rank}_{Id[v]}(v-1)] + 1)$. Computing $\mathrm{LF}_{\mathcal{C}}(X, [v_l, v_r])$ is more complicated: we compute LF in all the CSAs, *i.e.* $\mathrm{LF}_j(X, [\mathrm{Rank}_j(v_l - 1), \mathrm{Rank}_j(v_r) - 1])$ for every $0 \leq j < q$. If $[xv_{j,l}, xv_{j,r}]$ are the resulting intervals then $\mathrm{LF}_{\mathcal{C}}(X, [v_l, v_r]) = [\min_{j=0}^{q-1}\{\mathrm{Select}_j(xv_{j,l}+1)\}, \max_{j=0}^{q-1}\{\mathrm{Select}_j(xv_{j,r}+1)\}]$. Consider for example, how to compute $\mathrm{LF}_{\mathcal{C}}(a, [5, 12])$. We compute $\mathrm{LF}_0(a, [3, 6]) = [1, 2]$ and $\mathrm{LF}_1(a, [2, 5]) = [1, 1]$ and use the results to obtain that $\mathrm{LF}_{\mathcal{C}}(a, [5, 12]) = [\min\{\mathrm{Select}_0(1+1), \mathrm{Select}_1(1+1)\}, \max\{\mathrm{Select}_0(2+1), \mathrm{Select}_1(1+1)\}] = [\min\{2, 4\}, \max\{3, 4\}] = [2, 4]$. This requires $O(\log q)$ accesses to slow memory to compute minima and maxima in parallel.

A problem with this approach is the space necessary to store sequence Id and support Rank and Select. An efficient approach is to unfold Id into q bitmaps, $BId_j[i] = 1$ iff $Id[i] = j$, and process each one for constant-time binary Rank and Select queries while storing them in compressed form [14]. Then since BId_j contains about n/q 1s, it requires $(n/q) \log q + O(n/q) + o(n)$ bits of space. We store each BId_j in the local memory of processor j, which requires

space $|CSA_j| + (n/q)\log q + O(n/q) + o(n)$ local bits. The total space usage is $|CSA_{\mathcal{C}}| + n\log q + O(n) + o(qn)$ bits (if the partitions are not equal it is even less; $n\log q$ bits is the worst case). This essentially lets each machine map its own local CSA positions to the global suffix array, as done in previous work for classical suffix arrays (where the global identifiers can be directly stored) [17].

In this setup, most of the accesses are to local memory. One model is that queries are sent to all processors and the one able of handling it takes the lead. For $A_{\mathcal{C}}[v]$, each processor j looks if $BId_j[v] = 1$, in which case $j = Id[v]$ and this is the processor solving the query locally, in $O(\Phi)$ accesses to fast memory (processor j also stores values $AccT[j]$ and $AccT[j+1]$ locally). For $A_{\mathcal{C}}^{-1}[i]$, each processor j checks if $AccT[j] \leq i < AccT[j+1]$ and the one answering positively takes the lead, answering again in $O(\Phi)$ local accesses. $\psi_{\mathcal{C}}$ proceeds similarly to $A_{\mathcal{C}}$, in $O(\Psi)$ local accesses. $\mathrm{LF}_{\mathcal{C}}$ is more complex since all the processors must be involved, each spending $O(\tau)$ local accesses, and then computing global minima and maxima in $O(\log q)$ accesses to slow memory. Compare to the alternative of storing $CSA_{\mathcal{C}}$ explicitly and splitting it lexicographically: all the local accesses in the time complexities become global.

The $o(qn)$ extra memory scales badly with q (as more processors are available, each needs more local memory). A way to get rid of it is to use bitmap representations that require $n\log q + o(n\log q) + O(n\log\log q) = O(n\log q)$ bits and solve Rank and Select queries within $o((\log\log n)^2)$ time [18]. We will now present a new technique that directly represents global FCSTs using tuples of ranges instead of a single suffix array range.

Distributed Fully-Compressed Suffix Trees. Consider the generalized suffix tree $\mathcal{T}_{\mathcal{C}}$ of a collection of texts $\mathcal{C} = \{T_j\}_0^{q-1}$ and the respective individual suffix trees $\mathcal{T}_i$. Assume, also, that we are storing the $\mathcal{T}_i$ trees with the FCST representation and want to obtain a representation for $\mathcal{T}_{\mathcal{C}}$. A node of $\mathcal{T}_{\mathcal{C}}$ can be represented all the time as a q-tuple of intervals $\langle v_0, \ldots, v_{q-1}\rangle = \langle [v_{0,l}, v_{0,r}], \ldots, [v_{q-1,l}, v_{q-1,r}]\rangle$ over the corresponding CSAs. For example the node *abbb* can be represented as $\langle [2,2],[1,1]\rangle$. In fact we have just explained, in the distributed LF operation, how to obtain from these intervals the $[v_l, v_r]$ representation of node v of $\mathcal{T}_{\mathcal{C}}$ (via Select on $Id_{\mathcal{C}}$ and distributed minima and maxima). Thus these intervals are enough to represent v.

To avoid storing the Id sequence we map every interval $[v_{i,l}, v_{i,r}]$ directly to the sampled tree of $FCST_{\mathcal{C}}$, instead of mapping it to an interval v over $CSA_{\mathcal{C}}$ and then reducing it to the sampled tree of $FCST_{\mathcal{C}}$ with $\mathrm{LSA}_{\mathcal{C}}(v)$. We use the same bitmap-based technique for $\mathrm{LSA}_{\mathcal{C}}$, but store q local bitmaps instead of just a global one. The bitmaps B_j are obtained from the bitmap B of the $FCST_{\mathcal{C}}$ by removing the zeros that do not correspond to leaves of T_j, see Fig 2. This means that, in B_j, we are representing the $O(n/\delta)$ nodes of the global sampled tree and the n/q leaves of T_j. As each B_j has n/q 0s and $O(n/\delta)$ 1s, the compressed representation [14] supporting constant-time Rank and Select requires $(n/q)\log(1+q/\delta) + O(n/q) + o(n/\delta + n/q)$ bits. This is slightly better than the extra space of CSAs, totalling $O(n\log(1+q/\delta)) + o(nq/\delta)$ bits. As before, the $o(\ldots)$ term can be removed by using the representation by Gupta et

al. [18] at the price of $o((\log\log n)^2)$ accesses to fast local memory. Now the same computation for LSA carried out on B_j gives a global interval.

We then compute $\mathrm{LSA}_{\mathcal{C}}(v) = \mathrm{LCA}_{S_{\mathcal{C}}}(\mathrm{LSA}_0(v_0), \ldots, \mathrm{LSA}_{q-1}(v_{q-1}))$, where $\mathrm{LCA}_{S_{\mathcal{C}}}$ is the LCA operation over the sampled tree of $\mathcal{T}_{\mathcal{C}}$ and $\mathrm{LSA}_j(v_j)$ is the global LSA value obtained by processor j. This operation is computed in parallel in $O(\log q)$ accesses to slow memory (which replaces the global minima/maxima of the CSA). The sampled tree S and its extra data (e.g., to compute LCA in constant time) is stored in the shared memory. Hence accesses to S are always slow, which does not change the stated complexities. This mechanism supports the usual representation of the global FCST.

Consider, for example, that we want to compute the SDEP of node *abbb*. Note that the SDEP of $[2,2]$ in $\mathcal{T}_0$ is 7 and that the SDEP of $[1,1]$ in $\mathcal{T}_1$ is 6. However the SDEP of *abbb* in $\mathcal{T}_{\mathcal{C}}$ is 4. In this example we do not have to use ψ to obtain the result, altough in general it is necessary. By reducing the $[2,2]$ and $[1,1]$ intervals to the sampled tree of $FCST_{\mathcal{C}}$ we obtain the node *abbb* and the leaf *abbbb#*, see Fig. 1. The node we want is the LCA of these nodes, *i.e. abbb*.

Theorem 2. *Given a collection of q texts $\mathcal{C} = \{T_j\}_0^{q-1}$ represented by compressed suffix arrays (CSA_j) that support ψ, ψ^i, $T[A[v]]$ and* LF *in times $O(\Psi)$, $O(\Phi)$, $O(1)$, and $O(\tau)$, respectively, it is possible to represent a distributed suffix tree with the properties given in Table 1 (column "distributed").*

Moreover this technique has the added benefit that we can simulate the generalized suffix tree from any subcollection of the q texts, by using only the intervals of the texts T_j that we want to consider. However in this case we lose the TDEP, TLAQ and SLAQ operations.

6 Conclusions and Future Work

Compressed indexes are a new and functional way to index text strings using little space, and their parallelization has not been studied yet. We have focused on parallel (shared RAM) and distributed suffix trees and arrays, which are the most pervasive compressed text indexes. We obtained almost linear speedups for the basic pattern search problem, and also for more complex ones such as computing matching statistics, longest common substrings, and maximal matches. The sequential algorithms for these problems are linear-time and easy to carry over compressed indexes, but hard to parallelize. Thanks to the stronger functionality of compressed indexes, namely the support of generalized branching, we achieve parallel versions for all of these. Some of our solutions can do with a compressed suffix array; others require a compressed suffix tree. We plan to apply this idea to other problems with applications in bioinformatics [11], such as all-pairs prefix-suffix queries.

Distributing the index across q machines further alleviates the space problem, allowing it to run on a larger virtual memory. Our distributed suffix arrays require $O(n \log q) + o(n)$ extra bits, whereas our suffix trees require $o(nq/\delta)$ extra bits. Both simulate a global index with $O(\log q)$ slowdown (measured in communication cost), so they achieve $O(q/\log q)$ speedup on each query.

A challenge for future work is to reduce this extra space, as $O(n \log q)$ can be larger than the compressed suffix array itself. We also plan to consider other models models such as BSP and batched queries [17]. An exciting direction is to convert the distributed index into an efficient external-memory representation for compressed text indexes, which suffer from poor locality of reference.

References

1. Giegerich, R., Kurtz, S., Stoye, J.: Efficient implementation of lazy suffix trees. Softw., Pract. Exper. 33(11), 1035–1049 (2003)
2. Manber, U., Myers, E.: Suffix arrays: A new method for on-line string searches. SIAM J. Comput. 22(5), 935–948 (1993)
3. Sadakane, K.: Compressed suffix trees with full functionality. Theory Comput. Syst. 41(4), 589–607 (2007)
4. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Comp. Surv. 39(1), article 2 (2007)
5. Ferragina, P., Manzini, G., Mäkinen, V., Navarro, G.: Compressed representations of sequences and full-text indexes. ACM Trans. Algor. 3(2), article 20 (2007)
6. Manzini, G.: An analysis of the Burrows-Wheeler transform. J. ACM 48(3), 407–430 (2001)
7. Russo, L., Navarro, G., Oliveira, A.: Fully-Compressed Suffix Trees. In: Laber, E.S., Bornstein, C., Nogueira, L.T., Faria, L. (eds.) LATIN 2008. LNCS, vol. 4957, pp. 362–373. Springer, Heidelberg (2008)
8. Fischer, J., Mäkinen, V., Navarro, G.: Faster entropy-bounded compressed suffix trees. Theor. Comp. Sci. 410(51), 5354–5364 (2009)
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. Clifford, R.: Distributed suffix trees. J. Discrete Algorithms 3(2-4), 176–197 (2005)
11. Gusfield, D.: Algorithms on Strings, Trees and Sequences. Cambridge University Press, Cambridge (1997)
12. Weiner, P.: Linear pattern matching algorithms. In: IEEE Symp. on Switching and Automata Theory, pp. 1–11 (1973)
13. Lee, S., Park, K.: Dynamic rank-select structures with applications to run-length encoded texts. In: Ma, B., Zhang, K. (eds.) CPM 2007. LNCS, vol. 4580, pp. 95–106. Springer, Heidelberg (2007)
14. Raman, R., Raman, V., Rao, S.S.: Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In: Proc 13th SODA, pp. 233–242 (2002)
15. Huynh, T.N.D., Hon, W.K., Lam, T.W., Sung, W.K.: Approximate string matching using compressed suffix arrays. Theor. Comput. Sci. 352(1-3), 240–249 (2006)
16. Russo, L., Navarro, G., Oliveira, A.: Dynamic Fully-Compressed Suffix Trees. In: Ferragina, P., Landau, G.M. (eds.) CPM 2008. LNCS, vol. 5029, pp. 191–203. Springer, Heidelberg (2008)
17. Marín, M., Navarro, G.: Distributed query processing using suffix arrays. In: Nascimento, M.A., de Moura, E.S., Oliveira, A.L. (eds.) SPIRE 2003. LNCS, vol. 2857, pp. 311–325. Springer, Heidelberg (2003)
18. Gupta, A., Hon, W.K., Shah, R., Vitter, J.: Compressed data structures: dictionaries and data-aware measures. In: Àlvarez, C., Serna, M. (eds.) WEA 2006. LNCS, vol. 4007, pp. 158–169. Springer, Heidelberg (2006)

Author Index