DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Proceedings of the Prague Stringology Conference '06

Edited by Jan Holub and Jan Žďárek

Department of Computer Science and Engineering

Czech Technical University in Prague
Czech Republic

Proceedings
of the Prague Stringology Conference '06

Edited by Jan Holub and Jan Žďárek

August 2006

Prague Stringology Club
`http://www.stringology.org/`

Department of Computer Science and Engineering
Faculty of Electrical Engineering
Czech Technical University in Prague
Karlovo náměstí 13, Praha 2, 121 35, Czech Republic.

Program Committee

Amihood Amir	(Bar-Ilan University, Israel)
Gabriela Andrejková	(P. J. Šafárik University, Slovakia)
Jun-ichi Aoe	(Tokushima University, Japan)
Maxime Crochemore	(Université de Marne la Vallée, France)
František Franěk	(McMaster University, Canada)
Jan Holub, *Co-chair*	(Czech Technical University in Prague, Czech Republic)
Costas S. Iliopoulos	(King's College London, United Kingdom)
Shmuel T. Klein	(Bar-Ilan University, Israel)
Thierry Lecroq	(Université de Rouen, France)
Bořivoj Melichar, *Co-chair*	(Czech Technical University in Prague, Czech Republic)
Yoan J. Pinzon	(King's College London, United Kingdom)
Marie-France Sagot	(Université Claude Bernard, Lyon, France)
Bruce W. Watson	(Technische Universiteit Eindhoven, Netherlands)

Organizing Committee

Miroslav Balík, Jan Holub, Bořivoj Melichar, Michal Voráček and Jan Žďárek

External Referees

Saïd Abdeddaïm	Inuka Jayasekera	Simon Puglisi
Pavlos Antoniou	Miriam Koppel	Sohel Rahman
Atlam El-Sayed	Derrick G. Kourie	William F. Smyth
Dominique Cellier	Manal Mohamed	Tinus Strauss
Loek Cleophas	Ernest Ketcha Ngassam	Fritz Venter

Conference Sponsors

Czech Technical University in Prague

LEDA Publishing House

Proceedings of the Prague Stringology Conference '06
Edited by Jan Holub and Jan Žďárek
Published by: Prague Stringology Club
Department of Computer Science and Engineering
Faculty of Electrical Engineering
Czech Technical University in Prague
Karlovo náměstí 13, Praha 2, 121 35, Czech Republic.
URL: `http://www.stringology.org/`
E-mail: `psc@cs.felk.cvut.cz` Phone: +420-2-2435-7470
Printed by Česká technika – Naklatelství ČVUT, Thákurova 550/1, Praha 6, 160 41, Czech Republic

ISBN 80-01-03533-6

Preface

The Prague Stringology Conference 2006 (PSC'06) was held at the Department of Computer Science and Engineering of the Czech Technical University in Prague, Czech Republic, on August 28–30, 2006. The conference focused on stringology and related topics. Stringology is a discipline concerned with algorithmic processing of strings and sequences.

The papers submitted were reviewed by the program committee and twenty were selected for presentation at the conference, based on originality and quality. This volume contains not only these selected papers but also abstract of one invited talk devoted to *dist tables*.

PSC'06 is the eleventh event of the Prague Stringology Club. In the years 1996–2000 the Prague Stringology Club Workshops (PSCW's) and the Prague Stringology Conferences (PSC's) in 2001–2005 preceded this conference. The proceedings of these workshops and the conferences had been published by Czech Technical University in Prague and are available on WWW pages of the Prague Stringology Club. Selected contributions were published in special issues of the journal Kybernetika, the Nordic Journal of Computing, the Journal of Automata, Languages and Combinatorics, and the International Journal of Foundations of Computer Science.

The Prague Stringology Club was founded in 1996 as a research group at the Department of Computer Science and Engineering of the Czech Technical University in Prague. The goal of the Prague Stringology Club is to study algorithms on strings and sequences with emphasis on finite automata theory. The first event organized by the Prague Stringology Club was the workshop PSCW'96 featuring only a handful invited talks. However, since PSCW'97 the papers and talks are selected by a rigorous peer review process. The objective is not only to present new results in stringology, but also to facilitate personal contacts among the people working on these problems.

I would like to thank all those who had submitted papers for PSC'06 as well as the reviewers. Special thanks goes to all the members of the program committee, without whose efforts it would not have been possible to put together such a stimulating program of PSC'06. Last, but not least, my thanks go to the members of the organizing committee for ensuring such a smooth running of the conference.

In Prague, Czech Republic
on August 2006
Jan Holub

Table of Contents

Invited Talk

Contributed Talks

Can Dist Tables Be Merged in Linear Time An Open Problem

(*Invited Talk*)

Gad M. Landau

Department of Computer Science
Faculty of Social Sciences
University of Haifa
31905 Haifa, Israel
landau@cs.haifa.ac.il

Dist tables are key players in the computation of dynamic programming tables in $o(n^2)$ time. Given two strings A and T, $dist(A, T)$ stores the scores of the edit distances between T and all substrings of A. Given $dist(A, T)$ and $dist(B, T)$ (strings A and B are each of length m and T is of length n) the best known algorithms that compute $dist(AB, T)$ run in $\mathcal{O}(nm)$ time or $\mathcal{O}(n^{1.5})$ time. We will discuss the use of dist tables and Schmidt and Tiskin's Algorithms as well as some thoughts on possible directions to answering the open problem.

An Asymptotic Lower Bound for the Maximal-Number-of-Runs Function

Frantisek Franek⋆ and Qian Yang

Department of Computing & Software
Faculty of Engineering
McMaster University
Hamilton, Ontario
Canada L8S 4K1
franek@mcmaster.ca, yangq6@univmail.cis.mcmaster.ca

Abstract. An asymptotic lower bound for the maxrun function $\boldsymbol{\rho}(n) = \mathbf{max}\,\{number\ of\ runs\ in\ string\ \boldsymbol{x} \mid all\ strings\ \boldsymbol{x}\ of\ length\ n\}$ is presented. More precisely, it is shown that for any $\varepsilon > 0$, $(\alpha{-}\varepsilon)n$ is an asymptotic lower bound, where $\alpha = \frac{3}{1+\sqrt{5}} \approx 0.927$. A recent construction of an increasing sequence of binary strings "rich in runs" is modified and extended to prove the result.

Keywords: run, lower bound, asymptotic lower run, maximum number of runs

1 Introduction

An important structural characteristic of a string over an alphabet is its periodicity. Repetitions (tandem repeats) have always been in the focus of the research into periodicities. The notion of runs captures maximal repetitions which themselves are not repetitions and allows for a succinct notation ([5]). Even though it had been known that there could be $O(n \log n)$ of repetitions in a string of length n ([1]), it was shown in 2000 by Kolpakov and Kucherov that number of runs was linear in the length of the input string ([4]). Their proof was existential and thus did not specify the constants of linearity. The behaviour of the **maxrun function** $\boldsymbol{\rho}(n) = \mathbf{max}\{number\ of\ runs\ in\ string\ \boldsymbol{x} \mid all\ strings\ \boldsymbol{x}\ of\ length\ n\}$ became an interest of study to many. Smyth et al. (e.g. [3], [6], [2]) presented a set of conjectures about $\boldsymbol{\rho}(n)$:

1. $\boldsymbol{\rho}(n) < n$,
2. $\boldsymbol{\rho}(n{+}1) \leq \boldsymbol{\rho}(n){+}2$,
3. $\boldsymbol{\rho}(n) = \boldsymbol{\rho}_2(n)$, the maxrun function for binary strings.

Just recently, Rytter improved the upper bound of $\boldsymbol{\rho}(n)$ to $6.3n$ (see [7]).

[3] introduced a construction of an increasing sequence $\{\boldsymbol{x}_n : n < \infty\}$ of binary strings "rich in runs" so that $\lim_{n\to\infty} \frac{\boldsymbol{r}(\boldsymbol{x_n})}{|\boldsymbol{x_n}|} = \alpha$, where $\alpha = \frac{3}{1+\sqrt{5}} \approx 0.927$ and $\boldsymbol{r}(\boldsymbol{x}) = number\ of\ runs\ in\ \boldsymbol{x}$. Although any such sequence does not establish a lower bound (not even an asymptotic one), it has been "viewed" as such. The assumption underneath that view is that $\boldsymbol{\rho}(n)$ behaves "reasonably", i.e. that $\boldsymbol{\rho}(n)$ does not exhibit wild jumps up, or equivalently, that $\frac{\boldsymbol{\rho}(n)}{n}$ does not exhibit wild oscillations,

⋆ Supported in part by a research grant from the Natural Sciences and Engineering Research Council of Canada.

which is generally expected to be the case (cf. the second conjecture). However, since the "reasonable behaviour" of $\boldsymbol{\rho}(n)$ is yet to be established, we modify and extend the method from [3] to provide formally a family of true asymptotic lower bounds arbitrarily close to αn by proving

Theorem: *For any $\varepsilon > 0$ there is a positive integer N so that for any $n \geq N$, $\boldsymbol{\rho}(n) \geq (\alpha - \varepsilon)n$.*

2 Basic notation, facts, and methods

A **run** $\mathcal{R}$ in a string $\boldsymbol{x}$ is a four-tuple of positive integers (s, p, e, t), where

1. s is the **starting position** of $\mathcal{R}$.
2. p is the **size of its period**.
3. $e \geq 2$ is its **exponent**, i.e. the maximal value e so that $\boldsymbol{x}[s..s{+}p{-}1] = \boldsymbol{x}[s{+}p..s{+}2p{-}1] = \cdots = \boldsymbol{x}[s{+}(e{-}1)p..s{+}ep{-}1]$.
4. The **period** of $\mathcal{R}$, $\boldsymbol{x}[s..s{+}p{-}1]$ itself is not a repetition.
5. The **square part** of the run $\mathcal{R}$, $\boldsymbol{x}[s..s{+}p{-}1] = \boldsymbol{x}[s{+}p..s{+}2p{-}1]$ is **left-maximal**, i.e. $\boldsymbol{x}[s{-}1..s{+}p{-}2] \neq \boldsymbol{x}[s{+}p{-}1..s{+}2p{-}2]$.
6. t is the **tail** of $\mathcal{R}$ and indicates how far to the right the run can be extended, i.e. t is a maximal number so that for any $0 < t' \leq t$, $\boldsymbol{x}[s{+}t'..s{+}t'{+}p{-}1] = \boldsymbol{x}[s{+}t'{+}p..s{+}t'{+}2p{-}1] = \cdots = \boldsymbol{x}[s{+}t'{+}(e{-}1)p..s{+}t'{+}ep{-}1]$.

Not too much is known about the behaviour of the maxrun function:

- *For any n, $\boldsymbol{\rho}(n{+}2) \geq \boldsymbol{\rho}(n){+}1$.*
  Take a string $\boldsymbol{x}$ of length n with $\boldsymbol{r}(\boldsymbol{x}) = \boldsymbol{\rho}(n)$. Take a letter c that does not occur in $\boldsymbol{x}$. Then $\boldsymbol{x}$cc is a string of length $n{+}2$ and $\boldsymbol{\rho}(n{+}2) \geq \boldsymbol{r}(\boldsymbol{x}cc) = \boldsymbol{r}(\boldsymbol{x}){+}1 = \boldsymbol{\rho}(n){+}1$.
- *For any n, $\boldsymbol{\rho}(n{+}1) \leq \boldsymbol{\rho}(n){+}\lfloor \frac{n}{2} \rfloor$.*
  Take a string $\boldsymbol{x}$ of length $n{+}1$ with $\boldsymbol{r}(\boldsymbol{x}) = \boldsymbol{\rho}(n{+}1)$. There can be at most $\lfloor \frac{n}{2} \rfloor$ squares starting at position 1. Then $\boldsymbol{\rho}(n) \geq \boldsymbol{r}(\boldsymbol{x}[2..n{+}1]) \geq \boldsymbol{r}(\boldsymbol{x}){-}\lfloor \frac{n}{2} \rfloor \geq \boldsymbol{\rho}(n{+}1){-}\lfloor \frac{n}{2} \rfloor$.
- *For some n, $\boldsymbol{\rho}(n{+}1) = \boldsymbol{\rho}(n)$.*
  Established by computations, it is not clear if this as an asymptotic property (for instance, $\boldsymbol{\rho}(33) = 27$ while $\boldsymbol{\rho}(34) = 27$).
- *For some n, $\boldsymbol{\rho}(n{+}1) = \boldsymbol{\rho}(n){+}2$.*
  Established by computations, it is not clear if this as an asymptotic property (for instance, $\boldsymbol{\rho}(13) = 8$ while $\boldsymbol{\rho}(14) = 10$).

Note that the function $\frac{\boldsymbol{\rho}(n)}{n}$ may thus not be monotonic. It is not even clear whether $\lim_{n\to\infty} \frac{\boldsymbol{\rho}(n)}{n}$ exists, as $\frac{\boldsymbol{\rho}(n)}{n}$ may be oscillating with a non-decreasing magnitude.

In [3] a special concatenation operator $\circ$ for binary strings was introduced:

$$\boldsymbol{x[1..n]} \circ \boldsymbol{y[1..m]} = \begin{cases} \boldsymbol{x}[1..n]\boldsymbol{y}[2..m] = \boldsymbol{x}[1..n{-}1]\boldsymbol{y}[1..m] & \text{if } \boldsymbol{x}[n] = \boldsymbol{y}[1], \\ \boldsymbol{x}[1..n{-}1]\boldsymbol{y}[2..m] & \text{if } \boldsymbol{x}[n] \neq \boldsymbol{y}[1]. \end{cases}$$

Morphism g was defined by

$$g(\boldsymbol{x}) = \begin{cases} 010010 & \text{if } \boldsymbol{x} = 0 \\ 101101 & \text{if } \boldsymbol{x} = 1 \\ g(\boldsymbol{x}[1..n]) = g(\boldsymbol{x}[1]) \circ g(\boldsymbol{x}[2]) \circ \cdots \circ g(\boldsymbol{x}[n]) & \text{if } |\boldsymbol{x}| > 1. \end{cases} \tag{1}$$

The strings 010010 and 101101 were selected as they provide in the concatenation one extra run:
$\boldsymbol{r}(g(0) \circ g(0)) = 6 = 2\boldsymbol{r}(g(0)){+}2$, the same for $g(1) \circ g(1)$, $\boldsymbol{r}(g(0) \circ g(1)) = 5 = \boldsymbol{r}(g(0)){+}\boldsymbol{r}(g(1)){+}1$, the same for $\boldsymbol{r}(g(1) \circ g(0))$. Let us remark that a computer search carried to the length of 20 did not discover any better pair of strings with such properties.

An important aspect of the morphism is that it "preserves" the runs in $\boldsymbol{x}$: it is a bit tedious to prove and thus not included in the paper, but any left-maximal square in $\boldsymbol{x}$ induces a square in $g(\boldsymbol{x})$. It follows that every run in $\boldsymbol{x}$ induces a run in $g(\boldsymbol{x})$. It is also important to show that two distinct runs in $\boldsymbol{x}$ do not get "glued" together by g.

Let us fix a string $\boldsymbol{x}$. Let $\lambda(\boldsymbol{x})$ denote the number of pairs 00 or 11 in $\boldsymbol{x}$. We can calculate the length of $g(\boldsymbol{x})$:

$$|g(\boldsymbol{x})| = 6|\boldsymbol{x}|{-}\lambda(\boldsymbol{x}){-}2(|\boldsymbol{x}|{-}\lambda(\boldsymbol{x}){-}1) = 4|\boldsymbol{x}|{+}\lambda(\boldsymbol{x}){+}2 \tag{2}$$

the number of pairs 00 or 11 in $g(\boldsymbol{x})$:

$$\lambda(g(\boldsymbol{x})) = |\boldsymbol{x}| \tag{3}$$

the number of runs in $g(\boldsymbol{x})$:

$$\boldsymbol{r}(g(\boldsymbol{x})) = \boldsymbol{r}(\boldsymbol{x}){+}2|\boldsymbol{x}|{+}(|\boldsymbol{x}|{-}1) = \boldsymbol{r}(\boldsymbol{x}){+}3|\boldsymbol{x}|{-}1 \tag{4}$$

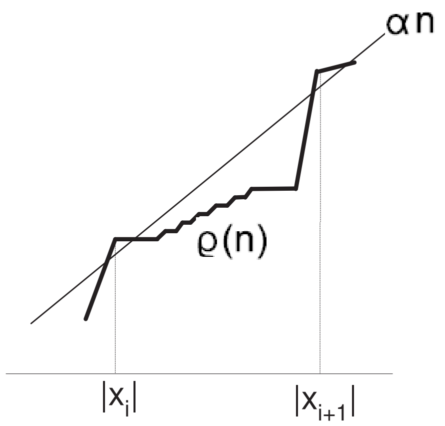


Figure 1. $\boldsymbol{\rho}(n)$ function between $|\boldsymbol{x}_i|$ and $|\boldsymbol{x}_{i+1}|$

In [3] a sequence of strings was generated inductively from a starting string, for instance: $\boldsymbol{x}_0 = 0$, $\boldsymbol{x}_1 = g(0) = 010010$, and $\boldsymbol{x}_{i+1} = g(\boldsymbol{x}_i)$ for $i \geq 1$. Then $|\boldsymbol{x}_{i+1}| = 4|\boldsymbol{x}_i|{+}|\boldsymbol{x}_{i-1}|{+}2$ according (2) and $\boldsymbol{r}(\boldsymbol{x}_{i+1}) = \boldsymbol{r}(\boldsymbol{x}_i){+}3|\boldsymbol{x}_i|{-}1$ according to (4). It is not hard to show that the limit $\lim_{i\to\infty} \frac{|\boldsymbol{x}_i|}{|\boldsymbol{x}_{i+1}|}$ exists and $\beta = \lim_{i\to\infty} \frac{|\boldsymbol{x}_i|}{|\boldsymbol{x}_{i+1}|} = -2{+}\sqrt{5}$. The limit $\lim_{i\to\infty} \frac{\boldsymbol{r}(\boldsymbol{x}_i)}{|\boldsymbol{x}_i|}$ also exists and $\alpha = \lim_{i\to\infty} \frac{\boldsymbol{r}(\boldsymbol{x}_i)}{|\boldsymbol{x}_i|} = \beta(\alpha{+}3)$ giving $\alpha = \frac{3}{1+\sqrt{5}}$.

The sequence $\{|\boldsymbol{x}_i| \;:\; i < \infty\}$ is only "probing" the domain of the function $\boldsymbol{\rho}(n)$ and $\boldsymbol{r}(\boldsymbol{x}_i)$ is "pushing" the value of $\boldsymbol{\rho}(n)$ above αn in these "probing" points (see Figure 1). Since the size of $\boldsymbol{x}_{i+1}$ is more than 4 times the size of $\boldsymbol{x}_i$, the gaps between $|\boldsymbol{x}_i|$ and $|\boldsymbol{x}_{i+1}|$ are getting bigger and bigger.

The basic idea of establishing an asymptotic lower bound for $\boldsymbol{\rho}(n)$ is to start similar sequences from various "starting" strings to cover the domain of $\boldsymbol{\rho}(n)$ densely enough with the "probing" points to get any n close to some "probing" point and hence the value of $\boldsymbol{\rho}(n)$ close to αn. To be able to do so, we must change a bit the way the sequences are generated. The details of this are in the next section.

3 The proof of the theorem

Let $\varepsilon > 0$ be given. We have to find N so that for any $n \geq N$, $\boldsymbol{\rho}(n) \geq (\alpha{-}\varepsilon)n$.

First we will chose and fix three parameters k, δ, and R that we will use throughout the proof. These parameters depend on the given ε: choose and fix a positive integer k so that $\frac{\alpha}{k+1} < \varepsilon$; choose and fix a positive real δ so that $\delta \leq \frac{k+1}{k}(\varepsilon - \frac{\alpha}{k+1})$. It follows that $\frac{k}{k+1}(\alpha{-}\delta) \geq \alpha{-}\varepsilon$. Let R be the smallest integer so that $\left(\frac{k+1}{k}\right)^R \geq 5$.

Consider an increasing sequence $\mathcal{S}_{a,b}$ of positive integers with two integer parameters a and b defined by $n_0(a,b) = a$, $n_1(a,b) = 4a{+}b$, and $n_{i+2}(a,b) = 4n_{i+1}(a,b){+}n_i(a,b)$ for $i \geq 0$. It is not hard to show that $lim_{i\to\infty}\frac{n_i(a,b)}{n_{i+1}(a,b)}$ exists and that

$$lim_{i\to\infty}\frac{n_i(a,b)}{n_{i+1}(a,b)} = -2{+}\sqrt{5}$$

Importantly, ranges of such sequences are "tied" together based on the parameters, i.e. for any integer $t \geq 1$ and any i

$$n_i(ta,tb) = tn_i(a,b). \tag{5}$$

For $0 \leq j < R$, set

$$a(j) = 3(k{+}1)^j k^{(R-j)} \text{ and } b(j) = \frac{a(j)}{3} = (k{+}1)^j k^{(R-j)}. \tag{6}$$

It follows that $\frac{k+1}{k}a(j) = a(j{+}1)$, $\frac{k+1}{k}b(j) = b(j{+}1)$, and $b(j) \geq 3$.

Based on the morphism $g(\boldsymbol{x})$ (see (1)) we define a new morphism $h(\boldsymbol{x})$ by removing the last 2 letters from $g(\boldsymbol{x})$:

$$\text{if } g(\boldsymbol{x}) = y[1..n], \text{then } h(\boldsymbol{x}) = \boldsymbol{y}[1..n{-}2] \tag{7}$$

We use the term *string* $\boldsymbol{s}$ *ends with a square* to indicate that $\boldsymbol{s}$ has a left-maximal square as its suffix. We call a string *good* if it ends with at most one square.
Claim: (a) if $\boldsymbol{x}$ is good, then $h(\boldsymbol{x})$ is good
(b) if $\boldsymbol{x}$ ends with 011, then $h(\boldsymbol{x})$ ends with 011
(c) if $\boldsymbol{x}$ is good, then $\boldsymbol{r}(g(\boldsymbol{x})) \geq \boldsymbol{r}(h(\boldsymbol{x})) \geq \boldsymbol{r}(g(\boldsymbol{x})) - 2$.
(*the claim will be proven after completing the proof of the theorem*)

Now we are in the position to define the "probing" sequences.
For any $0 \leq j < R$ we define a sequence of binary strings $\{\boldsymbol{x}_i(j) \ : \ i < \infty\}$ by:

$$\boldsymbol{x}_0(j) = (011)^{b(j)}$$

and for any $i \geq 0$,

$$\boldsymbol{x}_{i+1}(j) = h(\boldsymbol{x}_i(j))$$

where $b(j)$ is defined in (6). From (2) and (4) it follows that for any $i \geq 0$,

$$|\boldsymbol{x}_0(j)| = 3b(j) = a(j),$$
$$|\boldsymbol{x}_1(j)| = 4a(j)+b(j), \text{and}$$
$$|\boldsymbol{x}_{i+2}(j)| = 4|\boldsymbol{x}_{i+1}(j)|+|\boldsymbol{x}_i(j)|.$$

Thus, the sequence $\{|\boldsymbol{x}_i(j)| \ : \ i < \infty\}$ is the $\mathcal{S}_{a(j),b(j)}$ sequence and so $\lim_{i\to\infty} \frac{|\boldsymbol{x}_i(j)|}{|\boldsymbol{x}_{i+1}(j)|} = -2+\sqrt{5}$.

Since our starting string $\boldsymbol{x}_0(j)$ is good as it equals $(011)^{b(j)}$ and $b(j) \geq 3$, according to the *Claim*, every $\boldsymbol{x}_i(j)$ is good and ends with 011, and

$$\boldsymbol{r}(g(\boldsymbol{x}_i(j))) \geq \boldsymbol{r}(\boldsymbol{x}_{i+1}(j)) \geq \boldsymbol{r}(g(\boldsymbol{x}_i(j)))-2$$

and so

$$\lim_{i\to\infty} \frac{\boldsymbol{r}(\boldsymbol{x}_i(j))}{|\boldsymbol{x}_i(j)|} = \alpha.$$

Therefore, for any $0 \leq j < R$ there is a positive integer I_j so that for any $i \geq I_j$,

$$\frac{\boldsymbol{\rho}(|\boldsymbol{x}_i(j)|)}{|\boldsymbol{x}_i(j)|} \geq \frac{\boldsymbol{r}(\boldsymbol{x}_i(j))}{|\boldsymbol{x}_i(j)|} \geq \alpha-\delta.$$

Let $I = \mathbf{max}\{I_j \ : \ 0 \leq j < R\}$. Then for any $i \geq I$ and any $0 \leq j < R$,

$$\frac{\boldsymbol{\rho}(|\boldsymbol{x}_i(j)|)}{|\boldsymbol{x}_i(j)|} \geq \frac{\boldsymbol{r}(\boldsymbol{x}_i(j))}{|\boldsymbol{x}_i(j)|} \geq \alpha-\delta. \tag{8}$$

From (5) and (6) it follows, that for any i and any $0 \leq j < R$,

$$n_i(a(j), b(j)) = (\frac{k+1}{k})n_i(a(j-1), b(j-1)) = \cdots = (\frac{k+1}{k})^j n_i(a(0), b(0)).$$

Set $N = \mathbf{max}\{n_I(a(j), b(j)) \ : \ 0 \leq j < R\}$. This is the N we were searching for. If $n \geq N$, then for some $i \geq I$,

$$n_i(a(0), b(0)) < n \leq n_{i+1}(a(0), b(0)).$$

Then there is $0 \leq j < R-1$ so that

$$(\frac{k+1}{k})^j n_i(a(0), b(0)) < n \leq (\frac{k+1}{k})^{j+1} n_i(a(0), b(0))$$

$\big[$ since $\left(\frac{k+1}{k}\right)^R \geq 5$, then $\left(\frac{k+1}{k}\right)^R n_i(a(0), b(0)) \geq n_{i+1}(a(0), b(0))$ $\big]$.

It follows that

$$n_i(a(j), b(j)) < n \leq \frac{k+1}{k} n_i(a(j), b(j)).$$

Now we can estimate the value of $\frac{\boldsymbol{\rho}(n)}{n}$ using (8):

$$\frac{\boldsymbol{\rho}(n)}{n} \geq \frac{\boldsymbol{\rho}(n_i(a(j), b(j))}{n} \geq \frac{k}{k+1} \frac{\boldsymbol{\rho}(n_i(a(j), b(j))}{n_i(a(j), b(j))} \geq \frac{k}{k+1}(\alpha-\delta) \geq \alpha-\varepsilon.$$

Thus $\boldsymbol{\rho}(n) \geq (\alpha-\varepsilon)n$. $\square$

Proof. (of *Claim*)
Let us assume that $\boldsymbol{x}$ ends with 011. Then $g(\boldsymbol{x})$ ends with 010010110101101, and so $h(\boldsymbol{x})$ ends with 0100101101011. Consider all runs in $g(\boldsymbol{x})$ that may be "destroyed" by removing the last 2 letters from $g(\boldsymbol{x})$:

(a) If $\boldsymbol{x}$ ends with a square, then the square may induce a left-maximal square in $g(\boldsymbol{x})$ and it will be "destroyed". Since $\boldsymbol{x}$ is good, there may be at most 1 such run destroyed.
(b) $g(\boldsymbol{x})$ ends with square 101|101 that will get destroyed.
(c) The run 01011|01011|01 in $g(\boldsymbol{x})$ becomes a left-maximal square suffix in $h(\boldsymbol{x})$.

No other runs in $g(\boldsymbol{x})$ are affected. Hence $h(\boldsymbol{x})$ is good and at most 2 runs in $g(\boldsymbol{x})$ are destroyed. □

4 Conclusion and further research

We showed that the expectation of αn being a lower bound for the maxrun function $\boldsymbol{\rho}(n)$ is valid by proving that there is a whole family of asymptotic lower bounds arbitrarily close to αn. The further research will include trying to push the lower bound higher up to see whether the conjecture $\boldsymbol{\rho}(n) < n$ holds. This will involve finding novel ways of creating strings "rich in runs" as the approach with concatenation $\circ$ seems to give as much as it could.

References

1. M. Crochemore: *An optimal algorithm for computing the repetitions in a word.* Inform. Process. Lett., 5(5) 1981, pp. 297–315.
2. Fan Kangmin and W. F. Smyth: *A new periodicity lemma.* to appear in SIAM J. of Discr. Math.
3. F. Franek, J. Simpson, and W. F. Smyth: *The maximum number of runs in a string*, in Proceedings of 14th Australasian Workshop on Combinatorial Algorithms AWOCA 2003, Seoul National University, Seoul, Korea, July 13-16 2003.
4. R. Kolpakov and G. Kucherov: *On maximal repetitions in words.* J. of Discrete Algorithms, (1) 2000, pp. 159–186.
5. M. G. Main: *Detecting leftmost maximal periodicities.* Discrete Applied Maths., (25) 1989, pp. 145–153.
6. S. J. Puglisi, W. F. Smyth, and A. Turpin: *Some restrictions on periodicity in strings*, in Proceedings of 16th Australasian Workshop on Combinatorial Algorithms AWOCA 2005, University of Ballarat, Victoria, Australia, September 18-21 2005, pp. 263–268.
7. W. Rytter: *The number of runs in a string: Improved analysis of the linear upper bound*, in Proceedings of 23rd Annual Symposium on Theoretical Aspects of Computer Science STACS 2006, Marseille, France, February 23-25 2006, pp. 184–195.

Flipping Letters to Minimize the Support of a String

Giuseppe Lancia, Franca Rinaldi, and Romeo Rizzi

Dipartimento di Matematica e Informatica
Via delle Scienze 206
33100 Udine, Italy
lancia,rinaldi,rizzi@dimi.uniud.it

Abstract. We consider a problem defined on strings and inspired by the way DNA encodes amino-acids as triplets of nucleotides. Given a string s on an alphabet Σ, a word-length k and a budget D, we want to determine the smallest number of distinct k-mers that can be left in s, if we are allowed to replace up to D letters of s. This problem has several parameters, and we discuss its complexity under all sorts of restrictions on the parameters values. We prove that some versions of the problem are polynomial, while the others are NP-hard.

Keywords: De Bruijn graphs, codons, string algorithms, parametrized complexity.

1 Introduction

In the problem studied in this paper, we consider a string s, of length n, over some alphabet Σ. For $1 \leq k \leq n$, we call a string of length k a k-mer. Note that there are altogether $|\Sigma|^k$ possible k-mers, and that s possesses (as substrings) at most $n-k+1$ *distinct* k-mers. Trivially, there are strings exhibiting only one k-mer (e.g., $s = 00000000$), while $s = 1110100011$ is a shortest string exhibiting all possible binary 3-mers (for results about the the problem of building a shortest string which possesses all possible k-mers see [2,6,5]).

Now, suppose we are allowed to replace up to D letters in s, so as to obtain a new string t. What is the smallest possible number of distinct k-mers that can be left in t?

For instance, assume $\Sigma = \{0, 1\}$ and

$$s = 011010011.$$

There are 6 different 3-mers in s, namely, 011, 110 101, 010, 100, 001. If we are allowed to flip up to $D = 2$ letters, we can obtain the new string

$$t = 01\mathbf{0}01001\mathbf{0},$$

which has only 3 distinct 3-mers, i.e., 010, 100 and 001.

The problem of flipping the right bits, so as to destroy the largest possible number of k-mers (i.e., to leave the fewest possible number of them) has the natural appeal for combinatorial mathematicians, in that it is a cute and challenging combinatorial question. We were inspired to study this problem by considering the way genes encode proteins in an organism's genome [4]. We will briefly discuss the situation here, but, before doing so, we remark that the applications of this paper's results to biology are just marginal. Our interest in the problem is purely on its theoretical aspects, and we will not focus on its practical applications.

A *gene* can be seen as a string over the alphabet {A,C,G,T}. The letters {A,C,G,T} are called *nucleotides*. Each substring of 3 consecutive nucleotides is called a *codon*, as it encodes for a particular *amino-acid*. The amino-acids are the basic constituents of *proteins*, and a protein is a chain of amino-acids, determined by the gene's sequence. A gene has 3 possible *(open) reading frames* (ORFs). I.e., depending on where we start to read, we may obtain a particular set of codons. At ORF number i, for $i = 1, 2, 3$, one reads all the codons that start at positions j, where $(j = i) \mod 3$. For instance, the 3 ORFs of the gene TACAGATAAGATACA are as follows:

T A C A G A T A A G A T A C A
ORF_1 ↑ ↑ ↑ ↑ ↑
ORF_2 ↑ ↑ ↑ ↑ ↑
ORF_3 ↑ ↑ ↑ ↑ ↑

giving rise to the following codons: $ORF_1 = \{TAC, AGA, TAA, GAT, ACA\}$, $ORF_2 = \{ACA, GAT, AAG, ATA\}$, $ORF_3 = \{CAG, ATA, AGA, TAC\}$. Altogether, the distinct codons that we see in this gene are

$$\{TAC, AGA, TAA, GAT, ACA, AAG, ATA, CAG\}.$$

The number of distinct amino-acids a protein consists of is related to its complexity. Hence, we can consider a "complex" gene as one which shows the use of a large number of distinct codons (therefore relating the number of codons to the information content of the gene). Since DNA undergoes random mutations during time, the change of some of the nucleotides in the gene may result in a new sequence which displays much fewer codons than it originally did. Hence, we are led to formulate the question of how few different codons can still be present, in the worst case, after a certain number of mutations have taken place.

1.1 Notation and paper organization

Let s be the input string over an alphabet Σ. We denote by $n = |s|$ the length of s, and by $\sigma = |\Sigma|$ the alphabet size. For a given $k \in \mathbf{N}$, we denote by $\mathcal{K} = \Sigma^k$ the set of all k-mers over Σ. Furthermore, for a string x, we define its *support*, denoted by $K(x) \subseteq \mathcal{K}$, as the set of all k-mers which are substrings of s. For a string x, and $1 \leq i \leq j \leq |x|$, we denote by $x[i, \cdots, j]$ the substring of x from position i to position j. If $i = j$, we shorthand $x[i] = x[i, \cdots, i]$ to represent the i-th character of x. Finally, given two strings x and y of the same length, we denote by $d_H(x, y)$ their *Hamming distance*, i.e., the number of positions in which they differ.

The problem studied in this paper can be stated as follows:

> "KMER": given $s \in \Sigma^n$, a length $k \in \mathbf{N}$ for k-mers, and a budget $D \in \mathbf{N}$, find a string $t \in \Sigma^n$ such that $d_H(s, t) \leq D$ and $|K(t)|$ is minimum.

Notice that this problem has four parameters:

1. The string length n;
2. The alphabet size σ;
3. The k-mer length k;
4. The budget D.

	σ FREE k FREE	σ BOUNDED k FREE	σ FREE k BOUNDED	σ BOUNDED k BOUNDED
n FREE D FREE	1 NP-hard	2 NP-hard for $\sigma = 2$	3 NP-hard for $k = 2$	4 POLY $O(n)$
n BOUNDED D FREE	5 IMP.	6 IMP.	7 IMP.	8 IMP.
n FREE D BOUNDED	9 POLY $O(k\,\sigma^D\,n^{D+1})$	10 POLY $O(k\,n^{D+1})$	11 POLY $O(\,\sigma^D\,n^{D+1})$	12 POLY $O(n^{D+1})$
n BOUNDED D BOUNDED	13 IMP.	14 IMP.	15 IMP.	16 $O(1)$

Table 1. Parameterized complexity of "KMER"

In the remainder of the paper we will address the complexity of the problem when one or more of these parameters are limited to take some bounded values. For instance, when reasoning about genes and DNA sequences, it is $\sigma = 4$ and $k = 3$.

In Section 2 we characterize all possible cases for (n, σ, k, D), classifying them as either polynomial or NP-hard. In Section 3 we consider the polynomial cases of the problem "KMER", arising when D is bounded (as shown in Section 3.1) and when σ and k are both bounded (as shown in Section 3.2). In Section 4 we address the NP-hard cases of the problem, which happen when n and D are unbounded and at most one of σ and k is bounded. Finally, we draw some conclusions in Section 5.

2 Parameterized complexity

The problem has four parameters: the string length n, the alphabet size σ, the k-mer size k, and the budget D. We may consider all possibilities in which some of the parameters are bounded by constants (denoted as "BOUNDED" in Table 1), and some depend on the input (denoted as "FREE" in Table 1). All the cases are described in Table 1.

Before analyzing the cases, we make the following remark:

Remark 1. We can always assume
(i) $D \leq n$;
(ii) $k \leq n$;
(iii) $\sigma \leq n$.

Proof. Remarks (i) and (ii) are obvious. As for Remark (iii), we can reason as follows. Let α be any symbol occurring in s. Let t' be obtained from t by replacing with α each symbol which does not occur in s. It is not difficult to see that $d_H(s, t') \leq d_H(s, t)$ and $|K(t')| \leq |K(t)|$. Hence, t' is an optimal solution as well. Therefore, we can restrict Σ to the symbols originally in s and hence $\sigma \leq n$. □

We have the following situation:

– Cells 1, 2, 3: These cases are NP-hard, as shown in the Section 4 (in particular, the problem is NP-hard even for $\sigma = 2$ or for $k = 2$).

- Cell 4: This case is polynomial, as described in Section 3.2. The complexity is $O(2^{\sigma^k}\sigma^{k+1}n) = O(n)$.
- Cells 5, 6, 7, 8: Because of Remark 1(i), these cases are impossible (denoted by "IMP." in the table).
- Cells 9, 10, 11, 12: All these cases are polynomial, as described in Section 3.1. The intuition is that, in these cases, there are only a polynomial number of possible solutions, which can be checked exhaustively.
- Cells 13, 15: Because of Remark 1(iii) these are impossible cases (when n is bounded by a constant, also σ is bounded by a constant).
- Cell 14: This case is impossible by Remark 1(ii).
- Cell 16: This case is trivial. There is only a finite number of possible problem instances.

3 Polynomial cases

3.1 The case for D fixed

Theorem 2. *When D is bounded by a constant, while n is free, the problem* "KMER" *can be solved in polynomial time.*

Proof. Notice that there are $\binom{n}{D} = O(n^D)$ possible choices for the letters of s to flip, and σ^D possible ways to flip each one. Hence, there are only $O(\sigma^D n^D)$ possible solutions, which is $O(n^D)$ when σ is fixed, and $O(n^{2D})$ otherwise (because of Remark 1(iii)). Since there are only a polynomial number of solutions, and each solution value can be clearly computed in polynomial time (see Remark 3 here below), the enumeration of all possible solutions solves "KMER" in polynomial time. □

Remark 3. Given a string t of length n, the number of different k-mers occurring in t can be computed in $O(kn)$ time.

Proof. To count the k-mers, we scan the string t from left to right, and insert each k-mer in a trie T, initially empty. The branches of T are associated to the symbols of Σ and each leaf of T will represent a k-mer of t. A k-mer describes a path in T to a (possibly non-existing) leaf. Each time a new k-mer x is inserted in T, the path is followed in T up to the point p where it is no longer possible. If this happens at a leaf, then x was already present in T. Otherwise, p is an internal node. From p, we create the new branches that will lead to a new leaf (corresponding to x), and we increase a leaf counter (which eventually counts all k-mers). Notice that the insertion in T has cost $O(k)$. □

The complexity of the algorithms for all cases when D is fixed are reported in Table 1, cells 9–12.

3.2 The case for σ and k fixed

In this section we prove that, when k and σ are both bounded, the problem "KMER" is polynomially solvable. In order to do so, we start by introducing the following auxiliary problem, which we call FEAS(A):

> **FEAS(A):** Consider an instance of "KMER" and a given set of k-mers $A \subseteq \mathcal{K}$. Does there exist a string t' such that $d_H(s,t') \leq D$ and $K(t') \subseteq A$?

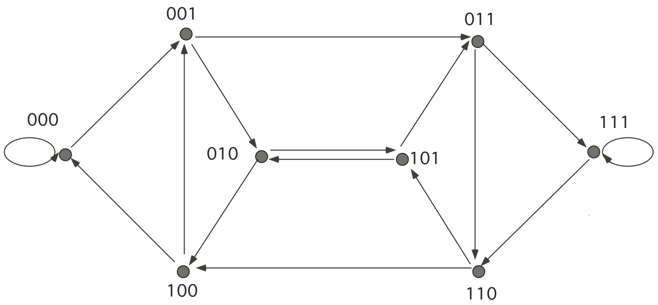


Figure 1. The Shift Register Graph G_3

In the remainder of the section we will prove the following lemma (actually, we obtain a slightly stronger result, i.e., we show how to find t' such that $K(t') \subseteq A$ and $d_H(s, t')$ is minimum):

Lemma 4. *The problem FEAS(A) can be solved in polynomial time.*

We now use Lemma 4 to derive the following theorem.

Theorem 5. *For fixed k and σ, the problem* "KMER" *is polynomially solvable.*

Proof. For fixed k and σ, there are $O(1)$ k-mers, and $O(1)$ possible supports for the solutions (i.e., $O(1)$ possible subsets A of k-mers such that the solution has all its k-mers in A). For each A, enumerated by non-decreasing cardinality, we check in polynomial time (by Lemma 4) if there is a solution t' with support A. We stop as soon as the answer is "yes", and t' is then the optimal solution to "KMER". $\square$

Note that, as $|\mathcal{K}| = \sigma^k$, the above procedure requires to examine, in the worst-case, $O(2^{\sigma^k})$ possible supports. Hence, although polynomial, the algorithm suggested is of little practical use.

We now devote the rest of this section to proving Lemma 4.

The *Shift Register Graph*, also called *De Bruijn Graph*, (SRG, [1]) G_k, for a given k, is a directed graph with node set $\mathcal{K}$, and arcs from i to j whenever $i[2, \cdots, k] = j[1, \cdots, k-1]$ (G_3 is depicted in Figure 1). For $A \subseteq \mathcal{K}$, we denote by $G_k[A]$ the subgraph of G_k induced by the vertices in A. With a slight abuse of notation, we write $(i, j) \in G_k[A]$ to assert that (i, j) is an arc of $G_k[A]$.

We now describe a Dynamic Programming recurrence for the solution of FEAS(A). For $i \in A$ and $h = 1, \ldots, n-k+1$, define $V(i, h)$ to be the minimum Hamming distance between $s[1, \cdots, h+k-1]$ and any string which has all its k-mers in A and ends with k-mer i. We are interested in finding $V(A) := \mathbf{min}_{i \in A} V(i, n-k+1)$. We have the following recurrence, for $1 < h \leq n-k+1$ and $i \in A$,

$$V(i, h) = \min_{i' : (i', i) \in G_k[A]} \left(V(i', h-1) + d_H(i[k], s[h+k-1]) \right). \tag{1}$$

The boundary conditions are that $V(i, 1) = d_H(i, s[1, \cdots, k])$ for all $i \in A$. The complexity of this Dynamic Program is $|A| \times n \times O(\sigma)$, where $O(\sigma)$ is the time of computing the **min** in (1). In fact, given $i \in A$ and a tree T whose leaves are all kmers of A, built as in the proof of Remark 3, each i' such that $(i', i) \in G_k[A]$ can be found as a leaf of T which has the same parent as i. Hence, one only needs to step up one level from i to its parent p, and follow each branch from p to another leaf.

From the above discussion, it follows that the overall time needed to solve "KMER" when σ and k are fixed is $O(2^{\sigma^k} \sigma^{k+1} n)$.

4 NP-hard cases

In this section we show that the problem "KMER" is NP-hard: For completeness, we restate here the problem "KMER":

> **INSTANCE**: An alphabet Σ, an integer k, a string s over Σ, an integer D.
> **PROBLEM**: Find a string $t \in \Sigma^{|s|}$ with $d_H(s,t) \leq D$ and having the smallest possible number of distinct k-mers.

4.1 NP-hardness for fixed *k*

Theorem 6. *The* "KMER" *Problem is NP-hard already for* $k = 2$.

The reduction we propose is from the following problem, called "Compact Bipartite Subgraph":

> **INSTANCE**: A bipartite graph $G = (U, V; E)$, an integer ϕ, an integer λ.
> **PROBLEM**: Find a set of nodes $X \subseteq U \cup V$ with $|X| \leq \phi$ and $|E(G[X])| \geq \lambda$.

Notice that the NP-hardness of the above problem follows trivially from the NP-completeness of "Balanced Complete Bipartite Subgraph". The "Balanced Complete Bipartite Subgraph" problem, also named "GT24" in Garey and Johnson [3], is the following problem.

> **INSTANCE**: A bipartite graph $G = (U, V; E)$, an integer K.
> **QUESTION**: Are there two sets $U' \subseteq U$ and $V' \subseteq V$ with $|U'| = |V'| = K$ and such that $(u, v) \in E(G[U' \cup V'])$ for each $u \in U'$ and $v \in V'$?

And, clearly, the answer to the above question is "yes" if and only if there exists a set of nodes $X \subseteq U \cup V$ such that $|X| \leq 2K$ and $|E(G[X])| \geq K^2$. (For the "only if", notice that if such an X exists then, necessarily, $|X \cap U| = |X \cap V| = K$).

Proof. (Theorem 6) Here we give a reduction from "Compact Bipartite Subgraph" to "KMER". Let A, B be two special symbols and consider $\Sigma = U \cup V \cup \{A, B\}$. Set $D := |E| - \lambda$ and let $M := D + 1$ play the role of a sufficiently big value. Consider the following string $s = s(G, M)$, where the product of two strings denotes their concatenation (and powers are defined accordingly)

$$s = \left(\prod_{v \in V} (Bv)^M\right) B^{2M} \left(\prod_{u \in U} (uB)^M\right) \left(\prod_{(u,v) \in E} (BBuAvBB)\right). \tag{2}$$

A word of explanation is in order to better agree on what the above "simplified" expression for s actually represents: In our intentions, the string s should be considered as uniquely defined. In practice, we refer to implicit orderings of the elements in U, in V, and in E, so that a writing like $\prod_{v \in V}(Bv)$ uniquely defines a string over $V \cup \{B\}$. More precisely, where $v_1, \ldots, v_n$ is an ordering of V, then $\prod_{v \in V}(Bv)$ should be understood as a shorthand for $\prod_{i=1}^{n}(Bv_i)$.

Lemma 7. *There exists a string* $t \in \Sigma^{|s|}$ *with* $d_H(s,t) \leq D$ *and with at most* $1 + 2|U| + 2|V| + \phi$ *distinct* 2*-mers if and only if there exists* $X \subseteq U \cup V$ *with* $|X| \leq \phi$ *and such that* $|E(G[X])| \geq \lambda$.

Proof: Assume given an $X \subseteq U \cup V$ with $|X| \leq \phi$ and such that $|E(G[X])| \geq \lambda$. Consider the following string t, where $S_{(u,v)} := BBuAvBB$ if $(u,v) \in E(G[X])$, and $S_{(u,v)} := BBuBvBB$ if $(u,v) \in E \setminus E(G[X])$:

$$t = \left(\prod_{v \in V} (Bv)^M\right) B^{2M} \left(\prod_{u \in U} (uB)^M\right) \left(\prod_{(u,v) \in E} S_{(u,v)}\right).$$

Notice that $|t| = |s|$ and $d_H(s,t) = |E| - |E(G[X])| \leq |E| - \lambda = D$. Moreover, any 2-mer appearing in t will fall into one of the following categories:

- the single 2-mer BB;
- the $2|U| + 2|V|$ 2-mers of the form zB and Bz for $z \in U \cup V$;
- the $|X \cap U|$ 2-mers of the form uA with $u \in X \cap U$;
- the $|X \cap V|$ 2-mers of the form Av with $v \in X \cap V$.

Hence, the number of distinct 2-mers in t is $1+2|U|+2|V|+|X| \leq 1+2|U|+2|V|+\phi$, as stated.

Conversely, let t be any string such that $|t| = |s|$ and $d_H(s,t) \leq D$. Then the 2-mer BB certainly appears in t since B^{2M}, which is a substring of s, contains M disjoint occurrences of BB. Similarly, for each node $z \in U \cup V$, the 2-mers zB and Bz certainly appear in t. We are assuming that besides these $1 + 2|U| + 2|V|$ 2-mers, string t contains at most ϕ other 2-mers. Let X be made by those $u \in U$ such that the 2-mer uA occurs in t plus the set of those $v \in V$ such that the 2-mer Av occurs in t. Hence, $|X| \leq \phi$. Remember that the string s contains the substring $\prod_{(u,v) \in E}(BBuAvBB)$. Since $d_H(s,t) \leq D$, it follows that $|E \setminus E(G[X])| \leq D$, and hence that $|E(G[X])| \geq |E| - D = \lambda$. ☐

4.2 NP-hardness for fixed $|\Sigma|$

In this subsection we prove the following.

Theorem 8. *The* "KMER" *problem is NP-hard already for* $|\Sigma| = 2$.

A noticeable property of the proposed reduction is that it constructs instances with $k = O(\log n)$, which allows us to infer the following result.

Theorem 9. *No algorithm solves the* "KMER" *problem in* $O(n^{POLY(k)})$ *time unless* $NP \subseteq DTIME(n^{POLY(\log n)})$. *This negative result holds also for* $|\Sigma| = 2$.

Theorem 9 gives strong evidence that there is no space for improving the $O(n^{|\Sigma|^k})$ approach of Section 3.2. Indeed, an $O(2^{|\Sigma|^k} n^{POLY(|\Sigma|,k)})$ algorithm would imply an $O(n^{POLY(k)})$ algorithm when $|\Sigma| = 2$, and $NP \subseteq DTIME(n^{POLY(\log n)})$ would follow.

The general approach of the reduction is the same as described in Subsection 4.1 In particular, the reduction will be again from "Compact Bipartite Subgraph".

The reduction. As in Subsection 4.1, let $D := |E| - \lambda$ and let $M := D + 1$ play the role of a sufficiently big value. We now work with $\Sigma = \{0, 1\}$. Before explaining how to construct s, t and k, we point out that in the construction given in Subsection 4.1 the fact that certain k-mers had to be present in every string t with $d_H(s,t) \leq D$ played a key role. We hence start the derivation of the reduction to be given here with the consideration that it is relatively easy to build a string s_0 which contains M

disjoint copies of each substring s' in $\{0,1\}^k$ such that either $s'[1] = 1$ or $s'[k] = 1$. Indeed, $s_0 := \left(\prod_{\sigma \in \{0,1\}^{k-1}} 1^k \sigma 1^k\right)^M$ will do the job. Notice also that s_0 contains no k-mer which both starts and ends with 0. Let $h = \lceil \log_2 |U \cup V| \rceil$. Then, to each node $z \in U \cup V$ we can associate a unique binary string $f(z) \in \{0,1\}^h$, called *the short encoding of* z. To each node $z \in U \cup V$ we also associate *the long encoding of* z, denoted by $f'(z) \in \{0,1\}^{2h+1}$, defined as $f'(z)[1] = 0$, $f'(z)[2i] = f(z)[i]$ and $f'(z)[2i+1] = 1$ for each $i = 1, 2, \ldots, h$. When s is a string we denote by $[s]^R$ the reverse of string s. Take $k = 2h + 2$. Consider the following string $s = s(G, M)$

$$s = s_0 \prod_{(u,v) \in E} (1^k \, 0 \, f'(u) \, 0 \, [f'(v)]^R \, 0 \, 1^k).$$

Lemma 10. *There exists a string $t \in \Sigma^{|s|}$ with $d_H(s,t) \leq D$ and with at most $3 \cdot 2^{k-2} + \phi$ distinct k-mers if and only if there exists $X \subseteq U \cup V$ with $|X| \leq \phi$ and such that $|E(G[X])| \geq \lambda$.*

Proof: The number of strings $s \in \{0,1\}^k$ is 2^k and for 2^{k-2} of them we have that $s[1] = s[k] = 0$. Hence, there are precisely $2^k - 2^{k-2} = 3 \cdot 2^{k-2}$ strings $s \in \{0,1\}^k$ with $s[1] = 1$ or $s[k] = 1$, which account for the numbers occurring in the statement. Assume to be given an $X \subseteq U \cup V$ with $|X| \leq \phi$ and such that $|E(G[X])| \geq \lambda$. Consider the following string t,

$$t = s_0 \prod_{(u,v) \in E} S_{(u,v)} \,,$$

where

$$S(u,v) := \begin{cases} 1^k 0 f'(u) 0 [f'(v)]^R \, 01^k & \text{if } (u,v) \in E(G[X]) \\ 1^k 0 f'(u) 1 [f'(v)]^R 01^k & \text{if } (u,v) \in E \setminus E(G[X]) \end{cases}$$

Notice that $|t| = |s|$ and $d_H(s,t) = |E| - |E(G[X])| \leq |E| - \lambda = D$. Moreover, any k-mer appearing in t falls into one of the following categories:

- the $3 \cdot 2^{k-2}$ k-mers starting or ending with a 1 symbol;
- the $|X \cap U|$ k-mers of the form $f'(u)0$ with $u \in X \cap U$;
- the $|X \cap V|$ k-mers of the form $0[f'(v)]^R$ with $v \in X \cap V$.

Hence, the number of distinct k-mers in t is at most $3 \cdot 2^{k-2} + \phi$, as stated.

Conversely, let t be any string such that $|t| = |s|$ and $d_H(s,t) \leq D$. Then all the $3 \cdot 2^{k-2}$ k-mers starting or ending with a 1 symbol do certainly appear in t since s contains at least $M > D$ disjoint occurrences of each of them.

Assume that besides these $3 \cdot 2^{k-2}$ k-mers, string t contains at most ϕ other k-mers. Let X be made by those $u \in U$ such that the k-mer $f'(u)0$ occurs in t plus the set of those $v \in V$ such that the k-mer $0[f'(v)]^R$ occurs in t. Hence, $|X| \leq \phi$. Remember that the string s contains the substring $\prod_{(u,v) \in E} (1^k 0 f'(u) 0 [f'(v)]^R 01^k)$. Since $d_H(s,t) \leq D$, it follows that $|E \setminus E(G[X])| \leq D$, and hence that $|E(G[X])| \geq |E| - D = \lambda$. □

5 Conclusions

In this paper we have characterized the complexity of the "KMER" problem, under all possible cases for its parameters. For the solution of the NP-hard cases, we have devised Integer Linear Programming formulations, which, for space reasons, are not included in this extended abstract. From our first experimental results, the ILP approach seems suitable for the solution of moderate-size instances of this problem, while for larger-size instances a possibly different (maybe combinatorial) approach should be sought.

References

1. N. D. de Bruijn: *A combinatorial problem.* Koninklijke Netherlands: Academe Van Wetenschappen, 49 1946, pp. 758–764.
2. A. Flaxman, A. W. Harrow, and G. B. Sorkin: *Strings with maximally many distinct subsequences and substrings.* The Electronic J. of Combinatorics, 11 2004.
3. M. R. Garey and D. S. Johnson: *Computers and Intractability, a Guide to the Theory of NP-Completeness*, W.H. Freeman and Co, 1979.
4. J. D. Watson, M. Gilman, J. Witkowski, and M. Zoller: *Recombinant DNA*, Scientific American Books, W. H. Freeman and Co., 1992.
5. D. B. West: *Introduction to Graph Theory*, Prentice Hall, 1996.
6. H. S. Wilf: *Combinatorial Algorithms: An Update*, SIAM CBMS-NSF Regional Conference Series in Applied Mathematics, Society for Industrial and Applied Mathematics, 1989.

Two-Dimensional Bitwise Memory Matrix: A Tool for Optimal Parallel Approximate Pattern Matching*

Jan Šupol and Bořivoj Melichar

Department of Computer Science & Engineering
Faculty of Electrical Engineering
Czech Technical University in Prague
Karlovo nám. 13, 121 35 Prague 2
jan.supol@gmail.com, melichar@fel.cvut.cz

Abstract. A very fast parallel approach to pattern matching is presented. The approach is based on the bit-parallel approach and we use two-dimensional bitwise memory matrix which helps to achieve very fast parallel pattern matching algorithms. The parallel pattern matching takes $\mathcal{O}(1)$ time for the exact pattern matching and $\mathcal{O}(k)$ for the approximate pattern matching, where k is the number of errors.

1 Introduction

The pattern matching problem is to find all occurrences of a given pattern $P = p_1p_2 \dots p_m$ in a larger text $T = t_1t_2 \dots t_n$, $n > m$, both sequences of symbols from a given alphabet $A = a_1a_2 \dots a_{|A|}$.

Many different solutions of this problems are known, and we are interested in the pattern matching using finite nondeterministic automata. A finite automaton (*FA*) is a quintuple (Q, A, δ, I, F) where Q is a finite set of states, A is a finite input alphabet, and $F \subseteq Q$ is a set of final states. If *FA* is nondeterministic (*NFA*), then δ is a mapping $Q \times (A \cup \{\varepsilon\}) \mapsto P(Q)$ and $I \subseteq Q$ is a set of initial states. If *FA*$= (Q, A, \delta, q_0, F)$ is deterministic (*DFA*), then δ is a (partial) function $Q \times A \mapsto Q$ and q_0 is the only initial state. We refer to *NFA* used for pattern matching as a pattern matching automaton (*PMA*).

Hamming distance $H(x, y) \leq k$ is maximum k substitutions (replace operations) required to transform string x into string y (see [4]). *Levenshtein distance* $L(x, y) \leq k$ is maximum k operations replace, insert, or delete required to transform string x into string y. *PMA* for pattern P using the Hamming distance k is a pattern matching automaton that matches any pattern X, such that $H(P, X) \leq k$. *PMA* for pattern P using the Levenshtein distance k is a pattern matching automaton that matches any pattern X, such that $L(P, X) \leq k$.

The running of *PMA* can be simulated by the bit-parallel algorithms. This technique was introduced in [2] ("shift-and" variation), and it was improved in [1,9] ("shift-or" variation used in this paper). It has been shown [5], that the bit-parallel algorithms simulates *NFA* and we use these algorithms for the parallel pattern matching.

* This research has been partially supported by the Ministry of Education, Youth, and Sport of the Czech Republic under research program MSM6840770014, by the Czech Science Foundation as project No. 201/06/1039, and by the Czech Technical University as project No. CTU0609613

Parallel pattern matching was quite a popular topic. We might cite a constant time string matching algorithm [3], which has the same time complexity as our algorithm for the exact pattern matching, though it does not use the bit-parallelism. We might also cite an $\mathcal{O}(log\ m + k)$ time [8], or an $\mathcal{O}(k)$ time [7] algorithms for the approximate pattern matching. These have similar time complexity to our $\mathcal{O}(k)$ algorithm for approximate pattern matching, but our algorithm needs a smaller number of processors.

Our algorithm needs *EREW PRAM* for the exact string matching and *CREW PRAM* for the approximate string matching. More information for the parallel computation models is e.g. in [6]. We also need a shared memory organized as a matrix of size $\mathcal{O}(n \times n)$ bits, where n is the length of the text. Since bit-parallel algorithms work with a computer word of length m, where m is the length of the pattern, we need to access a whole word in the bit-memory matrix. Here we have a strong condition. We need to access this memory both on rows and on columns. Therefore we use two operations to access the memory matrix. The first is $MEMX[index_x][index_y]$ accessing a word in a column with index $index_x$ starting with the bit on a row with index $index_y$ and the second is $MEMY[index_x][index_y]$ accessing a word on a row with index $index_y$ starting with the bit in a column with index $index_x$. This accessibility is enough to present a cost-optimal parallel approximate pattern matching algorithm.

A parallel algorithm is cost-optimal if its time processor product is equal to the time of the best known sequential algorithm solving the same issue.

We use some bitwise operations in this paper. Operation **or** is a standard bitwise OR operation and operation **and** is a standard bitwise AND operation. Operation **shl** is a standard shift-left bitwise operation, and the right-most bit is set to 0. Operation **shr** is a standard shift-right bitwise operation, and the left-most bit is set to 1. We also use operation $\mathbf{shl}^i(x)$ as the operation **shl** performed i times on bit-vector x.

This paper is organized as follows. Section 2 explains the “shift-or” variation of a bit-parallel algorithm. Section 3 discuss the parallel variation of thr “shift-or” algorithm. Section 4 provides a conclusion.

2 Bit-Parallelism

Here we explain the “shift-or” variation of the bit-parallel algorithm. It uses matrices $R^l, 0 \le l \le k$ of size $m \times (n+1)$, and matrix D of size $m \times |A|$, where k is the maximum number of edit operations in pattern P. Each element $r^l_{j,i}, 0 \le i \le n$ contains 0, if the edit distance between string $p_1 p_2 \dots p_j$ and the string ending at position i in text T is $\le l$, or 1, otherwise. Each element $d_{j,x}, 0 < j \le m$, $x \in A$, contains 0, if $p_j = x$, or 1, otherwise.

In exact string matching, vectors $R^0_i, 0 \le i \le n$, are computed as follows:

$$\begin{array}{lll} r^0_{j,0} &= 1, & 0 < j \le m \\ R^0_i &= \mathbf{shl}(R^0_{i-1})\ \mathbf{or}\ D[t_i], & 0 < i \le n \end{array} \tag{1}$$

In approximate string matching using the Hamming distance, vectors $R^l_i, 0 \le l \le k, 0 \le i \le n$, are computed as follows:

$$\begin{array}{lll} r^l_{j,0} &= 1, & 0 < j \le m, 0 \le l \le k \\ R^0_i &= \mathbf{shl}(R^0_{i-1})\ \mathbf{or}\ D[t_i], & 0 < i \le n \\ R^l_i &= (\mathbf{shl}(R^0_{i-1})\ \mathbf{or}\ D[t_i])\ \mathbf{and}\ \mathbf{shl}(R^{l-1}_{i-1}), & 0 < i \le n, 0 < l \le k \end{array} \tag{2}$$

In approximate pattern matching using the Levenshtein distance, vectors $R_i^l, 0 \leq l \leq k, 0 \leq i \leq n$, are computed as follows:

$$\begin{array}{lll} r_{j,0}^l = 0, & & 0 < j \leq l, 0 < l \leq k \\ r_{j,0}^l = 1, & & l < j \leq m, 0 \leq l \leq k \\ R_i^0 = \mathbf{shl}(R_{i-1}^0) \ \mathbf{or} \ D[t_i], & & 0 < i \leq n \\ R_i^l = (\mathbf{shl}(R_{i-1}^l) \ \mathbf{or} \ D[t_i]) & & \\ \quad \mathbf{and} \ \mathbf{shl}(R_{i-1}^{l-1} \ \mathbf{and} \ R_i^{l-1}) & & \\ \quad \mathbf{and} \ (R_{i-1}^{l-1} \ \mathbf{or} \ V), & & 0 < i \leq n, 0 < l \leq k \end{array} \tag{3}$$

The auxiliary vector V is computed as follows:

$$V = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_m \end{bmatrix}, \text{ where } v_m = 1 \text{ and } v_j = 0, \forall j, 1 \leq j < m. \tag{4}$$

The term $\mathbf{shl}(R_{i-1}^l)$ **or** $D[t_i]$) represents matching – position i in text T is increased, the position in pattern P is increased by operation **shl**, and the positions corresponding to the input symbol t_i are selected by term **or** $D[t_i]$. The term $\mathbf{shl}(R_{i-1}^{l-1})$ represents edit operation replace – position i in text T is increased, the position in pattern P is increased, and edit distance l is increased. The term $\mathbf{shl}(R_i^{l-1})$ represents edit operation delete – the position in pattern is increased, the position in the text is not increased, and edit distance l is increased. The term R_{i-1}^{l-1} represents edit operation insert – the position in the pattern is not increased, the position in the text is increased, and edit distance l is increased. The term **or** V provides that no insert transition leads from any final state.

D	a	b	c	d	$A \setminus \{a,b,c,d\}$
a	0	1	1	1	1
d	1	1	1	0	1
b	1	0	1	1	1
b	1	0	1	1	1
c	1	1	0	1	1
a	0	1	1	1	1

Table 1. Matrix D for the pattern $P = adbbca$

An example of mask matrix D for the pattern $P = adbbca$ is shown in Table 1 and an example of matrix R^0 for exact pattern matching and matrix R^1 for approximate string matching using the Levenshtein distance $k = 2$, respectively, is shown in Table 2.

3 Parallelization of the bit-parallel algorithms

This section focuses on bit-parallel simulation of the nondeterministic *PMA*. The motivation is to use many processors, each with computer words long enough to fit into a single register bit-vector m bits in length, and to make the bit-parallel algorithms truly parallel.

R^0	–	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a
a	1	0	1	1	0	1	1	0	0	1	0	1	1	1	1	0
d	1	1	0	1	1	1	1	1	1	1	1	0	1	1	1	1
b	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1
b	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1
c	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1
a	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0
R^1	–	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a
a	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
d	1	0	0	0	0	0	1	0	0	0	0	0	0	1	1	0
b	1	1	0	0	1	0	1	1	1	0	1	0	0	0	1	1
b	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	1
c	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0
a	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0
R^2	–	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a
a	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
d	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
b	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
b	1	1	0	0	0	0	0	1	1	0	0	0	0	0	0	0
c	1	1	1	0	1	1	0	1	1	1	1	1	0	0	0	0
a	1	1	1	1	0	1	1	0	1	1	1	1	1	0	0	0

Table 2. Matrices R^0, R^1, and R^2 for pattern matching using the Levenshtein distance, $T = adcabcaabadbbca$, $k = 2$, $P = adbbca$

Now we explain the idea originating in Formula (1) using the "shift-or" algorithm. The first bit-vectors may be computed as follows:

$$
\begin{aligned}
R_0^0 &= R_0^0 \\
R_1^0 &= \mathbf{shl}(R_0^0) \ \mathbf{or} \ D[t_1] \\
R_2^0 &= \mathbf{shl}(R_1^0) \ \mathbf{or} \ D[t_2] = \mathbf{shl}(\mathbf{shl}(R_0^0) \ \mathbf{or} \ D[t_1]) \ \mathbf{or} \ D[t_2] = \\
&= \mathbf{shl}^2(R_0^0) \ \mathbf{or} \ \mathbf{shl}(D[t_1]) \ \mathbf{or} \ D[t_2] \\
R_3^0 &= \mathbf{shl}(R_2^0) \ \mathbf{or} \ D[t_3] = \mathbf{shl}(\mathbf{shl}(R_1^0) \ \mathbf{or} \ D[t_2]) \ \mathbf{or} \ D[t_3] = \\
&= \mathbf{shl}(\mathbf{shl}(\mathbf{shl}(R_0^0) \ \mathbf{or} \ D[t_1]) \ \mathbf{or}(D[t_2])) \ \mathbf{or} \ D[t_3] = \\
&= \mathbf{shl}^3(R_0^0) \ \mathbf{or} \ \mathbf{shl}^2(D[t_1]) \ \mathbf{or} \ \mathbf{shl}(D[t_2]) \ \mathbf{or} \ D[t_3]
\end{aligned}
$$

Hence the following equation might be proven:

$$R_i^0 = \mathbf{shl}^i(R_0^0) \ \mathbf{or} \ \mathbf{shl}^{i-1}(D[t_1]) \ \ldots \ \mathbf{or} \ \mathbf{shl}(D[t_{i-1}]) \ \mathbf{or} \ D[t_i], \ 1 \leq i \leq n \tag{5}$$

Using Formula (5) we may compute the example for the pattern $P = adbbca$ and text $T = adcbcadbbca$ depicted in Table 3. Note that we use the same matrix D as in Section 2, Table 1.

The initial bit-vector, left-shifted by $i = n = 11$ bits, is in the first column labeled with the symbol "-". In each other column, except the last one, there is a bit-vector from matrix D left-shifted by some number of bits. The **or** operation between all these rows gives the result, and as we can see, a match occurs only in one position, the same as in Table 2. We may conclude many interesting observations from this example.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
R^0	$-$	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a	OR
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**0**	1
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	**1**	1
2	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	**1**	**1**	1
3	0	0	0	0	0	0	0	0	0	0	0	0	**1**	**1**	**1**	**1**	1
4	0	0	0	0	0	0	0	0	0	0	0	**1**	**1**	**0**	**1**	**1**	1
5	0	0	0	0	0	0	0	0	0	0	**0**	**0**	**0**	**0**	**0**	**0**	0
6	0	0	0	0	0	0	0	0	0	**1**	**1**	**1**	**0**	**1**	**1**	1	1
7	0	0	0	0	0	0	0	0	**0**	**1**	**1**	**1**	**1**	**1**	1	1	1
8	0	0	0	0	0	0	0	**0**	**1**	**0**	**1**	**1**	**1**	1	1	1	1
9	0	0	0	0	0	0	**1**	**1**	**1**	**0**	**1**	**1**	1	1	1	1	1
10	0	0	0	0	0	**1**	**1**	**1**	**1**	**1**	**0**	1	1	1	1	1	1
11	0	0	0	0	**0**	**1**	**1**	**1**	**1**	**1**	1	1	1	1	1	1	1
12	0	0	0	**1**	**1**	**0**	**1**	**1**	**0**	1	1	1	1	1	1	1	1
13	0	0	**1**	**1**	**1**	**0**	**0**	**0**	1	1	1	1	1	1	1	1	1
14	0	**0**	**0**	**1**	**1**	**1**	**1**	1	1	1	1	1	1	1	1	1	1
15	**1**	**1**	**1**	**1**	**1**	**1**	1	1	1	1	1	1	1	1	1	1	1
16	**1**	**1**	**1**	**0**	**0**	1	1	1	1	1	1	1	1	1	1	1	1
17	**1**	**1**	**1**	**1**	1	1	1	1	1	1	1	1	1	1	1	1	1
18	**1**	**1**	**1**	1	1	1	1	1	1	1	1	1	1	1	1	1	1
19	**1**	**0**	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
20	**1**	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Table 3. Matrix R^0 for pattern matching for the exact pattern, $P = adbbca$, $T = adcabcaabadbbca$

There are only two reasons for the initial bit-vector R^0_0 to have something to start sequentially with, and to disallow a match on the prefix of the text T shorter than pattern P.

The only interesting part of the matrix is the bold part. We can also exclude the first $m - 1 = 5$ rows labeled with symbols "a, d, b, b, c" and the last $m = 6$ rows, since there cannot be a match (in the exact case). Also the initial bit-vector is not important.

We implement the shift by j-bits $\mathbf{shl}^j(D[t_i])$ operation as a writing of one computer word m-bits long into a memory in a specified position using operation $MEMX[n-j][j] \leftarrow D[t_i]$. Having this memory organization, there is a word on each row of the memory $MEMY[n-j][j]$ (a bold one in Table 3), which contains a crucial information:

Proposition 1. *The word in memory* $MEMY[n-j+1][j-1] > 0, 1 \leq j \leq n$ *if and only if* $t_{n-j+1}t_{n-j+2}\ldots t_{n-j+m} \neq p_1p_2\ldots p_m$.

Proof. Simply from the definition of matrix D. If $t_{n-j+i} = p_i$ then $d_{i,n-j+i} = 0$ or 1 otherwise, $1 \leq j \leq n, 1 \leq i \leq m$. But each bit $d_{i,n-j+i}$ is in the $(j-1)$-th row, because vector $D[t_{n-j+i}]$ has been shifted by $j - i$ bits. □

The **or** operation is then a comparison, whether a computer word in a row is zero (a match), or non-zero (a mismatch) and we may compute:

$$R^0_i = MEMY[n-i+1][i-1], \quad m \leq i \leq n \tag{6}$$

We are not interested in the bit-vectors $R_i^0, 0 < i < m-1$, because there can not be a match.

If the access time to the memory using both *MEMX* and *MEMY* operations is $\mathcal{O}(1)$, string-matching in parallel takes only $\mathcal{O}(1)$ parallel time using n processors.

3.1 Parallel pattern matching using the Hamming distance

The observation of Table 3 may continue. Matrix R^l no longer consists only of bit-vectors $R_i^l, 0 \leq l \leq k, 0 \leq i \leq n$.

The bit-vectors R_i^0 computed by Formula 1 in the sequential algorithm on the diagonal of the matrix were composed of the bit-vectors $n-i+1, n-i+2, \ldots, n-i+m$. However, they are slightly different: the bit-vectors computed in parallel contain more bits set to zeros, more active states, because they were not deactivated using prefix computation by the operation **or**. In exact pattern matching this was not important, because Proposition 1 ensures at least one bit set to one if there is no match.

The advantage is that the number of bits set to 1 indicates the number of replace operations needed for a matching.

However, using the Hamming distance this difference is most important. Since we are going to perform "replace" operations in parallel, we could perform more than one "replace", each in a different position and we cannot guarantee the number of these "replace" operations. Therefore we need vectors R_j^0 exactly as in the sequential version. This ensures only one "replace" operation performed in each of k levels, because the "replace" is only after a "match", except the first symbol.

C^0	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a	OR
5	0	0	0	0	0	0	0	0	0	**0**	**0**	**0**	**0**	**0**	**0**	0
6	0	0	0	0	0	0	0	0	**1**	**1**	**1**	**1**	**1**	**1**	1	1
7	0	0	0	0	0	0	0	**0**	**1**	**1**	**1**	**1**	**1**	1	1	1
8	0	0	0	0	0	0	**0**	**1**	**1**	**1**	**1**	**1**	1	1	1	1
9	0	0	0	0	0	**1**	**1**	**1**	**1**	**1**	**1**	1	1	1	1	1
10	0	0	0	0	**1**	**1**	**1**	**1**	**1**	**1**	1	1	1	1	1	1
11	0	0	0	**0**	**1**	**1**	**1**	**1**	**1**	1	1	1	1	1	1	1
12	0	0	**1**	**1**	**1**	**1**	**1**	**1**	1	1	1	1	1	1	1	1
13	0	**1**	**1**	**1**	**1**	**1**	**1**	1	1	1	1	1	1	1	1	1
14	**0**	**0**	**1**	**1**	**1**	**1**	1	1	1	1	1	1	1	1	1	1
R^1	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a	OR
5	0	0	0	0	0	0	0	0	0	**0**	**0**	**0**	**0**	**0**	**0**	0
6	0	0	0	0	0	0	0	0	**0**	**1**	**1**	**0**	**1**	**1**	1	1
7	0	0	0	0	0	0	0	**0**	**0**	**1**	**1**	**1**	**1**	1	1	1
8	0	0	0	0	0	0	**0**	**0**	**0**	**1**	**1**	**1**	1	1	1	1
9	0	0	0	0	0	**0**	**1**	**1**	**0**	**1**	**1**	1	1	1	1	1
10	0	0	0	0	**0**	**1**	**1**	**1**	**1**	**0**	1	1	1	1	1	1
11	0	0	0	**0**	**0**	**1**	**1**	**1**	**1**	1	1	1	1	1	1	1
12	0	0	**0**	**1**	**0**	**1**	**1**	**0**	1	1	1	1	1	1	1	1
13	0	**0**	**1**	**1**	**0**	**0**	**0**	1	1	1	1	1	1	1	1	1
14	**0**	**0**	**0**	**1**	**1**	**1**	1	1	1	1	1	1	1	1	1	1

Table 4. Matrices C^0, and R^1 for pattern matching using the Hamming distance, $T = adcbcadbbca$, $k = 3$, $P = adbbca$

C^1	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a	OR
5	0	0	0	0	0	0	0	0	0	**0**	**0**	**0**	**0**	**0**	**0**	0
6	0	0	0	0	0	0	0	0	**0**	**1**	**1**	**1**	**1**	**1**	1	1
7	0	0	0	0	0	0	0	**0**	**0**	**1**	**1**	**1**	**1**	1	1	1
8	0	0	0	0	0	0	**0**	**0**	**0**	**1**	**1**	**1**	1	1	1	1
9	0	0	0	0	0	**0**	**1**	**1**	**1**	**1**	**1**	1	1	1	1	1
10	0	0	0	0	**0**	**1**	**1**	**1**	**1**	**1**	1	1	1	1	1	1
11	0	0	0	**0**	**0**	**1**	**1**	**1**	**1**	1	1	1	1	1	1	1
12	0	0	**0**	**1**	**1**	**1**	**1**	**1**	1	1	1	1	1	1	1	1
13	0	**0**	**1**	**1**	**1**	**1**	**1**	1	1	1	1	1	1	1	1	1
14	**0**	**0**	**0**	**1**	**1**	**1**	1	1	1	1	1	1	1	1	1	1

R^2	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a	OR
5	0	0	0	0	0	0	0	0	0	**0**	**0**	**0**	**0**	**0**	**0**	0
6	0	0	0	0	0	0	0	0	**0**	**0**	**1**	**0**	**1**	**1**	1	1
7	0	0	0	0	0	0	0	**0**	**0**	**0**	**1**	**1**	**1**	1	1	1
8	0	0	0	0	0	0	**0**	**0**	**0**	**0**	**1**	**1**	1	1	1	1
9	0	0	0	0	0	**0**	**0**	**1**	**0**	**1**	**1**	1	1	1	1	1
10	0	0	0	0	**0**	**0**	**1**	**1**	**1**	**0**	1	1	1	1	1	1
11	0	0	0	**0**	**0**	**0**	**1**	**1**	**1**	1	1	1	1	1	1	1
12	0	0	**0**	**0**	**0**	**1**	**1**	**0**	1	1	1	1	1	1	1	1
13	0	**0**	**0**	**1**	**0**	**0**	**0**	1	1	1	1	1	1	1	1	1
14	**0**	**0**	**0**	**0**	**1**	**1**	1	1	1	1	1	1	1	1	1	1

Table 5. Matrices C^1, and R^2 for pattern matching using the Hamming distance, $T = adcbcadbbca$, $k = 3$, $P = adbbca$

It is very easy to find the first (highest, left-most) bit set to 1 and to set all lower bits to 1 as well. This operation can be performed as:

$$R_i^0 \leftarrow R_i^0 \ \textbf{or} \ 2^{\lceil log_2 R_i^0 \rceil}, \quad m \leq i \leq n \tag{7}$$

This computation is very fast, $\mathcal{O}(1)$ time. For example the operation $log_2 x$ is computed using instruction $FYL2X$ on X86 processors.

We use the observations on this section and we can formulate the idea of parallel pattern matching using the Hamming distance, which could be used later with the Levenshtein distance.

We compute the corrected matrix $R^l, 0 \leq l \leq k$ from each matrix R^l using Formula 7. We refer to this corrected matrix as C^l. The shifted vectors $D[t_i], 0 \leq i \leq n$ placed in matrix R^l are no longer in matrix C^l. Thus we refer to these vectors (shown in bold in Table 3) as C_i^l.

The observation of matrix R^0 revealed that each vector $R_l^0, m \leq l \leq n$ contains the bits set to 1 if a replace operation is needed. Each vector R_i^l in matrix C^l refers to a prefix successfully matched with at most l substitutions and in matrix R^{l+1} we may add one substitution more. Thus we compute each matrix $R^l, 0 < l \leq k$ as follows:

$$\begin{aligned} R_i^l = & \ \mathbf{shl}^i(R_0^l) \ \textbf{or} \ (\mathbf{shl}^{i-1}(D[t_1]) \ \textbf{and} \ C_0^{l-1}) \ \ldots \\ & \textbf{or} \ (\mathbf{shl}(D[t_{i-1}]) \ \textbf{and} \ C_{i-2}^{l-1}) \\ & \textbf{or} \ (D[t_i] \ \textbf{and} \ C_{i-1}^{l-1}), \ 1 \leq i \leq n, \ 1 \leq l \leq k \end{aligned} \tag{8}$$

An example of parallel pattern matching using the Hamming distance for the pattern $P = adbbca$, $k = 3$, $T = adcbcadbbca$ is given in Table 4, Table 5, and

C^2	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a	OR
5	0	0	0	0	0	0	0	0	0	**0**	**0**	**0**	**0**	**0**	**0**	0
6	0	0	0	0	0	0	0	0	**0**	**0**	**1**	**1**	**1**	**1**	1	1
7	0	0	0	0	0	0	0	**0**	**0**	**0**	**1**	**1**	**1**	1	1	1
8	0	0	0	0	0	0	**0**	**0**	**0**	**0**	**1**	**1**	1	1	1	1
9	0	0	0	0	0	**0**	**0**	**1**	**1**	**1**	**1**	1	1	1	1	1
10	0	0	0	0	**0**	**0**	**1**	**1**	**1**	**1**	1	1	1	1	1	1
11	0	0	0	**0**	**0**	**0**	**1**	**1**	**1**	1	1	1	1	1	1	1
12	0	0	**0**	**0**	**0**	**1**	**1**	**1**	1	1	1	1	1	1	1	1
13	0	**0**	**0**	**1**	**1**	**1**	**1**	1	1	1	1	1	1	1	1	1
14	**0**	**0**	**0**	**0**	**1**	**1**	1	1	1	1	1	1	1	1	1	1

R^3	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a	OR
5	0	0	0	0	0	0	0	0	0	**0**	**0**	**0**	**0**	**0**	**0**	0
6	0	0	0	0	0	0	0	0	**0**	**0**	**0**	**0**	**1**	**1**	1	1
7	0	0	0	0	0	0	0	**0**	**0**	**0**	**0**	**1**	**1**	1	1	1
8	0	0	0	0	0	0	**0**	**0**	**0**	**0**	**0**	**1**	1	1	1	1
9	0	0	0	0	0	**0**	**0**	**0**	**0**	**1**	**1**	1	1	1	1	1
10	0	0	0	0	**0**	**0**	**0**	**1**	**1**	**0**	1	1	1	1	1	1
11	0	0	0	**0**	**0**	**0**	**0**	**1**	**1**	1	1	1	1	1	1	1
12	0	0	**0**	**0**	**0**	**0**	**1**	**0**	1	1	1	1	1	1	1	1
13	0	**0**	**0**	**0**	**0**	**0**	**0**	1	1	1	1	1	1	1	1	0
14	**0**	**0**	**0**	**0**	**0**	**1**	1	1	1	1	1	1	1	1	1	1

Table 6. Matrices C^2, and R^3 for pattern matching using the Hamming distance, $T = adcbcadbbca$, $k = 3$, $P = adbbca$

Table 6, respectively. The first table contains matrix C^0, which has been corrected by Formula 7 from matrix R^0, and matrix R^1, computed by Formula 8. The second table contains matrix C^1 corrected by Formula 7 from matrix R^1, and matrix R^2 computed by Formula 8. The last table contains matrix C^2 corrected by Formula 7 from matrix R^2, and matrix R^3 computed by Formula 8. These tables are shortened, as described above.

3.2 Parallel pattern matching using the Levenshtein distance

Parallel pattern matching using the Hamming distance is very similar to the pattern matching using the Levenshtein distance, though in addition we must consider the operations "insert" and "delete".

Recall Formula 3. We computed $R^l_i, 0 < i \leq n, 0 < l \leq k$ in the sequential algorithm as:

$$\begin{aligned} R^l_i = {} & (\mathbf{shl}(R^l_{i-1})\ \mathbf{or}\ D[t_i]) \\ & \mathbf{and\ shl}(R^{l-1}_{i-1}\ \mathbf{and}\ R^{l-1}_i) \\ & \mathbf{and}\ (R^{l-1}_{i-1}\ \mathbf{or}\ V), \qquad 0 < i \leq n, 0 < l \leq k \end{aligned} \tag{9}$$

We use exactly the same logic rules for operations "insert" and "delete", and we may formulate for the Levenshtein distance:

$$
\begin{aligned}
R_i^l = {} & \mathbf{shl}^i(R_0^l) \\
& \mathbf{or}\ (\mathbf{shl}^{i-1}(D[t_1])\ \mathbf{and}\ C_0^{l-1}\ \mathbf{and}\ \mathbf{shl}(C_1^{l-1})\ \mathbf{and}\ (\mathbf{shr}(C_0^{l-1})\ \mathbf{or}\ V)) \\
& \dots \\
& \mathbf{or}\ (\mathbf{shl}(D[t_{i-1}]\ \mathbf{and}\ C_{i-2}^{l-1}\ \mathbf{and}\ \mathbf{shl}(C_{i-1}^{l-1})\ \mathbf{and}\ (\mathbf{shr}(C_{i-2}^{l-1})\ \mathbf{or}\ V)) \\
& \mathbf{or}\ (D[t_i]\ \mathbf{and}\ C_{i-1}^{l-1}\ \mathbf{and}\ \mathbf{shl}(C_i^{l-1})\ \mathbf{and}\ (\mathbf{shr}(C_{i-1}^{l-1})\ \mathbf{or}\ V)), \\
& 1 \le i \le n,\ 1 \le l \le k
\end{aligned}
\tag{10}
$$

C^0	$-$	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a	OR
3	0	0	0	0	0	0	0	0	0	0	0	0	**1**	**1**	**1**	**1**	
4	0	0	0	0	0	0	0	0	0	0	0	**1**	**1**	**1**	**1**	**1**	
5	0	0	0	0	0	0	0	0	0	0	**0**	**0**	**0**	**0**	**0**	**0**	0
6	0	0	0	0	0	0	0	0	0	**1**	**1**	**1**	**1**	**1**	**1**	1	1
7	0	0	0	0	0	0	0	0	**0**	**1**	**1**	**1**	**1**	**1**	1	1	1
8	0	0	0	0	0	0	0	**0**	**1**	**1**	**1**	**1**	**1**	1	1	1	1
9	0	0	0	0	0	0	**1**	**1**	**1**	**1**	**1**	**1**	1	1	1	1	1
10	0	0	0	0	0	**1**	**1**	**1**	**1**	**1**	**1**	1	1	1	1	1	1
11	0	0	0	0	**0**	**1**	**1**	**1**	**1**	**1**	1	1	1	1	1	1	1
12	0	0	0	**1**	**1**	**1**	**1**	**1**	**1**	1	1	1	1	1	1	1	1
13	0	0	**1**	**1**	**1**	**1**	**1**	**1**	1	1	1	1	1	1	1	1	1
14	0	**0**	**0**	**1**	**1**	**1**	**1**	1	1	1	1	1	1	1	1	1	1
15	**1**	**1**	**1**	**1**	**1**	**1**	1	1	1	1	1	1	1	1	1	1	1
16	**1**	**1**	**1**	**1**	**1**	1	1	1	1	1	1	1	1	1	1	1	1

R^1	$-$	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a	OR
3	0	0	0	0	0	0	0	0	0	0	0	0	**0**	**1**	**1**	**1**	
4	0	0	0	0	0	0	0	0	0	0	0	**0**	**0**	**0**	**0**	**0**	
5	0	0	0	0	0	0	0	0	0	0	**0**	**0**	**0**	**0**	**0**	**0**	0
6	0	0	0	0	0	0	0	0	0	**0**	**0**	**0**	**0**	**0**	**0**	1	0
7	0	0	0	0	0	0	0	0	**0**	**0**	**1**	**1**	**1**	**1**	1	1	1
8	0	0	0	0	0	0	0	**0**	**0**	**0**	**1**	**1**	**1**	1	1	1	1
9	0	0	0	0	0	0	**0**	**0**	**1**	**0**	**1**	**1**	1	1	1	1	1
10	0	0	0	0	0	**0**	**1**	**1**	**1**	**1**	**0**	1	1	1	1	1	1
11	0	0	0	0	**0**	**0**	**1**	**1**	**1**	**1**	1	1	1	1	1	1	1
12	0	0	0	**0**	**0**	**0**	**1**	**1**	**0**	1	1	1	1	1	1	1	1
13	0	0	**0**	**0**	**1**	**0**	**0**	**0**	1	1	1	1	1	1	1	1	1
14	0	**0**	**0**	**0**	**1**	**1**	**1**	1	1	1	1	1	1	1	1	1	1
15	**0**	**0**	**0**	**1**	**1**	**1**	1	1	1	1	1	1	1	1	1	1	1
16	**1**	**1**	**1**	**0**	**0**	1	1	1	1	1	1	1	1	1	1	1	1

Table 7. Matrices C^0 and R^1 for pattern matching using the Levenshtein distance, $T = adcbcadbbca$, $k = 2$, $P = adbbca$

When computing vector R_i^l, each term C_{i-1}^{l-1} represents operation "replace" as when using the Hamming distance. Since this bit-vector has been already shifted by $n - i + 1$ positions once, that is one more than $n - i$ shifts when computing the i-th column, thus no further shift operation is needed to transform the sequential term $\mathbf{shl}(R_{i-1}^{l-1})$ into a parallel term.

The term ($\mathbf{shr}(C_{i-1}^{l-1})$ **or** V) represents the operation "insert". This vector has been shifted too much by the same logic as the bit-vector for operation "replace".

Therefore we need to shift it one bit back, when transforming the sequential "insert" term (R_{i-1}^{l-1} **or** V) into a parallel term.

The term $\mathbf{shl}(C_i^{l-1})$ represents the operation "delete". The bit-vector C_i^{l-1} has been shifted exactly enough to shift it once more for the same reason as in the sequential algorithm.

Due to the operations "delete" and "insert" we need to shorten the original matrix R^0 less than when using the Hamming distance. We need the initial bit-vector R_0^l defined by Formula 3, which sets more states initial because of the "delete" ε-transitions from the initial state. We also need k rows before the $(m-1)$-th row for the operation "delete" and we need k rows after the $(n-1)$-th row for operation "insert".

C^1	$-$	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a	OR
3	0	0	0	0	0	0	0	0	0	0	0	0	**0**	**1**	**1**	**1**	
4	0	0	0	0	0	0	0	0	0	0	0	**0**	**0**	**0**	**0**	**0**	
5	0	0	0	0	0	0	0	0	0	0	**0**	**0**	**0**	**0**	**0**	**0**	0
6	0	0	0	0	0	0	0	0	0	**0**	**0**	**0**	**0**	**0**	**0**	1	0
7	0	0	0	0	0	0	0	0	**0**	**0**	**1**	**1**	**1**	**1**	1	1	1
8	0	0	0	0	0	0	0	**0**	**0**	**0**	**1**	**1**	**1**	1	1	1	1
9	0	0	0	0	0	0	**0**	**0**	**1**	**1**	**1**	**1**	1	1	1	1	1
10	0	0	0	0	0	**0**	**1**	**1**	**1**	**1**	**1**	1	1	1	1	1	1
11	0	0	0	0	**0**	**0**	**1**	**1**	**1**	**1**	1	1	1	1	1	1	1
12	0	0	0	**0**	**0**	**0**	**1**	**1**	**1**	1	1	1	1	1	1	1	1
13	0	0	**0**	**0**	**1**	**1**	**1**	**1**	1	1	1	1	1	1	1	1	1
14	0	**0**	**0**	**0**	**1**	**1**	**1**	1	1	1	1	1	1	1	1	1	1
15	**0**	**0**	**0**	**1**	**1**	**1**	1	1	1	1	1	1	1	1	1	1	1
16	**1**	**1**	**1**	**1**	**1**	1	1	1	1	1	1	1	1	1	1	1	1
R^2	$-$	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a	OR
3	0	0	0	0	0	0	0	0	0	0	0	0	**0**	**0**	**0**	**0**	
4	0	0	0	0	0	0	0	0	0	0	0	**0**	**0**	**0**	**0**	**0**	
5	0	0	0	0	0	0	0	0	0	0	**0**	**0**	**0**	**0**	**0**	**0**	0
6	0	0	0	0	0	0	0	0	0	**0**	**0**	**0**	**0**	**0**	**0**	1	0
7	0	0	0	0	0	0	0	0	**0**	**0**	**0**	**0**	**0**	**0**	1	1	0
8	0	0	0	0	0	0	0	**0**	**0**	**0**	**0**	**1**	**1**	1	1	1	1
9	0	0	0	0	0	0	**0**	**0**	**0**	**0**	**1**	**1**	1	1	1	1	1
10	0	0	0	0	0	**0**	**0**	**0**	**1**	**1**	**0**	1	1	1	1	1	1
11	0	0	0	0	**0**	**0**	**0**	**1**	**1**	**1**	1	1	1	1	1	1	1
12	0	0	0	**0**	**0**	**0**	**0**	**1**	**0**	1	1	1	1	1	1	1	1
13	0	0	**0**	**0**	**0**	**0**	**0**	**0**	1	1	1	1	1	1	1	1	0
14	0	**0**	**0**	**0**	**0**	**1**	**1**	1	1	1	1	1	1	1	1	1	1
15	**0**	**0**	**0**	**0**	**1**	**1**	1	1	1	1	1	1	1	1	1	1	1
16	**0**	**0**	**0**	**0**	**0**	1	1	1	1	1	1	1	1	1	1	1	0

Table 8. Matrices C^1 and R^2 for pattern matching using the Levenshtein distance, $T = adcbcadbbca$, $k = 2$, $P = adbbca$

An example of parallel pattern matching using the Levenshtein distance for the pattern $P = adbbca$, text $T = adcbcadbbca$ and $k = 2$ is given in Table 7 and Table 8. The former table contains matrix C^0 computed by Formula 7 from matrix R^0 in Table 3 and it also contains matrix R^1 computed from matrix C^0 by Formula 10. The latter contains matrix C^1 computed from matrix R^1 by Formula 7 and it also contains matrix R^2 computed from matrix C^1 by Formula 10.

4 Conclusion

We have presented the idea of parallel pattern matching using bit-parallelism (the "shift-or" variation).

We used a two-dimensional memory matrix which enabled very fast parallel pattern matching. Parallel pattern matching for an exact pattern takes $\mathcal{O}(1)$ parallel time, using n processors. The algorithm does not need any concurrent read or write operation, thus it can be implemented on *EREW PRAM* with shared two-dimensional memory. Since the processor-time product is $\mathcal{O}(n)$, it is a cost-optimal algorithm.

Parallel pattern matching using the Hamming distance with k substitutions takes $\mathcal{O}(k)$ parallel time, using n processors. The algorithm also does not need any concurrent read or write operation, thus it can also be implemented on *EREW PRAM* with shared two-dimensional memory. The processor-time product is equal to the sequential time $\mathcal{O}(kn)$, hence this algorithm is also cost-optimal.

Parallel pattern matching using the Levenshtein distance with k substitutions takes $\mathcal{O}(k)$ parallel time, using $max(n+1, n-m+2k+1)$ processors. The algorithm needs a concurrent read operation when reading the same bit-vector $C_i^l, 0 \leq l < k, 1 \leq i \leq n-1$, thus it can be implemented on *CREW PRAM* with shared two-dimensional memory. The processor-time product is also equal to the sequential time $\mathcal{O}(kn)$.

Known parallel pattern matching algorithms are derived from dynamic programming, which takes $\mathcal{O}(mn)$ sequential time and therefore these parallel pattern matching algorithms are non-optimal. Parallel pattern matching derived from the bit-parallel algorithms can provide an optimal parallel pattern matching algorithm even when not using two-dimensional memory based on some ideas mentioned in this paper.

References

1. R. A. Baeza-Yates and G. H. Gonnet: *A new approach to text searching.* Commun. ACM, 35(10) 1992, pp. 74–82.
2. B. Dömölki: *An algorithm for syntactic analysis.* Computational Linguistics, 8 1964, pp. 29–46.
3. Z. Galil: *A constant-time optimal parallel string-matching algorithm*, in Proceedings of the twenty-fourth annual ACM symposium on Theory of Computing, ACM Press, 1992, pp. 69–76.
4. R. W. Hamming: *Coding and information theory (2nd ed.)*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986.
5. J. Holub: *Simulation of Nondeterministic Finite Automata in Pattern Matching*, Ph.D. Thesis, Czech Technical University in Prague, Feb. 2000.
6. J. JÁJÁ: *An Introduction to Parallel Algorithms*, Addison-Wesley, 1992.
7. Y. Jiang and A. H. Wright: *O(k) parallel algorithms for approximate string matching.* Neural, Parallel and Scientific Computations, 1 1993, pp. 443–452.
8. G. M. Landau and U. Vishkin: *Fast parallel and serial approximate string matching.* Journal of Algorithms, 10(2) 1989, pp. 157–169.
9. S. Wu and U. Manber: *Fast text searching allowing errors.* Commun. ACM, 35(10) 1992, pp. 83–91.

Efficient Algorithms for (δ, γ, α)-Matching

Kimmo Fredriksson[1]* and Szymon Grabowski[2]

[1] Department of Computer Science, University of Joensuu
PO Box 111, FIN–80101 Joensuu, Finland
kfredrik@cs.joensuu.fi

[2] Technical University of Łódź, Computer Engineering Department,
Al. Politechniki 11, 90–924 Łódź, Poland
sgrabow@kis.p.lodz.pl

Abstract. We propose new algorithms for (δ, γ, α)-matching. In this string matching problem we are given a pattern $P = p_0 p_1 \ldots p_{m-1}$ and a text $T = t_0 t_1 \ldots t_{n-1}$ over some integer alphabet $\Sigma = \{0 \ldots \sigma - 1\}$. The pattern symbol p_i matches the text symbol t_j iff $|p_i - t_j| \leq \delta$. The pattern P (δ, γ)-matches some text substring $t_j \ldots t_{j+m-1}$ iff for all i it holds that $|p_i - t_{j+i}| \leq \delta$ and $\sum |p_i - t_{j+i}| \leq \gamma$. Finally, in (δ, γ, α)-matching we also permit at most α length gaps (text substrings) between each matching text symbol. The only known previous algorithm runs in $O(mn)$ time. We give several algorithms that improve the average case up to $O(n)$ for small α, and the worst case to $O(\mathbf{min}\{mn, |\mathcal{M}|\alpha\})$ or $O(mn \log \gamma / w)$, where $\mathcal{M} = \{(i, j) \mid |p_i - t_j| \leq \delta\}$ and w is the number of bits in a machine word. We conclude with experimental results showing that the algorithms are very efficient in practice.

Keywords: approximate string matching, music information retrieval, bit-parallelism, sparse dynamic programming

1 Introduction

Background and problem setting. Many notions of approximateness have been proposed in string matching literature, usually motivated by some real problems. One of seemingly underexplored problems with applications in music information retrieval and molecular biology is (δ, γ, α)-matching [4] and its variations. In this problem, the pattern $p_0 p_1 \ldots p_{m-1}$ is allowed to match a substring of the text $t_0 t_1 \ldots t_{n-1}$ with α-limited gaps, and the respective pairs of matching characters' numerical values may differ only by δ, and the total sum of differences is limited to γ. Translating this model into a music (melody seeking) application, we can allow for small distortions of the original melody because the (presumably unskilled) human user may sing or whistle the melody imprecisely. The gaps, on the other hand, allow to skip over ornamenting notes (e.g., arpeggios), which appear especially in classical music. Other assumptions here, that is, monophonic melody and using pitch values only (without note durations), are reasonable in most practical cases.

Previous work. There are many algorithms that solve some restricted variant of (δ, γ, α)-matching, such as δ-matching [3], (δ, γ)-matching [5,6] and (δ, α)-matching [13,1,2,8]. There are also algorithms that allow transpositions and insertions and deletions of symbols simultaneously with (δ, γ) or (δ, α)-matching [11,12]. However, none of these algorithms can handle (δ, γ, α)-matching. We are aware of only one algorithm for (δ, γ, α)-matching problem [4]. This is based on dynamic programming, and runs in $O(nm)$ time.

* Supported by the Academy of Finland, grant 202281.

Our results. We improve the basic dynamic programming based algorithm [4] to run in $O(n\alpha\delta/\sigma)$ average time. We develop a simple sparse dynamic programming algorithm that runs in $O(n)$ average time, and in $O(\mathbf{min}\{mn, |\mathcal{M}|\alpha\})$ worst case time, where $\mathcal{M} = \{(i,j) \mid p_i =_\delta t_j\}$. Finally, we develop a bit-parallel dynamic programming algorithm that runs in $O(mn\log(\gamma)/w + n\delta)$ worst case time, where w is the number of bits in computer word. The average time of this algorithm is close to $O(n\log(\gamma)/w\ \alpha\delta/\sigma + n)$. The average case analyzes assume that α is small enough.

2 Preliminaries

Let the pattern $P = p_0p_1p_2\ldots p_{m-1}$ and the text $T = t_0t_1t_2\ldots t_{n-1}$ be numerical strings, where $p_i, t_j \in \Sigma$ for $\Sigma = \{0, 1, \ldots, \sigma-1\}$. The number of distinct symbols in the pattern is denoted by σ_{P}.

In δ-approximate string matching the symbols $a, b \in \Sigma$ match, denoted by $a =_\delta b$, iff $|a-b| \le \delta$. Pattern P (δ,α)-matches the text substring $t_{i_0}t_{i_1}t_{i_2}\ldots t_{i_{m-1}}$, if $p_j =_\delta t_{i_j}$ for $j \in \{0,\ldots,m-1\}$, where $0 < i_{j+1} - i_j \le \alpha+1$. Finally, in (δ,γ,α)-matching we require also that $\sum |p_j - t_{i_j}| \le \gamma$. If string A (δ,γ,α)-matches string B, we sometimes write $A =^\alpha_{\delta,\gamma} B$.

In all our analysis we assume uniformly random distribution of characters in T and P, and constant δ and σ. Note that $\gamma < m\delta$, as otherwise (δ,γ,α)-matching degenerates into (δ,α)-matching. It is also meaningless to have $\delta > \gamma$.

For the bit-parallel algorithms we number the bits from the least significant bit (0) to the most significant bit $(w-1)$. C–like notation is used for the bit-wise operations of words; & is bit-wise `and`, | is `or`, ~ negates all bits, << is shift to left, and >> shift to right, both with zero padding.

3 Dynamic programming

A straight-forward solution to (δ,γ,α)-matching is to use dynamic programming. The following recurrence can be used:

$$D_{i,j} = \begin{cases} D_{i-1,j'} + |p_i - t_j|, p_i =_\delta t_j \text{ AND } 0 < j - j' \le \alpha+1, \ \mathbf{min}\, D_{i-1,j'} \le \gamma \\ \gamma + 1, \qquad\qquad \text{otherwise.} \end{cases} \tag{1}$$

If $D_{m-1,j} \le \gamma$, then $P =^\alpha_{\delta,\gamma} t_h \ldots t_j$ for some h. The matrix D is simple to compute in $O(\alpha mn)$ time. As we are only interested in the matching text positions, the $O(mn)$ space complexity can be easily improved. Using row-wise computation only the current and the previous rows need to be in memory, and hence the space complexity is just $O(n)$. For column-wise computation the space complexity is $O(\alpha m)$ as up to $\alpha+1$ columns have to be stored.

As shown in [4] the time complexity can be improved to $O(mn)$ using *min-queue* data structures [9]. However, in practical MIR applications α is usually so small that the simple brute-force evaluation is faster than using sophisticated data structures that have large (constant) overhead. Instead, we propose a simple cut-off trick that improves the average case.

3.1 Cut-off

We make the following observation: if $D_{i\ldots m-1,j-\alpha\ldots j} > \gamma$, for some i, j, then $D_{i+1\ldots m-1,j+1} > \gamma$. This is because there is no way the recurrence can introduce

Alg. 1 DPCO$(T, n, P, m, \delta, \gamma, \alpha)$.

```
for i ← 0 to α + 1 do for j ← 0 to m − 1 do D[i][j] ← γ + 1
for j ← 0 to m − 1 do C[j] ← −α − 1
D[0][0] ← |T[0] − P[0]|
if D[0][0] > δ then D[0][0] ← γ + 1
if D[0][0] ≤ γ then C[0] ← 0
top ← m − 1
for i ← 1 to n − 1 do
    C′ ← C[0]
    k ← i % (α + 2)
    D[k][0] ← |T[i] − P[0]|
    if D[k][0] > δ then D[k][0] ← γ + 1
    if D[k][0] ≤ γ then C[0] ← i
    for j ← 1 to top do
        d ← |T[i] − P[j]|
        min ← γ + 1
        if d ≤ δ AND i − C′ ≤ α + 1 then
            k′ ← (i − 1) % (α + 2)
            min ← D[k′][j − 1]
            for h ← max{0, i − α − 1} to i − 2 do
                k′ ← h % (α + 2)
                if D[k′][j − 1] < min then min ← D[k′][j − 1]
        D[k][j] ← min + d
        C′ ← C[j]
        if D[k][j] ≤ γ then
            C[j] ← i
            if j = m − 1 then report match
    while top ≥ 0 AND i − C[top] > α + 1 do top ← top − 1
    if top < m − 1 then top ← top + 1
```

any other value for those matrix cells. In other words, if $p_0 \dots p_i$ does not (δ, γ, α)-match $t_h \dots t_{j-k}$ for any $k = 0 \dots \alpha$, then the match at the position $j+1$ cannot be extended to $p_0 \dots p_{i+1}$. This can be utilized by keeping track of the highest row number *top* of the current column j such that $D_{top+1 \dots m-1, j-\alpha \dots j} > \gamma$, and computing the next column only up to row $top + 1$. For this sake we maintain an array C so that $C[i]$ gives the largest j such that $p_0 \dots p_i =^{\alpha}_{\delta,\gamma} t_h \dots t_j$. This is easy to do in $O(1)$ time per accessed matrix cell. Alg. 1 shows the complete pseudo code.

Now consider the average time of this algorithm. Computing a single cell $D_{i,j}$ costs $O(\alpha)$ in the worst case. However, this happens only if $p_0 \dots p_{i-1} =^{\alpha}_{\delta,\gamma} t_h \dots t_{j'}$ and $p_i =_\delta t_j$ for some $j' \geq j - \alpha - 1$, and otherwise the cost is just $O(1)$. Therefore on average each cell is computed in $O(\alpha\delta/\sigma)$ time. Maintaining *top* costs only $O(n)$ time in total, since it can be incremented only by one per text character, and the number of decrements cannot be larger than the number of increments. The average time of this algorithm also depends on the average value of *top*, i.e. the total time is $O(n \operatorname{avg}(top)\, \alpha\delta/\sigma)$. For $\gamma = \infty$ it can be shown that $\operatorname{avg}(top) = O\left(\frac{\delta}{\sigma(1-\delta/\sigma)^{\alpha+1}}\right)$ [2]. This is $O(\alpha\delta/\sigma)$ for $\delta/\sigma < 1 - \alpha^{-1/(\alpha+1)}$, so the average time is at most $O(n(\alpha\delta/\sigma)^2)$. We have neglected the effect of γ, but by forcing the γ condition the time can only improve, hence our analysis is pessimistic. In the worst case the time is $O(\alpha mn)$, but this can be improved to $O(mn)$ as in [4], the only difference being that we need m queues, since we are computing column-wise (as opposed to row-wise in [4]).

4 Simple algorithm

In this section we will develop a variant of the Simple algorithm for (δ, α)-matching [7]. This performs very well on small (δ, γ, α).

Alg. 2 Simple($T, n, P, m, \delta, \gamma, \alpha$).

```
h ← 0
for i ← 0 to n − 1 do
    M[i] ← γ + 1
    d ← |T[i] − P[0]|
    if d ≤ δ then
        L1[h] ← i
        G[h] ← d
        h ← h + 1
for j ← 1 to m − 1 do
    pn ← h; h ← 0;
    for i ← 0 to pn − 1 do
        g ← G[i]
        for k ← L1[i] + 1 to min(L1[i] + α + 1, n − 1) do
            d ← |T[k] − P[j]|
            if d ≤ δ AND g + d ≤ γ then
                if M[k] ≤ γ then
                    if g + d < M[k] then M[k] ← g + d
                else
                    L2[h] ← k
                    h ← h + 1
                    M[k] ← g + d
                if j = m − 1 AND M[k] ≥ 0 then
                    report match
                    M[k] ← −1
    if j < m − 1 then for i ← 0 to h − 1 do
        k ← L2[i]
        G[i] ← M[k]
        M[k] ← γ + 1
    Lt ← L1; L1 ← L2; L2 ← Lt;
```

The algorithm begins by computing a list L of δ-matches for p_0:

$$L_0 = \{j \mid t_j =_\delta p_0\}. \tag{2}$$

This takes $O(n)$ time (and solves the (δ, γ, α)-matching problem for patterns of length 1). The matching prefixes are then iteratively extended, subsequently computing lists:

$$L_i = \{j \mid p_i =_\delta t_j \text{ AND } D_{i-1,j'} + |p_i - t_j| \leq \gamma \text{ AND } j' \in L_{i-1} \text{ AND } 0 < j - j' \leq \alpha + 1\}. \tag{3}$$

List L_i can be easily computed by linearly scanning list L_{i-1}, and checking if any of the text characters $t_{j'+1} \ldots t_{j'+\alpha+1}$, for $j' \in L_{i-1}$ δ-matches p_i, and if so whether the sum of errors is still at most γ. When some j is appended into L_i, the corresponding matrix cell $D_{i,j}$ is also updated to hold the sum of errors for the matching pattern prefix $p_0 \ldots p_i$. Note that we put each j only once into L_i, but there can be up to $\alpha + 1$ different $j' \in L_{i-1}$ that may cause it. In the case that j is already in L_i we only update $D_{i,j}$ if the new sum is smaller. This takes $O(\alpha|L_{i-1}|)$ time. Alg. 2 shows the code.

Clearly, in the worst case the total length of all the lists is $\sum_i |L_i| = |\mathcal{M}|$, where $\mathcal{M} = \{(i, j) \mid p_i =_\delta t_j\}$, and hence the algorithm runs in $O(\alpha|\mathcal{M}|)$ worst case time. Consider now the average case. List L_0 is computed in $O(n)$ time. The length of this list is $O(n\delta/\sigma)$ on average. Hence the list L_1 is computed in $O(\alpha n\delta/\sigma)$ average time, resulting in a list L_1, whose average length is $O(n\delta/\sigma \times \alpha\delta/\sigma)$. In general, computing the list L_i takes

$$O(\alpha|L_{i-1}|) = O(n\alpha^i(\delta/\sigma)^i) = O(n(\alpha\delta/\sigma)^i) \tag{4}$$

average time. This is exponentially decreasing if $\alpha\delta/\sigma < 1$, i.e. if $\alpha < \sigma/\delta$, and hence, summing up, the total average time is $O(n)$. Note that we did not use γ in this analysis, making it pessimistic.

4.1 Improving the worst case

As a theoretical option, we can improve the worst case of this algorithm to $O(\mathbf{min}\{mn, \alpha|\mathcal{M}|\})$. The idea is to avoid brute force handling of overlapping windows of size $\alpha + 1$. We make use of the min-queue data structure [9], similarly to the concept from [4] where the min-queue was used with plain dynamic programming.

For the current cell $D_{i+1,j}$, the keys in the queue are the values of $D_{i,j'}$, where $j' \in \{L_i \mid 0 < j - L_i < \alpha+1\}$. For calculating $D_{i+1,j}$ it is enough to add its individual error to the minimum sum of errors from the queue. An algorithmic challenge is to update the queue quickly. For each processed cell only 0 or 1 values have to be inserted to the front of the queue and from 0 to $\alpha + 1$ deleted from the tail. Note however that only $O(1)$ cells (amortized) are inserted or deleted at each step. All the operations can then be done in $O(1)$ time with the min-queue data structure. This gives $O(\mathbf{min}\{mn, \alpha|\mathcal{M}|\})$ worst case time.

Finally, the $O(\alpha)$ factor can be removed by precomputing $\mathcal{M}$. This can be done in $O(\mathbf{min}\{|\mathcal{M}|+n, \delta n\})$ worst case time and $O(n(\delta\sigma_{\text{P}}/\sigma+1))$ average case time for integer alphabets (see Sec. 5). Having $\mathcal{M}$ available, we can avoid the brute force scanning for δ-matches. $\mathcal{M}$ can be stored e.g. in Johnson's data structure [10] which supports a homogeneous sequence of insertions and successor searches in $O(\log\log(mn/|\mathcal{M}|))$ time. This gives $O(|\mathcal{M}| \log\log(mn/|\mathcal{M}|))$ worst case time, but destroys the good average case because of the costly precomputation. Note that $O(|\mathcal{M}| + n)$ worst case algorithm is easy to obtain by simply scanning $\mathcal{M}$ linearly, but this then becomes also the average case.

5 Bit-parallel dynamic programming

We now show how the basic dynamic programming algorithm can be bit-parallelized. The algorithm is based on the bit-parallel dynamic programming algorithm for (δ, α)-matching [8]. All the interesting values in the matrix D are at most γ, and all other values can be represented as any value greater than γ. Hence $O(\log\gamma)$ bits per matrix cell is sufficient, and we can compute $O(w/\log\gamma)$ cells in parallel, where w is the number of bits in a machine word. Moreover, we show how to handle α up to $O(w/\log\gamma)$ efficiently. We obtain $O(mn\log\gamma/w)$ worst case time algorithm.

Each matrix cell is represented with

$$\ell = \lceil \log_2(2\gamma + 1) \rceil \tag{5}$$

bits, and number zero is represented (using ℓ bits) as $2^{\ell-1} - (\gamma+1)$. This representation has been used before e.g. for (δ, γ)-matching [5]. We still need an additional bit per cell, and hence each machine word packs

$$C = \lfloor w/(\ell + 1) \rfloor \tag{6}$$

cells, or *counters*. This representation solves three problems we are going to face shortly: (*i*) counter overflows can be handled in parallel; (*ii*) it is easy to check in parallel if some of the counters have exceeded γ; (*iii*) thanks to the additional bit it is easy to compute pair-wise minima over two sets of counters in parallel.

Assume then that in the preprocessing phase we have computed a helper matrix (whose efficient computation we will consider later) V:

$$V_{i,j} = \begin{cases} |p_i - t_j|, & p_i =_\delta t_j \\ \gamma + 1, & \text{otherwise.} \end{cases} \tag{7}$$

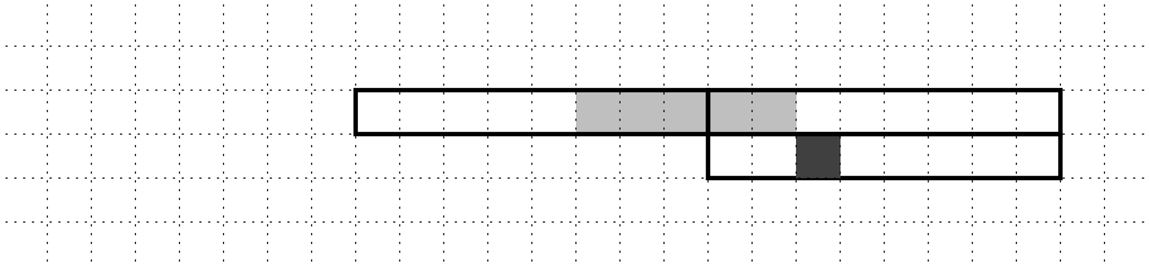


Figure 1. Tiling the dynamic programming matrix with $C = \lfloor w/(\ell+1) \rfloor \times 1$ vectors ($C = 8$). The dark gray cell of the current tile depends on the light gray cells of the two tiles in the previous row ($\alpha = 4$).

The computation of D will proceed column-wise, C columns at once. We adopt the notation $D^C_{i,j} = D_{i,jC\ldots(j+1)C-1}$, and analogously V^C for V, to make the parallelism explicit. Assume now that $\alpha < C$. The goal is then to produce $D^C_{i,j}$ from $V^C_{i,j}$, $D^C_{i-1,j}$ and $D^C_{i-1,j-1}$. $D^C_{i,j}$ does not depend on any other D^C element, according to the definition of D, and given our assumption that $\alpha < C$. Fig. 1 illustrates.

Now, according to the recurrence, the kth counter in $D^C_{i,j}$ is the sum of (i) the kth counter of $V^C_{i,j}$ (i.e. $|p_i - t_{jC+k}|$) and (ii) the minimum of the counters $k-\alpha-1\ldots k-1$ in $D^C_{i-1,j}$ and the counter $k+C-\alpha-1\ldots C-1$ in $D^C_{i-1,j-1}$ (i.e. the gap length to the previous match is at most α), see Fig. 1.

To compute item (ii) efficiently we assume that we have available function $M(x)$, that replaces each counter in x with the minima of the $\alpha+1$ previous counters in x. The recurrence for D^C then becomes:

$$D^C_{i,j} = V^C_{i,j} + (M((D^C_{i-1,j} << w) \mid (D^C_{i-1,j-1} << (w - w \ \% \ C))) >> w), \tag{8}$$

where for simplicity we have assumed that $M(x)$ can handle words of length $2w$. However, the above equation may cause counter overflow. To prevent this we use

$$D^C_{i,j} = (V^C_{i,j} + (M' \ \& \sim hmsk)) \mid (M' \ \& \ hmsk), \tag{9}$$

instead, where

$$M' = M((D^C_{i-1,j} << w) \mid (D^C_{i-1,j-1} << (w - w \ \% \ C))) >> w, \tag{10}$$

and $hmsk$ selects the ℓth bit of each counter. That is, $M' \ \& \sim hmsk$ clears the highest bit of each counter, so that the result can be safely added to $V^C_{i,j}$, and then $\mid (M' \ \& \ hmsk)$ restores the highest bit. This works correctly, as if the highest bit was set, then the sum is certainly greater than γ, and its exact value is not interesting anymore. The $(\ell+1)$th bit is not affected by the summation as the maximum value added is $\gamma+1$.

Finally, to detect the possible pattern occurrences we must add our representation of zero $(2^{\ell-1} - (\gamma+1))$ to each counter. If some of the counters have still not overflowed, the corresponding text positions match. This can be detected as

$$q = \sim(((D^C_{m-1,j} \ \& \sim hmsk) + zeromsk) \mid D^C_{m-1,j}) \ \& \ hmsk, \tag{11}$$

where $zeromsk$ has the value $2^{\ell-1} - (\gamma+1)$ in each counter position. Each set bit in q then indicates a pattern occurrence.

Alg. 3 $\text{vmin}(x, y, msk)$.

$F \leftarrow ((x \mid msk) - y)\ \&\ msk$
$F \leftarrow F - (F >> \ell)$
return $(x\ \&\ {\sim}F) \mid (y\ \&\ F)$

Alg. 4 $M(x, y, \alpha, msk)$.

$x \leftarrow (x << w) \mid (y << (w - w\ \%\ C))$
while $\alpha \neq 0$ **do**
  $r \leftarrow \alpha\ \%\ 2$
  $\alpha \leftarrow \lfloor \alpha/2 \rfloor$
  $x \leftarrow \text{vmin}(x, x << ((\ell+1)\alpha), msk)$
  if $r = 0$ **then continue**
  $x \leftarrow \text{vmin}(x, x << (\ell+1), msk)$
return $(x << (\ell+1)) >> w$

Consider now the computation of $M(x)$. One possible solution is to use table look-ups to compute it in constant time. Since w can be too large to make this approach feasible, we can precompute the answers e.g. to only $w/2$ or $w/4$ bit numbers, and correspondingly compute $M(x)$ in 2 or 4 pieces without affecting the time complexity (in our tests we used at most $w/2 = 16$ bit numbers for computing $M(x)$).

Another solution is to use repeated shifting and minimization. That is, assuming that $\text{vmin}(x, y)$ computes pair-wise minima of the counter sets x and y, we compute $x \leftarrow \text{vmin}(x, (x << (\ell+1)) \mid (\gamma+1))$ and repeat that α times, and then perform the final shift $x \leftarrow x << (\ell+1)$, which gives the desired result. The minimization can be done in $O(1)$ time [14], see Alg. 3. The total time for computing $M(x)$ is then $O(\alpha)$. This can be easily improved to $O(\log \alpha)$. Without loss of generality assume that α is a power of two. Instead of shifting one counter position at a time we first shift by $\alpha/2$ counter positions, then $\alpha/4$ counter positions, and so on $\log_2 \alpha$ times, performing the minimization at each step. Alg. 4 shows the code, handling the general case as well. This algorithm takes the counter sets $D^C_{i-1,j}$ and $D^C_{i-1,j-1}$, that can affect the current counters $D^C_{i,j}$, as parameter. For simplicity these are handled as a concatenated single word of $2w$ bits. Eq. (10) then becomes

$$M' = M(D^C_{i-1,j}, D^C_{i-1,j-1}, \alpha, msk), \tag{12}$$

where msk has every $(\ell+1)$th bit set, needed at the counter minimization.

We also need to compute V efficiently. This is easy with table look-ups as we have an integer alphabet. We first compute a table L, such that for all $c \in \Sigma$ the list $L[c]$ contains all the distinct characters p_i that satisfy $p_i =_\delta c$. Using this table we build a table V', which we will use as a terse representation of V, namely we have that $V'[p_i] = V_i$. This can be done by scanning through the text, and setting the jth counter of $V'[c]$ to $|c - t_j|$ for each $c \in L[t_j]$. This process takes $O(\lceil n/C \rceil \sigma_{\text{P}} + m + \sigma + \delta\sigma_{\text{P}} + \delta n) = O(\lceil n/C \rceil \sigma_{\text{P}} + \delta n)$ worst case time. The probability that two characters δ-match is at most $(2\delta+1)/\sigma$, and hence the expected number of matching pattern characters for each text character is $O(\delta\sigma_{\text{P}}/\sigma)$. Therefore, the average case complexity of the preprocessing is $O(\lceil n/C \rceil \sigma_{\text{P}} + n(\delta\sigma_{\text{P}}/\sigma + 1))$. Searching clearly takes only $O(\lceil n/C \rceil m) = O(\lceil n \log \gamma / w \rceil m)$ time if table look-ups are used for computing $M(x)$, and $O(\lceil n \log \alpha \log \gamma / w \rceil m)$ if Alg. 4 is used. For α larger than $O(w/\log \gamma)$ the search time must be multiplied by $O(\lceil \alpha \log \gamma / w \rceil)$.

Extended patterns. We note that this algorithm can be easily adapted to handle character classes, both in the pattern and the text. I.e. the pattern and text symbols can be subsets of the alphabet, that is, $p_i, t_j \subseteq \Sigma$. The search algorithm does not change, we just change the definition (and preprocessing) of V:

$$V_{i,j} = \begin{cases} \mathbf{min}\, |p - t|, p =_\delta t \text{ AND } p \in p_i, t \in t_j \\ \gamma + 1, \qquad \text{otherwise.} \end{cases} \tag{13}$$

5.1 Cut-off

The cut-off trick used in Sec. 3.1 obviously works for the bit-parallel algorithm as well. More formally, we define (for D^C) the maximum row top^C_j for the column j as:

$$top^C_j = \text{argmax}_i\{\ (\text{MatchMsk}(D^C_{i-1,j-1}) >> ((C - \alpha - 1)(\ell + 1))) \neq 0 \text{ OR} \tag{14}$$

$$(\text{MatchMsk}(D^C_{i-1,j}) << (\ell + 1)) \neq 0\ \}, \tag{15}$$

where

$$\text{MatchMsk}(x) = \sim(((x\ \&\ \sim hmsk) + zeromsk) \mid x)\ \&\ hmsk. \tag{16}$$

Consider first the part (14). The rationale is as follows. When we are computing $D^C_{i,j}$, only the last $\alpha + 1$ counters of $D^C_{i-1,j-1}$ that are at most γ can affect the counters in $D^C_{i,j}$. We therefore select the corresponding counter bits that indicate whether or not the sum have exceeded γ. However, since we are computing C columns in parallel, the $C - 1$ first counters that have a value of at most γ in $D^C_{i-1,j}$ (15), i.e. in the previous row of the *current* set of columns, can affect the counters in $D^C_{i,j}$ as well. Obviously, this second part cannot be computed at column $j-1$. We solve this simply by computing the first part of top^C_j after the column $j - 1$ have been computed, and when processing the column j, we increase top^C_j if needed according to the second part (15).

Alg. 5 gives the pseudo code. It uses the $O(\log \alpha)$ time algorithm for the $M(\cdot)$ function. The average case running time of this algorithm depends on what is the average value of top^C. For $C = 1$ and $\gamma = \infty$ avg$(top^1) = O(\frac{\delta}{\sigma(1-\delta/\sigma)^{\alpha+1}})$, see Sec. 3.1. We are not able to analyze avg(top^C) exactly, but we have trivially that avg$(top^1) \leq$ avg$(top^C) \leq$ avg$(top^1) + C - 1$, and hence the amortized average search time of Alg. 5 is at most $O((\lceil n/C \rceil \lceil \alpha\delta/\sigma \rceil + n) \log \alpha)$. The $\log \alpha$ factor can be easily removed with precomputation.

5.2 Lazy preprocessing

This can be still improved by interweaving the preprocessing and search phases, so that we initialize and preprocess V^C only for top^C_j length prefixes of the pattern for each j. At the time of processing the column j, we only know top^C_{j-1}, so we use an estimate $\varepsilon \times top^C_{j-1}$ for top^C_j, where $\varepsilon > 1$ is a small constant. If this turns out to be too small, we just increase the estimate and re-preprocess for the current column. The total preprocessing cost on average then becomes only $O(\lceil n/C \rceil \sigma_{top^C} \delta/\sigma + n)$, where σ_{top^C} is the alphabet size of top^C length prefix of the pattern. Hence the initialization time is at most $O(\lceil n/C \rceil \lceil \alpha\delta/\sigma \rceil + n)$ on average. This matches the search time, and together with the preprocessing the total is $O(\lceil n/C \rceil \lceil \alpha\delta/\sigma \rceil + n \lceil \alpha\delta/\sigma \rceil \delta/\sigma + n)$ on average.

Alg. 5 BPCO$(T, n, P, m, \delta, \gamma, \alpha)$.

```
ℓ ← ⌈log2(2γ + 1)⌉
f ← (w/(ℓ + 1))
Qb ← (f/2)(ℓ + 1)
zmsk ← (1 << (ℓ + 1)) − 1
for i ← 0 to σ − 1 do A[i] ← 0
for i ← 0 to m − 1 do
    if A[P[i]] then continue
    A[P[i]] ← 1
    for j ← max{0, P[i] − δ} to min{P[i] + δ, σ − 1} do
        Lt[j] ← Lt[j] ∪ {P[i]}
zero ← (1 << (l − 1)) − (γ + 1)
hhmsk ← 0
for i ← 0 to f − 1 do hhmsk ← hhmsk | (1 << ((i + 1)(ℓ + 1) − 1))
hmsk ← hhmsk >> 1
b ← (n + f − 1)/f
for i ← 0 to σ − 1 do
    V[i] ← 0
    if A[i] ≠ 0 then
        for j ← 0 to b − 1 do V[i][j] ← hmsk
for i ← 0 to n − 1 do
    for j ← 0 to |Lt[T[i]]| − 1 do
        c ← Lt[T[i]][j]
        V[c][i/f] ← V[c][i/f] & ∼(zmsk << ((i % f)(ℓ + 1)))
        V[c][i/f] ← V[c][i/f] | (|c − T[i]| << ((i % f)(ℓ + 1)))
top ← m − 1
D1[0] ← V[P[0]][0]
for i ← 1 to top do
    x ← M(D1[i − 1], hmsk, α, hhmsk)
    D1[i] ← (V[P[i]][0] + (x & ∼hmsk)) | (x & hmsk)
zeromsk ← 0
for i ← 0 to f − 1 zeromsk ← zeromsk | (zero << (i(ℓ + 1)))
x ← ∼(((D1[m − 1] & ∼hmsk) + zeromsk) | D1[m − 1]) & hmsk
if x ≠ 0 then report matches
k ← ((f − α − 1)(ℓ + 1))
for j ← 1 to b − 1 do
    D2[0] ← V[P[0]][j]
    if top = 0 then
        if (∼(((D2[0] & ∼hmsk) + zeromsk) | D2[0]) & hmsk) ≠ 0 then D1[0] ← hmsk; top ← top + 1
    for i ← 1 to top do
        x ← M(D2[i − 1], D1[i − 1], α, hhmsk)
        D2[i] ← (V[P[i]][j] + (x & ∼hmsk)) | (x & hmsk)
        x ← ∼(((D2[i] & ∼hmsk) + zeromsk) | D2[i]) & hmsk
        if i = top AND top < m − 1 AND (x << (ℓ + 1)) ≠ 0 then D1[i] ← hmsk; top ← top + 1
    if top = m − 1 AND x ≠ 0 then report matches
    do  x ← (∼(((D2[top] & ∼hmsk) + zeromsk) | D2[top]) & hmsk) >> k
        if x = 0 then top ← top − 1
    while top ≥ 0 AND x = 0
    if top < m − 1 top ← top + 1
    Dt ← D1; D1 ← D2; D2 ← Dt
```

5.3 Multiple patterns

The algorithm has relatively high preprocessing cost $O(\delta n + \sigma_{\text{P}}\lceil n/C\rceil)$ in the worst case. However, if we want to search a set of r patterns, instead of only one pattern, the preprocessing remains essentially the same, since it depends only on the text and the pattern alphabet. The total (worst case) preprocessing time increase only $O(\delta n + \sigma_{\text{P}}\lceil n/C\rceil + rm)$, where we have pessimistically considered that m is the length of the longest pattern in the set, and that σ_{P} is the number of distinct symbols in the whole pattern set. The search times have to be multiplied by r, but the amortized preprocessing cost per pattern is considerably reduced. If r is small as compared to σ/δ, the search cost can be reduced by "superimposing" the patterns, that is we define

$$V_{i,j} = \begin{cases} \mathbf{min}\,|p - t_j|, p =_\delta t_j \text{ AND } p \in p_i^h, h \in 0 \ldots r-1 \\ \gamma + 1, \qquad \text{otherwise,} \end{cases} \tag{17}$$

where we use the notation p_i^h to denote the ith symbol of the hth pattern. We then need only one search, but the potential matches must be verified. Superimposing works for the other algorithms as well.

5.4 Filtering

Alg. 5 is substantially more complex than its ancestor, the (δ, α)-matching algorithm [8]. In addition to being simpler, the previous algorithm achieves greater parallelism, the worst case search time being only $O(\lceil n/w \rceil m)$. However, we note that this algorithm (as any (δ, α)-matching algorithm) can be used as a filter, since it implicitly assumes that $\gamma = \infty$. The potential occurrences have to verified, which can be done using any of the algorithms given in this paper. The worst case time then becomes that of the verification algorithm.

6 Experimental results

We have run experiments to evaluate the performance of our algorithms. The experiments were run on Pentium4 2.4GHz with 512Mb of RAM, running GNU/Linux 2.4.20 operating system. We have implemented all the algorithms in C, and compiled with `icc 9.0`.

For the text we used a concatenation of 7543 music pieces, obtained by extracting the pitch values from MIDI files. The total length is 1,828,089 bytes. The pitch values are in the range $[0 \ldots 127]$. This data is far from random; the six most frequent pitch values occur 915,082 times, i.e. they cover about 50% of the whole text, and the total number of different pitch values is just 55. We also repeated the experiments on uniformly random data, with $\sigma = 128$. A set of 100 patterns were randomly extracted from the text. Each pattern was then searched for separately, and we report the average user times.

We experimented with the following algorithms:

BP Cut-off Bit-parallel dynamic programming with cut-off, Alg. 5 (without the lazy preprocessing);
BP Filter The (δ, α)-matching version of BP Cut-off [8] used as a filter, and Alg. 1 used for the verifications;
DP Cut-off Dynamic programming with cut-off, Alg. 1;
Simple Simple sparse dynamic programming, Alg. 2.

We omitted the results for basic dynamic programming based algorithms, since these are orders of magnitude slower. Fig. 2 shows the timings. Simple is the clear winner in most of the cases. BP Cut-off suffers from the large preprocessing cost, especially if the pattern alphabet is large. The same is true for the BP Filter, but this is more competitive in MIDI data, where the pattern alphabet is effectively very small.

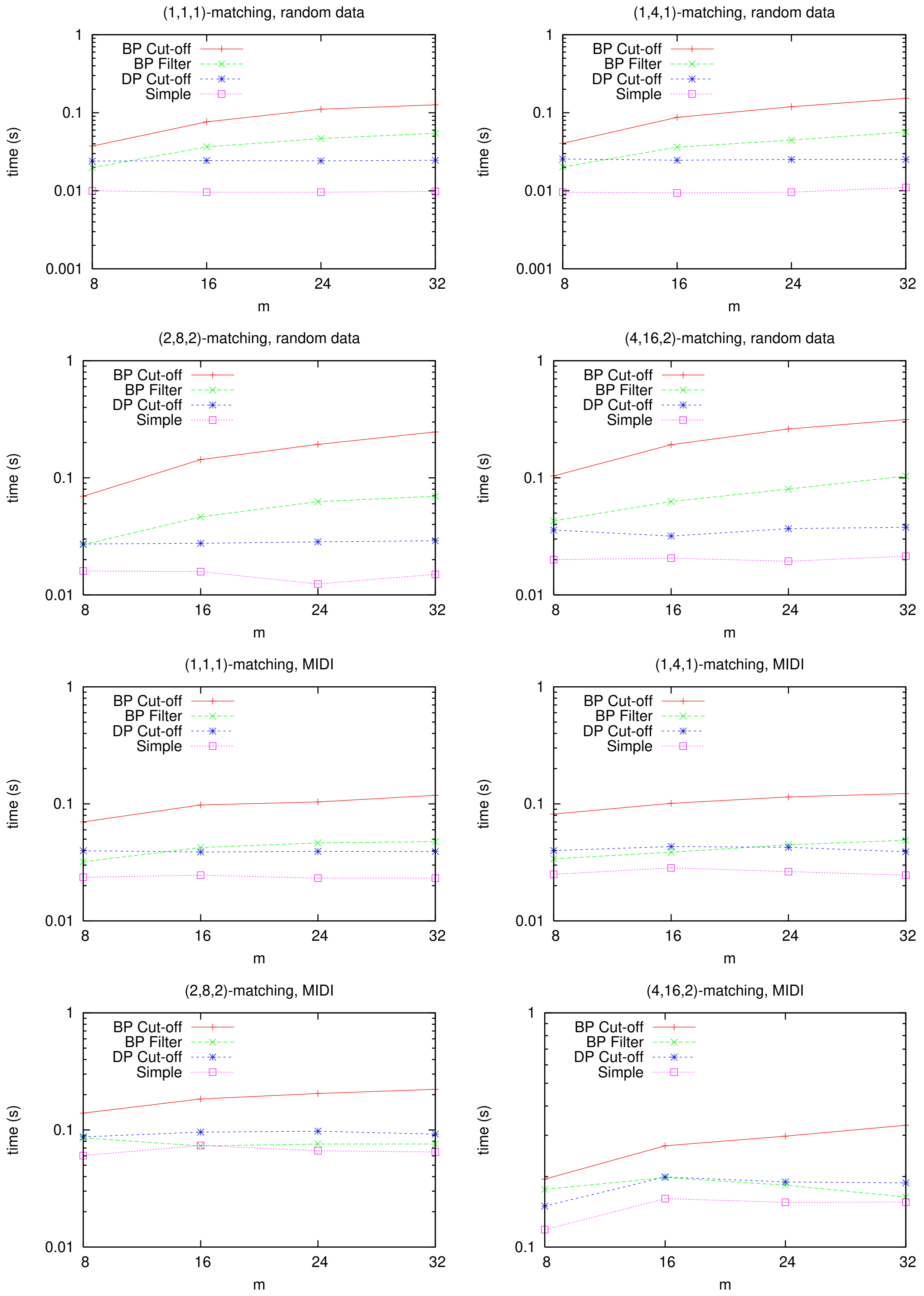


Figure 2. Execution times in seconds for $m = 8 \dots 32$. Note the logarithmic scale.

7 Conclusions

We have presented new efficient algorithms for (δ, γ, α)-matching. Our algorithms are based on aborting the computation early where the match cannot be extended and on bit-parallelism. Besides having theoretically good worst and average case complexities, the algorithms are shown to work well in practice.

References

1. D. Cantone, S. Cristofaro, and S. Faro: *An efficient algorithm for δ-approximate matching with α-bounded gaps in musical sequences.*, in Proceesings of WEA'05, vol. 3503 of LNCS, Springer, 2005, pp. 428–439.
2. D. Cantone, S. Cristofaro, and S. Faro: *On tuning the (δ, α)-sequential-sampling algorithm for δ-approximate matching with α-bounded gaps in musical sequences*, in Proceedings of ISMIR'05, 2005.
3. M. Crochemore, C. Iliopoulos, T. Lecroq, Y. Pinzon, W. Plandowski, and W. Rytter: *Occurrence and substring heuristics for δ-matching.* Fundamenta Informaticae, 56(1–2) 2003, pp. 1–15.
4. M. Crochemore, C. Iliopoulos, C. Makris, W. Rytter, A. Tsakalidis, and K. Tsichlas: *Approximate string matching with gaps.* Nordic Journal of Computing, 9(1) 2002, pp. 54–65.
5. M. Crochemore, C. Iliopoulos, G. Navarro, Y. Pinzon, and A. Salinger: *Bit-parallel (δ, γ)-matching suffix automata.* Journal of Discrete Algorithms (JDA), 3(2–4) 2005, pp. 198–214.
6. K. Fredriksson, V. Mäkinen, and G. Navarro: *Flexible music retrieval in sublinear time*, in Proceedings of the 10th Prague Stringology Conference (PSC'05), 2005, pp. 174–188.
7. K. Fredriksson and Sz. Grabowski: *Efficient algorithms for pattern matching with general gaps and character classes*, in Proc. SPIRE'06, 2006, to appear.
8. K. Fredriksson and Sz. Grabowski: *Efficient bit-parallel algorithms for (δ, α)-matching.*, in Proc. 5th Workshop on Efficient and Experimental Algorithms (WEA'06), LNCS 4007, 2006, pp. 170–181.
9. H. Gajewska and R. E. Tarjan: *Dequeus with heap order.* Information Processing Letters, 22(4) 1986, pp. 197–200.
10. D. B. Johnson: *A priority queue in which initialization and queue operations take $O(\log\log D)$ time.* Mathematical Systems Theory, 15 1982, pp. 295–309.
11. V. Mäkinen: *Parameterized approximate string matching and local-similarity-based point-pattern matching*, PhD thesis, Department of Computer Science, University of Helsinki, Aug. 2003.
12. V. Mäkinen, G. Navarro, and E. Ukkonen: *Transposition invariant string matching.* Journal of Algorithms, 56(2) 2005, pp. 124–153.
13. G. Navarro and M. Raffinot: *Fast and simple character classes and bounded gaps pattern matching, with applications to protein searching.* Journal of Computational Biology, 10(6) 2003, pp. 903–923.
14. W. Paul and J. Simon: *Decision trees and random access machines*, in ZUERICH: Proc. Symp. Logik und Algorithmik, 1980, pp. 331–340.

Song Classifications for Dancing

Manolis Christodoulakis[1,4], Costas S. Iliopoulos[1,3,⋆], M. Sohel Rahman[1,3,⋆⋆,⋆⋆⋆], and William F. Smyth[4,5,†]

[1] Algorithm Design Group
Department of Computer Science, King's College London
Strand, London WC2R 2LS, England
http://www.dcs.kcl.ac.uk/adg
[2] Algorithms Research Group,
Department of Computing and Software,
McMaster University, Canada
[3] {csi, sohel}@dcs.kcl.ac.uk
[4] manolis.christodoulakis@kcl.ac.uk
[5] smyth@mcmster.ca

Abstract. A fundamental problem in music is to classify songs according to their rhythm. A rhythm is represented by a sequence of *Quick* (Q) and *Slow* (S) symbols, which correspond to the (relative) duration of notes, such that $S = QQ$. In this paper we present a linear algorithm for locating the maximum-length substring of a music text t that can be covered by a given rhythm r. An efficient algorithm to solve this problem, can then be used to find which rhythm, from a given set of such rhythms, covers the largest part of the music sequence under question, and thus best describes that sequence.

Keywords: algorithms, music sequence

1 Introduction

The subject of musical representation for use in computer application has been studied extensively in computer science literature [2,1,4,9,13,11]. Computer assisted music analysis [12,10] and music information retrieval [5,8,7,6] has a number of tasks that can be related to fundamental combinatorial problems in computer science and in particular to stringology. A survey of computational tasks arising in music information retrieval can be found in [3]. We, in this paper, are interested in automatic music classification which is one of the fundamental tasks in the area of computational musicology. Songs need to be classified by one or more of their characteristics, like genre, melody, rhythm, etc. For human beings, the process of identifying those characteristics seems natural. Computerized classification though is hard to achieve, given that there does not exist a complete agreement on the definition of those features.

In this work, we will be concerned with classification by dancing rhythm. We will define what a dancing rhythm is, and how it can be identified in a musical sequence, a song. The musical sequences we will be considering consist of a series of onsets (or events) that correspond to music signals, such as drum beats, guitar picks, horn hits, etc. It is the intervals between those events, that characterize how the song is danced.

⋆ Supported by EPSRC and Royal Society grants.
⋆⋆ Supported by the Commonwealth Scholarship Commission in the UK under the Commonwealth Scholarship and Fellowship Plan (CSFP).
⋆⋆⋆ On Leave from Department of CSE, BUET, Dhaka-1000, Bangladesh.
† Supported in part by an NSERC grant.

In particular, there are two types of intervals in the dancing rhythm of a song: *quick* (Q) and *slow* (S). *Quick* means that the duration between two (not necessarily successive) onsets is q milliseconds, while the *slow* interval is equal to $2q$. For example, a cha-cha is given as the sequence $SSQQSSSQQS$ while a foxtrot is given as $SSQQSSQQ$, and a jive is given as $SSQQSQQS$.

The paper is organized as follows. In Section 2 we describe the notation that is used throughout the paper, and we define the terms matching and covering in musical sequences. In Section 3 we describe in detail our algorithm for finding the largest area in a musical sequence that is covered by a given rhythm. As will be seen, under the restrictions we impose on our problem, the algorithms we devise run in linear time. Finally, Section 4 contains our concluding remarks.

2 Definitions

A *musical sequence* t is a string $t = t[1]t[2]\dots t[n]$, where $t[i] \in \mathbb{N}^+$, for all $1 \leq i \leq n$. For example the sequence

$$[0, 50, 100, 200, 250, 300, 350, 400, 500, 550]$$

represents a sequence of events occurring at 0 milliseconds, 50 milliseconds, 100 milliseconds, and so on, in the original music signal. Alternatively, we can represent musical sequences by the duration of the events, as follows

$$[50, 50, 100, 50, 50, 50, 50, 100, 50]$$

The two definitions above are equivalent. We prefer the latter here for the sake of clarity. The above musical sequence can then be represented graphically as shown in the following figure.

50	50	100	50	50	50	50	100	50

0 50 100 200 250 300 350 400 500 550

A *rhythm* r is a string $r = r[1]r[2]\dots r[m]$, where $r[j] \in \{Q, S\}$, for all $1 \leq j \leq m$. For example, $r = QSS$. Q and S correspond to intervals between events, such that the length of an interval represented by an S is double the length of an interval represented by Q. However, the exact length of Q or S is not a priori known. The length m of the rhythm, in practical cases, is usually 10-13 characters and thus we can consider it to be constant.

Let Q represent intervals of size $q \in \mathbb{N}^+$ milliseconds, and S represent intervals of size $2q$. Then Q is said to *match* with the substring $t[i..i']$ of the musical sequence t, if and only if

$$q = t[i] + t[i+1] + \dots + t[i']$$

where $1 \leq i \leq i' \leq n$. If $i = i'$ then the match is said to be *solid*. Similarly, S is said to match with $t[i..i']$ if and only if either of the following is true

– $i = i'$ and $t[i] = 2q$, or
– $i \neq i'$ and there exists $i \leq i_1 < i'$ such that

$$q = t[i] + t[i+1] + \dots + t[i_1] = t[i_1+1] + t[i_1+2] + \dots + t[i']$$

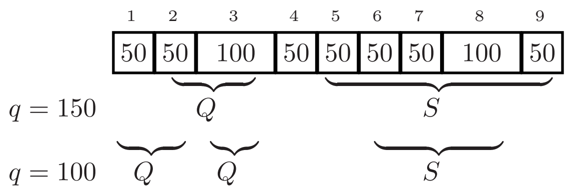


Figure 1. Q- and S-matching in musical sequences

As with Q, the match of S is said to be solid if $i = i'$.

For example, consider the musical sequence shown in Figure 1. For $q = 150$, Q matches with $t[2..3]$ and S matches with $t[5..9]$. For $q = 100$, Q matches with $t[1..2]$, $t[3]$ etc. and S matches with $t[6..8]$. However, note that for $q = 100$, S does not match with $t[7..9]$ despite the fact that $\sum_{i=7}^{9} t[i] = 2q$.

Consequently, a rhythm $r = r[1] \ldots r[m]$ is said to *match* with the substring $t[i..i']$ of the musical sequence t, if and only if there exists an integer $q \in \mathbb{N}^+$, and integers $i_1 < i_2 < \ldots < i_m < i_{m+1}$ such that

1. $i_1 = i$, $i_{m+1} = i' + 1$, and
2. $r[j]$ matches $t[i_j..i_{j+1} - 1]$, for all $1 \leq j \leq m$

For instance, the rhythm $r = QSS$ matches with $t[2..5]$ as well as with $t[5..8]$, in Figure 2, for $q = 50$.

Finally, a rhythm r is said to *cover* the substring $t[i..i']$ of the musical sequence t, if and only if there exist integers $i_1, i'_1, i_2, i'_2, \ldots, i_k, i'_k$, for some $k \geq 1$, such that

– r matches $t[i_\ell..i'_\ell]$, for all $1 \leq \ell \leq k$, and
– $i'_{\ell-1} \geq i_\ell - 1$, for all $2 \leq \ell \leq k$

In our example, Figure 2, $r = QSS$ covers $t[2..8]$ for $q = 50$.

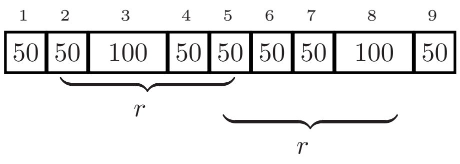


Figure 2. Matches of $r = QSS$ in t, for $q = 50$

3 Maximal Coverability Algorithm

In this section, we tackle the *maximal coverability* problem, which is formally defined as follows:

Problem 1. Given a musical sequence $t = t[1]t[2] \ldots t[n]$, $t[i] \in \mathbb{N}^+$, and a rhythm $r = r[1]r[2] \ldots r[m]$, $r[j] \in \{Q, S\}$, find the largest (longest) substring $t[i..i']$ of t that is covered by r.

Note that the definition above is very general, allowing extreme cases like the following: consider a musical sequence consisting of a single tone repeated every 1ms, $t = 111 \ldots 1$. Consider also a rhythm r consisting of Q's and S's. Then r will match t in every position i regardless of the value of q, since any Q in r will match with a

Algorithm 6 Stage 1: Computing vectors $first$ and $next$

```
function FindOccurrences(t[1..n])
    first[1..|Σ|] ← 00...0
    next[1..n] ← 00...0
    last[1..|Σ|] ← 00...0        ▷ Keeps track of the last occurrence of a particular σ ∈ Σ so far
    for i ← 1 to n do
        if last[t[i]] = 0 then
            first[t[i]] ← i
        else
            next[last[t[i]]] ← i
        last[t[i]] ← i
    return first, next
```

sequence of q 1's, and any S in r will match with a sequence of $2q$ 1's. To avoid such cases, we introduce the following restriction for the matching of a rhythm r with a substring $t[i..i']$ of t:

Restriction 1. *For each match of r with a substring $t[i..i']$, there must exist at least one S in r whose match in $t[i..i']$ is solid; that is, there exists at least one $1 \le j \le m$ such that $r[j] = t[k] = 2q$, $i \le k \le i'$, for some value of q.*

As explained before, the value of q is not *a priori* given. Therefore each $\sigma \in \Sigma$ should be considered as a candidate q, provided of course that $2\sigma \in \Sigma$, and for that particular q all the occurrences of the rhythm r must be identified. Equivalently, we can consider each σ to be equal to $S = 2q$, provided that $\sigma/2 \in \Sigma$. In our algorithm, we will be using the latter form. Then, for each such $\sigma \in \Sigma$, the algorithm sets $S = 2q = \sigma$ and proceeds in three stages:

- *Stage 1*: Find all occurrences of S in t.
- *Stage 2*: Transform the areas around all the S's into a sequences of Q's.
- *Stage 3*: Find the maximal area covered by r, for the current q.

We next explain each of these stages in detail.

3.1 Stage 1 – Finding all occurrences

In this stage, we need to find all occurrences of $S = \sigma$, for the chosen σ, so that we can (in Stage 2) transform the areas around each of those occurrences to sequences of Q's. A single scan through the input string suffices to find all occurrences of σ. Since the stage is repeated for every distinct $\sigma \in \Sigma$, overall the algorithm would need $O(|\Sigma|n)$ time on this stage alone.

However, it is easy to speedup this stage, by collectively computing linked lists of the occurrences of all the symbols. Given that the alphabet Σ is indexed and its size is bounded, this can be done in $O(n)$ time and $O(n + |\Sigma|)$ space in the following manner. Consider vectors $first$, of size $|\Sigma|$, and $next$, of size n, such that

- $first[\sigma] = i$ if and only if the first occurrence of the symbol σ appears at position i
- $next[i] = j$ if and only if $t[i] = t[j]$ and for all k, $i < k < j$, $t[k] \neq t[i]$; if no such j exists, then $next[i] = 0$

A single scan through t suffices to compute vectors $first$ and $next$. Algorithm 6 shows how this is done in detail.

3.2 Stage 2 – Transformation

The task of this stage is to transform t, which is a sequence of integers, into a sequence t', over $\{Q, S\}$ for the chosen $q = \sigma/2$, so that all the matches of r into t' (and consequently, into t) are identified. However, this transformation is ambiguous, in several ways, as the following example demonstrates.

Consider the musical sequence shown in Figure 3(a), and let $q = 50$. One does not know whether two consecutive Q's should be transformed as QQ or S, and creating all the possible combinations is too time consuming. Moreover, as shown in Figure 3(b) the transformation that is generated while processing t from left to right is different from that generated while moving from right to left.

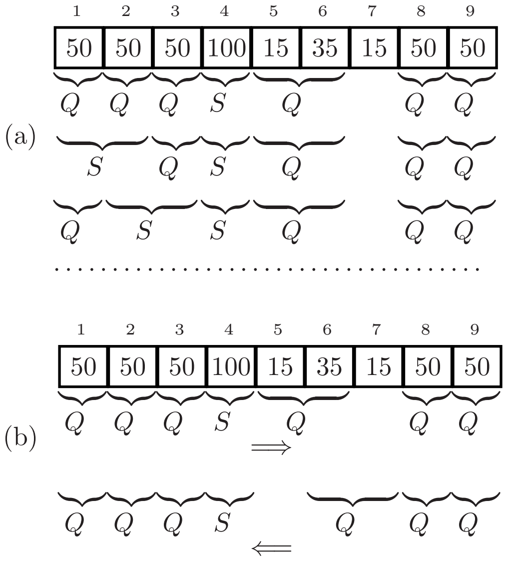


Figure 3. Ambiguities in transformation

For each occurrence of the current symbol $\sigma = 2q = S$, we convert the area surrounding that S into sequences of Q's. Algorithm 7 gives the details.

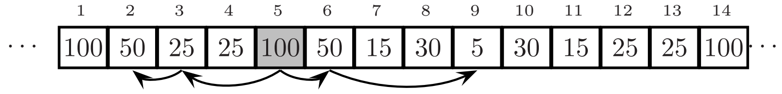


Figure 4. Transforming the area around $t[5] = S = 100$

3.3 Stage 3 – Maximal Covered Area

As soon as we get t', a sequence over $\{Q, S\}$, transformed from t in Stage 2, our job is to first identify all the occurrences of r in t'. To do that efficiently we exploit a bit-masking technique as described below. We first define some notations that we use for sake of convenience. We define $S_{t'}$ and S_r to indicate an S in t' and r respectively. $Q_{t'}$ and Q_r are defined analogously. We first perform a preprocessing as follows. We construct t'' from t' where each $S_{t'}$ is replaced by 01 and each $Q_{t'}$ is replaced by 1. Note that we have to keep track of the corresponding positions of t' in t''. We then construct the 'Invalid' set I for t'' where I includes each position of '1' of $S_{t'}$ in t''.

Algorithm 7 Stage 2: Transformation

```
function TRANSFORM(t[1..n],σ)
    q ← σ/2
    R_σ ← {}
    i ← first[σ]
    while i ≠ 0 do
        x ← “S”
        r ← 0
        j ← i
        while r < q and j < n do
            j ← j + 1
            r ← r + t[j]
            if r = q then
                Push Q at the back of x
                r ← 0
        r ← 0
        j ← i
        while r < q and j > 1 do
            j ← j − 1
            r ← r + t[j]
            if r = q then
                Push Q at the front of x
                r ← 0
        R_σ ← R_σ ∪ {x}
        i ← next[i]
    return R_σ
```

For example, if $t' = QQSQS$ then $t'' = 1101101$ and I = 4,7. It is easy to see that no occurrence of r can start at $i \in I$. We also construct r' from r where each S_r is replaced by 10 and each Q_r is replaced by 0. This completes the preprocessing. After the preprocessing is done, at each position $i \notin I$ of t'' we perform a bitwise ‘or’ operation between $t''[i..i + |r'| - 1]$ and r'. If the result of the ‘or’ operation is all 1’s then we report an occurrence at position i of t''. The details are formally given in the form of Algorithm 8.

We now discuss the correctness of Algorithm 8. We use the symbol $\sim$ and $\nsim$ to denote, respectively “matches” and “doesn’t match”. It is easy to see that for the problem in hand we must meet the following conditions.

1. $Q_{t'} \sim Q_r$
2. $Q_{t'}Q_{t'} \sim S_r$
3. $S_{t'} \sim S_r$
4. $S_{t'} \nsim Q_rQ_r$

All the conditions stated above are obeyed by the encoding we use as shown below. Recall that we do bitwise or operation and that we report a match when the result of the operation is all 1’s.

1. $Q_{t'}(= 1)$ and $Q_r(= 0)$ always matches: $(1\ or\ 0 = 1)$.
2. $Q_{t'}Q_{t'}(= 11)$ always matches with $S_r(= 10)$: $(11\ or\ 10 = 11)$.
3. $S_{t'}(= 01)$ can only match with $S_r(= 10)$: $(01\ or\ 10 = 11)$.
4. Since $S_{t'}(= 01)$ can’t give a match with $Q_rQ_r(= 00)$: $(01\ or\ 00 = 01)$.

However we have a problem when the S_r and $S_{t'}$ are ‘miss-aligned’. We define $start(S_r) = 1$ and $end(S_r) = 0$. Similarly, we have, $start(S_{t'}) = 0$ and $end(S_{t'}) = 1$. Assume that we have an $S_r(say\ S_r^k)$ miss-aligned with an $S_{t'}(say\ S_{t'}^l)$.

Algorithm 8 Reporting Occurrences of r in t'

```
function FindMatch(t′,r)
    Occ[1..|t′|] ← 0 0 . . . 0                          ▷ Preprocessing Step
    I[1..|t″|] ← 0 0 . . . 0
    j = 1
    for i = 1 to t′ do
        track[j] = i
        if t′[i] = “S” then
            t″[j] = “01”
            I[j + 1] = 1                                 ▷ Position j + 1 is invalid
            j = j + 2
        else
            t″[j] = “1”
            j = j + 1
    j = 1
    for i = 1 to r do
        if r[i] = “S” then
            r′[j] = “10”
            j = j + 2
        else
            r′[j] = “0”
            j = j + 1
                                                         ▷ Matching Step
    for i = 1 to t″ do
        if I[i] ≠ 1 then
            if t′[i..i + m1 − 1] or p′ = “11 . . . 1” then
                Occ[track[i]] = 1
    return Occ
```

Case 1- $end(S_r^k)$ is aligned with $start(S_{t'}^l)$: We have $end(S_r^k)$ or $start(S_{t'}^l)$ $0\ or\ 0 = 0$. So we have no match as required.

Case 2- $start(S_r^k)$ is aligned with $end(S_{t'}^l)$: Unfortunately here we have $start(S_r^k)$ or $end(S_{t'}^l) = 1\ or\ 1 = 1$ which may create problems. We distinguish between two subcases. We say an S_r is 'inside' r (or equivalently r') if this S_r is not the start of r.

Case2.a- S_r^k is inside r: There must be either a Q_r or another S_r (say S_r^j) just before this S_r^k. In any case we will have either $Q_r(= 0)$ or $end(S_r^j)(= 0)$ to align with $start(S_{t'}^l)(= 0)$ which will give 0 after the or operation and hence we have no problem.

Case2.b- S_r^k is the start of r: In this case we have $start(S_r^k)\ or\ end(S_{t'}^j) = 1$ which may give us a 'false positive' starting at this position. To exclude these false positives we have the 'Invalid' set $\mathcal{I}$. The main idea is that no occurrence of the rhythm can start at $end(S_{t'})$. So each $end(S_{t'})$ is included in $\mathcal{I}$. And we check whether the position we are checking is in $\mathcal{I}$ or not.

Here we give an example of a 'false positive' as discussed above. Suppose $t' = QQSQQ$ and $r = SQ$. Then we have $t'' = 110111$ and $r' = 100$. It is easy to see that if we perform the bitwise or operation at each position of t'' we get two matches starting at $t''[3]$ and also at $t''[4]$. But it is easy to verify that position 4 of t'' doesn't really exist in t'. So its a 'false positive'.

The above discussion establishes the correctness of Algorithm 8. Since the size of the rhythm is considered constant, Algorithm 8 runs in $O(|t''|/w)$ time where w is the size of the word of the target machine. Finally, once we get the occurrences of the rhythm r in t' considering every choice of σ it is easy to report the maximal covered area in linear time. In fact we can compute this area on the fly while computing all the occurrences of r in t' by slightly modifying Algorithm 8.

4 Open Problems

In this paper we have presented algorithms for computerized song classifications under some specific constraints. A number of issues remain unsolved as follows:

1. Designing an algorithm that avoids the restriction that one symbol has to be *solid*.
2. Applying a limit on the number of "additions" in the numeric text to match a Q and/or S.
3. Removing the dependency on m from the algorithm.

References

1. A. R. Brinkman: *PASCAL Programming for Music Research*, The University of Chicago Press, Chicago and London, 1990.
2. D. Byrd and E. Isaacson: *A music representation requirement specification for academia.* The Computer Music Journal, 27(4) 2003, pp. 43–57.
3. T. Crawford, C. Iliopoulos, and R. Raman: *String matching techniques for musical similarity and melody recognition.* Computing in Musicology, 11 1998, pp. 227–236.
4. P. Howell, R. West, and I. Cross, eds., *Representing Musical Structure*, Academic Press London, 1991.
5. C. S. Iliopoulos, K. Lemstrom, M. Niyad, and Y. J. Pinzon: *Evolution of musical motifs in polyphonic passages*, in Symposium on AI and Creativity in Arts and Science, Proceedings of AISB'02, G. Wiggins, ed., 2002, pp. 67–76.
6. K. Lemstrom: *String matching techniques for music re trieval.* PhD Thesis, University of Helsinki, Department of Computer Science, 2000.
7. K. Lemstrom and P. Laine: *Musical information retrieval using musical parameters*, in International Computer Music Conference, 1998, pp. 341–348.
8. K. Lemstrom and J. Tarhio: *Detecting monophonic patterns within polyphonic sources*, in Multimedia Information Access Conference, vol. 2, 2000, pp. 1261–1279.
9. A. Marsden and A. Pople, eds., *Computer Representations and Models in Music*, Academic Press London, 1992.
10. M. Mongeau and D. Sankoff: *Comparison of musical sequences.* Computers and the Humanities, 24 1990, pp. 161–175.
11. E. Selfridge-Field, ed., *Beyond MIDI: The Handbook of Musical Codes*, The MIT Press, 1997.
12. D. Stech: *A computerassisted approach to micro analysis of melodic lines.* Computers and the Humanities, 15 1981, pp. 211–221.
13. G. A. Wiggins, E. Miranda, A. Smaill, and M. Harris: *A framework for the evaluation of music representation systems.* The Computer Music Journal, 17(3) 1993, pp. 31–42.

On Some Combinatorial Problems Concerning the Harmonic Structure of Musical Chord Sequences

Domenico Cantone, Salvatore Cristofaro, and Simone Faro

Università di Catania, Dipartimento di Matematica e Informatica
Viale Andrea Doria 6, I-95125 Catania, Italy
{cantone | cristofaro | faro}@dmi.unict.it

Abstract. We present some combinatorial problems which arise in the fields of music representation and music processing, especially in contexts such as the analysis of the harmonic structure of chord sequences. We concern ourselves with those chord sequences which exhibit a certain kind of *regular harmonic structure*, and discuss some problems related to them. We provide also algorithms to solve some of these problems.

Keywords: music processing, harmonic structure analysis, chord sequences.

1 Introduction

Musical chord sequences, or chord progressions, possess a combinatorial structure very rich and complex, which require efficient computational methods to be fully understood and analyzed.

By using a convenient symbolic representation of musical notes and chords, it is possible to apply suitable mathematical methods to discover that kind of regularity in the harmonic structure which many chord sequences seem to exhibit [7,8].

Musical notes can be coded in various ways. A typical example is provided by the standard MIDI representation, where notes are coded by integers [9]. Once a particular coding of the notes is fixed, a chord can be conveniently represented by the collection of the symbols corresponding to the notes in the chord. Notice that in codings like the standard MIDI representation, notes that differ by one or more octaves are represented by distinct symbols. Such kind of codings are especially appropriate in the context of Music Information Retrieval, where the representation of (monophonic) musical sequences by strings of integers gives the possibility of applying powerful string matching techniques to discover musical pattern repetitions and melodic similarity [2,4,5,3].

However, in many cases, especially when one is interested in the interval content of chords, it is more convenient to assume octave equivalence of notes, i.e., to regard as equal any two notes which are one or more octaves apart [6]. In such a case, only 12 symbols are needed to represent notes, at least in the equal temperament system of western music. For example, let us consider the two simple chord sequences S_1 and S_2 represented in Figure 1. If we assume octave equivalence, and use the traditional naming of notes with the symbols $\mathsf{C}, \mathsf{C}\sharp, \mathsf{D}, \mathsf{D}\sharp, \mathsf{E}, \mathsf{F}, \mathsf{F}\sharp, \mathsf{G}, \mathsf{G}\sharp, \mathsf{A}, \mathsf{A}\sharp, \mathsf{B}$, as shown in Figure 2, then we may represent both sequences as the following list of sets

$$\{\mathsf{C}, \mathsf{E}, \mathsf{G}\}, \{\mathsf{C}, \mathsf{E}, \mathsf{A}\}, \{\mathsf{C}, \mathsf{F}, \mathsf{A}\}, \{\mathsf{D}, \mathsf{F}, \mathsf{A}\}, \{\mathsf{D}, \mathsf{F}, \mathsf{B}\}, \{\mathsf{D}, \mathsf{G}, \mathsf{B}\}, \{\mathsf{E}, \mathsf{G}, \mathsf{B}\}\,.$$

In fact, the corresponding chords of the sequences are made up of the same notes but in different octaves, i.e., they differ only in the voicings. Thus the two chord sequences

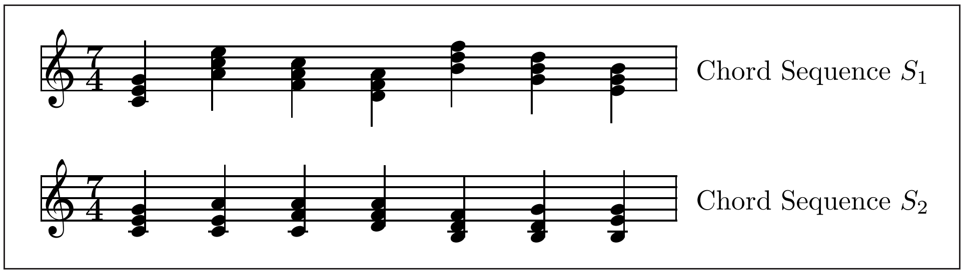


Figure 1. Two chord sequences related by octave equivalence. Chord sequence S_1 exhibits a regular harmonic structure.

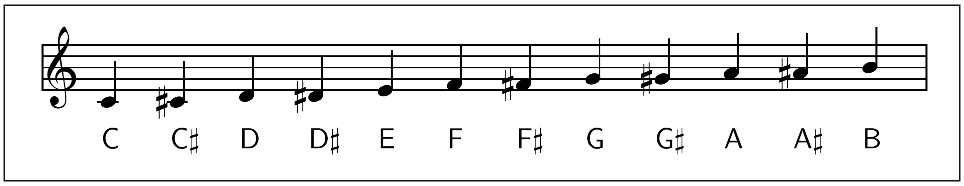


Figure 2. The traditional naming of notes with symbols $\mathsf{C}, \mathsf{C}\sharp, \mathsf{D}, \mathsf{D}\sharp, \mathsf{E}, \mathsf{F}, \mathsf{F}\sharp, \mathsf{G}, \mathsf{G}\sharp, \mathsf{A}, \mathsf{A}\sharp, \mathsf{B}$

can be really considered as two distinct variants, or voice leadings,[1] of a same chord sequence (assuming octave equivalence). However, if we regard the chords as ordered sets of notes, we may discover some regularities in the structure of the sequence S_1. Indeed, if we order the notes of each chord from the lowest to the highest one, i.e., if we look at the voicings of the chords, we get a representation of each chord as a string of symbols, and the chord sequence can be conveniently represented by a matrix $\mathcal{M}$ whose columns correspond to such strings:

$$\mathcal{M} = \begin{bmatrix} \mathsf{G\,E\,C\,A\,F\,D\,B} \\ \mathsf{E\,C\,A\,F\,D\,B\,G} \\ \mathsf{C\,A\,F\,D\,B\,G\,E} \end{bmatrix}.$$

A simple inspection of the matrix $\mathcal{M}$ reveals the regular structure of the chord sequence. Simply look at the secondary diagonal elements of each square submatrix of $\mathcal{M}$. Also, observe that any two consecutive chords of the sequence share at least two notes, and any three consecutive chords share at least one note, so that the chords are connected in such a way that the “transition” from a chord to a next one is gradually achieved by a series of smooth chord-passages: from a perceptual point of view, this translates into a pleasant sensation when the chord sequence is heard.

Notice also that if we “glue” at the left (or right) end of the matrix $\mathcal{M}$ a copy of itself, we get a matrix with the very same structure of $\mathcal{M}$. Musically speaking, this property can be interpreted by saying that the chord sequence has a kind of “circular” harmonic structure which, when heard, tends to resolve on itself, i.e., when the chord sequence is heard for the first time, one expects that it will be played

[1] Notice that in music, the usual meaning of voice leading concerns the horizontal motion of the notes, or voices, of the chords inside a chord sequence, where a chord sequence is regarded as the superimposition of two or more melodies played simultaneously. For us, a voice leading is simply a sequence of chord-voicings. But from a formal point of view, the two notions are equivalent, up to minor details (see Section 2 and [10]).

again. This is strictly related to the phenomenon of musical expectation, which plays a fundamental role in music composition. Thus, the ability of creating and discovering harmonic structures similar to that of the chord sequence S_1 in Figure 1 may have important applications in the fields of automatic music composition and automatic music analysis.

Chord sequences having a harmonic structure of the kind described above will be referred to as *regular chord progressions*.

In this paper we report some preliminary results concerning an ongoing investigation on various combinatorial questions about regular chord progressions and voicings.

1.1 Paper's organization

The paper is organized as follows. In Section 2 we introduce some basic notions and give a formal definition of regular chord progression. Subsequently, in Section 3 we present algorithms, based on the bit-parallelism technique, for some problems concerning combinatorial aspects of regular chord progressions, and also we discuss some other related questions. Then, in Section 4 we draw our conclusions. Finally, an appendix containing some theoretical results concludes the paper.

2 Basic definitions and properties

Before entering into details, we need a bit of notations and terminology. Let Σ be a finite alphabet. A string X of length $m \geq 0$ is represented as a finite array $X[0\,..\,m-1]$. For $m = 0$ we obtain the empty string ε. The length of X is denoted by $|X|$. By $X[i]$ we denote the $(i+1)$-th symbol of X, for $0 \leq i < |X|$. Likewise, by $X[i\,..\,j]$ we denote the substring of X contained between the $(i+1)$-th symbol and $(j+1)$-th symbol of X, for $0 \leq i \leq j < |X|$.

For convenience, we do not distinguish between a symbol s and the one-character string "s". Thus, for any two strings X and Y and any symbol s, we write $X_\bullet s_\bullet Y$ for the string Z of length $|X| + |Y| + 1$ such that

- $Z[0\,..\,|X| - 1] = X$,
- $Z[|X|] = s$, and
- $Z[|X| + 1\,..\,|X| + |Y|] = Y$.

A chord over Σ is a nonempty set C of two or more symbols of Σ. The size of a chord C, denoted by $size(C)$ or by $|C|$, is the number of symbols in C. A voicing over Σ is a string V of symbols of Σ such that $|V| \geq 2$ and $V[i] \neq V[j]$, for all distinct $i, j \in \{0, 1, \ldots, |V| - 1\}$. The base chord $Set(V)$ of a voicing V is the collection of the symbols occurring in V. A voicing V is said to be a voicing of a chord C if $Set(V) = C$.[2] A chord progression is a sequence $\mathcal{C} = \langle C_0, C_1, \ldots, C_n \rangle$ of chords with the same size.

A voice leading over Σ is a sequence $\mathcal{V} = \langle V_0, V_1, \ldots, V_n \rangle$ of voicings over Σ of the same length. A voice leading $\mathcal{V} = \langle V_0, V_1, \ldots, V_n \rangle$ is a voice leading of a chord progression $\mathcal{C} = \langle C_0, C_1, \ldots, C_m \rangle$, provided that $n = m$ and V_i is a voicing of the chord C_i, for $i = 0, 1, \ldots, n$.

Let V and W be voicings over the alphabet Σ. We say that V is (immediately) connected to W, and write $V \longrightarrow W$, if $W = s_\bullet V[0\,..\,|V| - 2]$, for some symbol

[2] Thus, there are $m!$ distinct voicings of any chord C of size m.

$s \in \Sigma$. Plainly, when $V \longrightarrow W$, the voicings V and W must have the same length. A voice leading $\mathcal{V} = \langle V_0, V_1, \ldots, V_n \rangle$ is CONNECTED if $V_i \longrightarrow V_{i+1}$, for $i = 0, 1, \ldots, n-1$; $\mathcal{V}$ is CIRCULARLY CONNECTED if it is connected and in addition $V_n \longrightarrow V_0$.[3]

A voicing V is CONNECTABLE to a voicing W with respect to an alphabet Σ, in symbols $V \Longrightarrow W$, if there is a connected voice leading $\mathcal{V} = \langle V_0, V_1, \ldots, V_n \rangle$ over Σ, with $n \geq 1$, such that $V_0 = V$ and $V_n = W$. In such a case, we say that the voice leading $\mathcal{V}$ CONNECTS V to W (with respect to Σ).

The *connectivity relation* "$\Longrightarrow$" is an equivalence relation, as shown in the following lemma.

Lemma 1. *Let V, W, and Z be voicings over an alphabet Σ. Then*

(1) $V \Longrightarrow V$;
(2) if $V \longrightarrow W$ then $W \Longrightarrow V$;
(3) if $V \Longrightarrow W$ then $W \Longrightarrow V$;
(4) if $V \Longrightarrow W$ and $W \Longrightarrow Z$, then $V \Longrightarrow Z$.

Therefore the relation $\Longrightarrow$ is an equivalence relation.

Proof. We give only the proof of (1) and (2), since (4) is an immediate consequence of the definition of the relation "$\Longrightarrow$" and (3) follows from (2) and (4).

Let V, W, and Z be voicings of the same length m, over the alphabet Σ.

Concerning (1), it can easily be verified that V is connected to V by the voice leading $\langle V_0, V_1, \ldots, V_m \rangle$, where $V_0 = V$ and

$$V_{i+1} =_{\text{Def}} V_i[m-1]_\bullet V_i[0 \mathinner{..} m-2]\,,$$

for $i = 0, 1, \ldots, m-1$.

Next, let $V \longrightarrow W$. In order to verify (2), we distinguish two cases, according to whether $V[m-1] = W[0]$ or $V[m-1] \neq W[0]$. If $V[m-1] = W[0]$, then W is connected to V by the voice leading $\langle V_0, V_1, \ldots, V_{m-1} \rangle$, where $V_0 = W$ and

$$V_{i+1} =_{\text{Def}} V_i[m-1]_\bullet V_i[0 \mathinner{..} m-2]\,,$$

for $i = 0, 1, \ldots, m-2$.
On the other hand, if $V[m-1] \neq W[0]$, then W is connected to V by the voice leading $\langle V_0, V_1, \ldots, V_m \rangle$, where $V_0 = W$ and

$$V_{i+1} =_{\text{Def}} V[m-i-1]_\bullet V_i[0 \mathinner{..} m-2]\,,$$

for $i = 0, 1, \ldots, m-1$. □

A chord C is CONNECTED to a chord D, written $C \longrightarrow D$, if $V \longrightarrow W$, for some voicings V of C and W of D.[4] A chord progression $\mathcal{C}$ is CONNECTED (resp., CIRCULARLY CONNECTED) if it has a connected (resp., circularly connected) voice leading. A chord progression $\mathcal{C} = \langle \mathcal{C}_0, \mathcal{C}_1, \ldots, \mathcal{C}_n \rangle$ is REGULAR if it is circularly connected and, in addition, $\mathcal{C}_i \neq \mathcal{C}_{(i+1) \bmod (n+1)}$, for $i = 0, 1, \ldots, n$. It can easily be verified that the chord progression S_1 in Figure 1 is regular.

We conclude the section with some examples.

[3] Notice that if a voice leading $\langle V_0, V_1, \ldots, V_n \rangle$ is circularly connected, then $n \geq |V_0| - 1$.
[4] From the context it will always be clear whether the symbol "$\longrightarrow$" denotes the connectivity relation between voicings or between chords.

Example 2. Given the alphabet $\Sigma = \{a, b, c, d, e\}$, the following strings

$$V_1 = abcd\,, \quad V_2 = dabc\,, \quad V_3 = edab\,, \quad V_4 = edabc$$

are voicings over Σ. The voicings V_1, V_2, and V_3 have length 4, whereas V_4 has length 5. On the other hand, the strings

$$X = abca\,, \quad Y = deece\,, \quad Z = abcdf$$

are not voicings over Σ. Indeed, $X[0] = X[3] = a$, $Y[1] = Y[2] = Y[4] = e$, and $Z[4] = f$ is not a symbol of Σ (however, Z is a voicing over the extended alphabet $\Sigma \cup \{f\}$). Moreover, the voice leading $\mathcal{V} = \langle V_1, V_2, V_3 \rangle$ is connected, so that the voicing V_1 is connectable to voicing V_3 with respect to Σ. □

Example 3. The following chord progression over the alphabet $\Sigma = \{a, b, c, d, e, f\}$

$$\mathcal{C} = \langle \{f, a, c\}, \{a, b, c\}, \{c, e, b\}, \{c, e, b\}, \{f, c, e\}, \{f, a, c\} \rangle$$

is circularly connected since it has the circularly connected voice leading

$$\mathcal{V} = \langle caf, bca, ebc, ceb, fce, afc \rangle\,.$$

However, $\mathcal{C}$ is not regular since the first and the last chord coincides.
Notice that the above voice leading $\mathcal{V}$ can also be represented by the matrix

$$\mathcal{M} = \begin{bmatrix} f & a & c & b & e & c \\ a & c & b & e & c & f \\ c & b & e & c & f & a \end{bmatrix}\,,$$

whose columns correspond, from left to right, to the voicings of $\mathcal{V}$, oriented from bottom to top (as are the notes in the staff). Observe that the secondary diagonal elements in each square submatrix of $\mathcal{M}$ are equal. □

Example 4. The chord progression

$$\langle \{a, b, c\}, \{a, b, f\}, \{a, d, f\}, \{b, d, f\} \rangle$$

over the alphabet $\Sigma = \{a, b, c, d, e, f\}$ is connected but not circularly connected, as can be easily verified by trying out all of its possible connected voice leadings. □

3 Discovering regular structures: some algorithms

In this section we discuss some problems concerning combinatorial properties of regular chord progressions, and provide also algorithms to solve them.
We begin by addressing the following question.

Problem 5. Given a chord progression $\mathcal{C} = \langle C_0, C_1, \ldots, C_n \rangle$ over an alphabet Σ, a voicing V of C_0, and a voicing W of C_n, construct, if it exists, a voice leading of $\mathcal{C}$ connecting V to W, i.e., a connected voice leading $\mathcal{V} = \langle V_0, V_1, \ldots, V_n \rangle$ such that $V_0 = V$, $V_n = W$, and $Set(V_i) = C_i$, for $i = 0, 1, \ldots, n$. □

We can solve Problem 5 as follows. Let m be the length of V. We show how to construct the desired connected voice leading $\mathcal{V}$ of $\mathcal{C}$, or determine that such a voice leading does not exist, by a sequence of $n+1$ stages. We start by setting $V_0 = V$ (this is the initial stage 0). Next, let us suppose that at the end of stage i we have constructed a connected voice leading $\mathcal{V}_i = \langle V_0, V_1, \ldots, V_i \rangle$ of $\mathcal{C}_i = \langle C_0, C_1, \ldots, C_i \rangle$, with $0 \leq i < n$. Then, we form the set $S_i =_{\text{Def}} Set(V_i[0\,..\,m-2])$ and check whether $S_i \subseteq C_{i+1}$. If this is the case, we prolongate the voice leading $\mathcal{V}_i$ with the new voicing V_{i+1} defined by

$$V_{i+1} =_{\text{Def}} c_\bullet V_i[0\,..\,m-2]\,,$$

where $c \in C_{i+1} \setminus S_i$, and proceed to the next stage. Otherwise, we stop the process and announce that there is no connected voice leading of $\mathcal{C}$ from V to W. If all stages are completed successfully and, in addition, the last voicing of $\mathcal{V}_n$ equals W, then it is immediate to check that $\mathcal{V}_n$ is a voice leading of $\mathcal{C}$ which connects V to W. The correctness of the above procedure follows immediately from the observation that if a voice leading of $\mathcal{C}$ connecting V to W does exist, then it is unique.

In Figure 3 we show the pseudo-code of an algorithm, named ALGO1, which implements the above construction process.

Remark 6. Notice that during the execution of the algorithm ALGO1, the string-variable X contains the voicings V_i, which form a connected voice leading of $\mathcal{C}$ from V to W, provided that it exist. Therefore, if immediately after line 8 of ALGO1 we add an instruction **OUTPUT**(X), we get as by-product the sequence $V_1, V_2, \ldots, V_k$, where k is the largest index less than or equal to n such that the chord progression $\mathcal{C}_k = \langle C_0, C_1, \ldots, C_k \rangle$ has a connected voice leading starting at V. In fact, $\mathcal{V}_k = \langle V, V_1, \ldots, V_k \rangle$ turns out to be a voice leading of $\mathcal{C}_k$. □

Concerning the complexity of the algorithm ALGO1, we notice that in the worst case we need to compute all the partial voice leadings $\mathcal{V}_0, \mathcal{V}_1, \ldots, \mathcal{V}_n$; thus the total time spent in the whole process is $\mathcal{O}(n \cdot f(m))$, where $f(m)$ is an upper bound to the time needed to check whether $S_i \subseteq C_{i+1}$ and to construct the voicing V_{i+1}, for $i = 0, 1, \ldots, n-1$.

If we represent sets by linear arrays, we get $f(m) = \mathcal{O}(m^2)$, yielding an overall running time of $\mathcal{O}(nm^2)$.

However, if the alphabet Σ is sufficiently small to fit into a computer word, we can conveniently use the bit-parallelism technique [1] to reduce the running time to $\mathcal{O}(n+m)$. Indeed, let us assume that $\sigma = |\Sigma| \leq \omega$, where ω is the number of bits in a computer word. Then, any subset of Σ can be represented by a bit mask of length σ, which fits into a computer word. By using such a representation, the set operations of union, intersection, and complement, as well as the set containment test, can be executed in constant time by suitable combinations of the bitwise operations "OR", "AND", and "NOT" (denoted by the symbols "$\vee$", "$\wedge$", and "$\sim$", respectively).

More precisely, after fixing an (arbitrary) ordering

$$s_0, s_1, \ldots, s_{\sigma-1}$$

of the symbols of Σ, we use the following representations:

- a singleton $\{s_i\} \subseteq \Sigma$ is represented as the bit mask $\mathsf{B}(s_i) = b_0 b_1 \cdots b_{\sigma-1}$ (of length σ), where

$$b_j = \begin{cases} \mathsf{1} & \text{if } j = \sigma - 1 - i \\ \mathsf{0} & \text{otherwise}\,, \end{cases}$$

```
ALGO1(𝓒, V, W)
–It is assumed that 𝓒 is a chord progression, V is a voicing of
–the first chord of 𝓒, and W is a voicing of the last chord of 𝓒.
m := |V|
n := length(𝓒) − 1
S := {V[0], ..., V[m − 2]}
X := V
for i := 1 to n do
    if S ⊆ 𝓒[i] then
        - let z be such that 𝓒[i] = S ∪ {z}
        X := z•X[0 .. m − 2]
        S := (S \ {X[m − 2]}) ∪ {z}
    else
        return false
if X ≠ W then
    return false
return true
```

Figure 3. Pseudo-code of the algorithm ALGO1 for determining whether a chord progression $\mathcal{C}$ has a voice leading which connects a voicing V to a voicing W

for $j = 0, 1, \ldots, \sigma - 1$;[5]

- a nonempty subset $A = \{s_{i_0}, s_{i_1}, \ldots, s_{i_k}\}$ of Σ is represented as the bit mask

$$\mathsf{B}(A) =_{\text{Def}} \mathsf{B}(s_{i_0}) \vee \mathsf{B}(s_{i_1}) \vee \cdots \vee \mathsf{B}(s_{i_k});$$

- the empty subset of Σ is represented by the bit mask $\mathsf{0}^\sigma$, i.e., the string consisting of σ copies of the bit $\mathsf{0}$;
- a chord progression $\mathcal{C} = \langle C_0, C_1, \ldots, C_n\rangle$ is represented as an array $\boldsymbol{\mathcal{C}}[0 \,..\, n]$ of $n+1$ bit masks, where $\boldsymbol{\mathcal{C}}[i] = \mathsf{B}(C_i)$ for $i = 0, 1, \ldots, n$;
- a voicing V is represented as an array $\mathbf{V}[0 \,..\, m-1]$ of m bit masks, where $\mathbf{V}[i] = \mathsf{B}(V[i])$, for $i = 0, 1, \ldots, m-1$ (this amounts to represent a voicing $V = v_0 v_1 \cdots v_{m-1}$ as the ordered tuple of the bit masks corresponding to the singletons $\{v_0\}, \{v_1\}, \ldots, \{v_{m-1}\}$).

It is convenient to use a queue $\mathcal{Q}$ to store the first $m-1$ symbols (represented as singleton bit masks) of the voicings $V_0, V_1, \ldots, V_n$, as they are generated during the construction process. More precisely, at stage i of the construction, the queue $\mathcal{Q}$ will have the following configuration

$$\mathsf{B}(V_i[0]), \mathsf{B}(V_i[1]), \ldots, \mathsf{B}(V_i[m-2]),$$

with the head pointing to the rightmost bit mask, $\mathsf{B}(V_i[m-2])$, and the tail pointing to the leftmost one, $\mathsf{B}(V_i[0])$. Notice that there is no need to store the last symbol of voicing V_i, as the subsequent voicing V_{i+1} is completely determined by the partial voicing $V_i[0 \,..\, m-2]$ and by the chord C_{i+1}. The collection $S_i = \mathit{Set}(V_i[0 \,..\, m-2])$ can be conveniently maintained in a bit mask S, so that the test $S_i \subseteq C_{i+1}$ becomes $\mathsf{S} \wedge \boldsymbol{\mathcal{C}}[i+1] = \mathsf{S}$. Then the construction of the voicing V_{i+1} can be accomplished by the following sequence of steps:

[5] Notice that this amounts to representing the singleton $\{s_i\}$ by the "machine integer" $(1 \ll i)$, where "$\ll$" denotes the bitwise operation of left-shifting.

– retrieve the unique element z in $C_{i+1} \setminus S_i$, which will be the first symbol of V_{i+1}, by setting $\mathsf{Z} := \boldsymbol{\mathcal{C}}[i+1] \wedge \sim \mathsf{S}$ (plainly, Z contains the bit mask $\mathsf{B}(z)$);
– retrieve the first bit mask D in $\mathcal{Q}$ by executing the operation *dequeue*($\mathcal{Q}$);
– enqueue Z in $\mathcal{Q}$.

After these steps, $\mathcal{Q}$ will have the following configuration

$$\mathsf{Z}\,,\ \mathsf{B}(V_i[0])\,,\ \mathsf{B}(V_i[1])\,,\ \ldots\,,\ \mathsf{B}(V_i[m-3])$$

and $V_{i+1}[0\,..\,m-2]$ will be correctly stored in $\mathcal{Q}$. As a final step, S will be set to $(\mathsf{S} \wedge \sim \mathsf{D}) \vee \mathsf{Z}$, so as to represent the set S_{i+1}.

Remark 7. Since each *dequeue* operation on $\mathcal{Q}$ is always followed by an *enqueue* operation, the queue $\mathcal{Q}$ may be conveniently implemented as an array $\mathbf{Q}[0\,..\,m-2]$ of bit masks with a pointer h, which at stage i stores the partial voicing $V_i[0\,..\,m-2]$ into the array $\mathbf{Q}$ in a circular manner, starting at position h. Then a *dequeue*($\mathcal{Q}$) operation is just performed by retrieving the element $\mathbf{Q}[h]$ and the subsequent operation *enqueue*($\mathcal{Q}, \mathsf{Z}$) is simply performed by setting $\mathbf{Q}[h]$ to Z and then shifting circularly the pointer h one position to the right. □

The complete algorithm, named ALGO2, is presented in details in Figure 4. By inspection, it is immediate to see that ALGO2 has a $\mathcal{O}(n+m)$-running time.

Remark 8. Analogously to the observation in Remark 6 relative to the algorithm ALGO1, also algorithm ALGO2 can be adapted so as to produce as output the longest connected voice leading starting at V (of an initial segment) of $\mathcal{C}$, by using an additional string-variable X and adding the following lines of code between lines 11 and 12:

$X := \mathit{decode}(\mathsf{Z})$
for $j := 0$ **to** $m-2$ **do**
  $X := X_{\bullet}\mathit{decode}(\mathbf{Q}[(h+m-2-j) \mod (m-1)])$
OUTPUT(X)

The one-argument function *decode* yields the symbol s_i, when applied to the bit mask $\mathsf{B}(s_i)$ which represents the singleton $\{s_i\}$, for $s_i \in \Sigma$. The function *decode* admits a simple constant-time implementation. To begin with, let us represent the alphabet Σ as an array $\Sigma[0\,..\,\sigma-1]$, so that $\Sigma[i] = s_i$, for $i = 0, 1, \ldots, \sigma-1$. Then, if $x = \mathsf{B}(s_i)$, we have immediately $s_i = \Sigma[\lceil \log_2 x \rceil]$, for $i = 0, 1, \ldots, \sigma-1$. Therefore, we can just put $\mathit{decode}(x) =_{\mathrm{Def}} \Sigma[\lceil \log_2 x \rceil]$. If we further assume that the symbols of the alphabet Σ are the first σ nonnegative integers, i.e. $s_i = i$, for $0 \leq i \leq \sigma-1$, the decoding function becomes more simply $\mathit{decode}(x) =_{\mathrm{Def}} \lceil \log_2 x \rceil$. Additionally, under such an assumption, we have also that $\mathsf{B}(s) = (1 \ll s)$, where $\ll$ denotes the bitwise operation of left-shifting, implying that also the coding of a singleton $\{s\}$ as the bit mask $\mathsf{B}(s)$ can be performed in constant time, for any symbol $s \in \Sigma$.

Notice, however, that if we modify the ALGO2 algorithm so as to output a voice leading as described above, its running time increases to $\mathcal{O}(nm)$, provided that $\log_2 x$ can be computed in constant time. □

A second question we address is the following.

Problem 9. Given a chord progression $\mathcal{C} = \langle C_0, C_1, \ldots, C_n \rangle$, check whether $\mathcal{C}$ is regular. □

```
ALGO2(C, V, W)
m := length(V)
n := length(C) − 1
for h = m − 2 down to 0 do
      Q[h] := V[m − 2 − h]
S := 0^σ
for i := 0 to m − 2 do
      S := S ∨ V[i]
h := 0
for i := 1 to n do
      if (C[i] ∧ S) = S then
            Z := (C[i] ∧ ∼ S)
            D := Q[h]
            Q[h] := Z
            h := (h + 1) mod (m − 1)
            S := (S ∧ ∼ D) ∨ Z
      else
            return false
for j := 0 to m − 2 do
      if Q[(h + j) mod (m − 1)] ≠ W[m − 2 − j] then
            return false
return true
```

Figure 4. An optimized variant with bit-parallelism of the ALGO1 algorithm

To solve this problem, a natural but inefficient solution could be the following. Let m be the size of the chords $C_0, C_1, \ldots, C_n$. We start by checking that $C_i \neq C_{i+1}$, for $i = 0, 1, \ldots, n-1$. Then we form all possible voicings of the first chord C_0, and for each such voicing V we run the algorithm ALGO2 to search for a connected voice leading of $\mathcal{C}$ from V to the voicing $W = V[1 \,..\, m-1]\centerdot w$, where w is the only symbol of C_n not contained in C_0 (if, indeed, $|C_n \setminus C_0| \neq 1$, then, certainly, $\mathcal{C}$ would not be regular). Since there are $m!$ possible voicings of C_0, such an approach has a $\mathcal{O}(m!(n+m))$-time complexity.

However, we note some facts. First of all, given the two distinct chords C_i and C_{i+1}, we have that $C_i \longrightarrow C_{i+1}$ if and only if C_i and C_{i+1} share exactly $m-1$ symbols. Thus we can check easily if $C_0 \longrightarrow C_1 \longrightarrow \cdots \longrightarrow C_n \longrightarrow C_0$, which is a necessary condition for the chord progression $\mathcal{C}$ to be regular. Thence, if we find a pair of chords C_i and C_{i+1} such that $C_i \longrightarrow C_{i+1}$ does not hold, we conclude immediately that $\mathcal{C}$ is not regular. But we point out that the condition $C_0 \longrightarrow C_1 \longrightarrow \cdots \longrightarrow C_n \longrightarrow C_0$ is not, in general, a sufficient condition for $\mathcal{C}$ to be regular. For instance, let us consider the chords $C' = \{a, b, c\}$, $C'' = \{a, b, x\}$ and $C''' = \{a, b, d\}$. Although $C' \longrightarrow C'' \longrightarrow C''' \longrightarrow C'$ and $C' \neq C'' \neq C''' \neq C'$, the chord progression $\langle C', C'', C''' \rangle$ is not regular.

We observe, however, that if $\mathcal{C} = \langle C_0, C_1, \ldots, C_n \rangle$ is regular, and we set

$$X_k = \bigcap_{i=0}^{m-1-k} C_i \,, \quad \text{for } k = 0, 1, \ldots, m-1, [6]$$

where $X_{m-1} = C_0$, then each of the $m-1$ sets $X_0, X_1, \ldots, X_{m-1}$, except the last one, must be a nonempty proper subset of the set which immediately follows it; i.e., there

[6] Notice that $\mathcal{C}$ cannot be regular unless $n \geq m-1$.

```
ALGO3(𝓒, m)
n := length(𝓒) − 1
for i := 0 to n − 1 do
    if 𝓒[i] = 𝓒[i + 1] then
        return false
X_{m−1} := 𝓒[0]
for k := m − 2 down to 0 do
    X_k := X_{k+1} ∩ 𝓒[m − k − 1]
    if |X_{k+1} \ X_k| = 1 then
        - let z be such that X_{k+1} = X_k ∪ {z}
        V[k + 1] := W[k] := z
    else
        return false
- let c be such that X_0 = {c}
V[0] := c
if c ∉ 𝓒[n] and |𝓒[n] ∩ 𝓒[0]| = m − 1 then
    - let w be such that 𝓒[n] \ 𝓒[0] = {w}
    W[m − 1] := w
    return ALGO1(𝓒, V, W)
else
    return false
```

Figure 5. The algorithm ALGO3 checks if a given chord progression $\mathcal{C}$ is regular. The size of the chords in $\mathcal{C}$ is m.

must be $m-1$ distinct symbols $c_0, c_1, \ldots, c_{m-2}$ of C_0 such that

$$X_0 = \{c_0\}\,, \quad X_1 = \{c_0, c_1\}\,, \quad \ldots\,, \quad X_{m-2} = \{c_0, c_1, \ldots, c_{m-2}\}.$$

Additionally, if c_{m-1} is the symbol of C_0 distinct from $c_0, c_1, \ldots, c_{m-2}$, then a circularly connected voice leading $\mathcal{V} = \langle V_0, V_1, \ldots, V_n \rangle$ of $\mathcal{C}$ must begin necessarily with the voicing $c_0 c_1 \cdots c_{m-2} c_{m-1}$; i.e. $V_0[i] = c_i$, for $i = 0, 1, \ldots, m-1$. These considerations are indeed immediate consequences of Theorem 19 in Appendix A. In addition, we must also have that $C_n = \{c_1, \ldots, c_{m-1}, w\}$, for some symbol w distinct from c_0, because $C_n \neq C_0$ and $V_n \longrightarrow V_0$, with V_n a voicing of C_n.

Then, given the chord progression $\mathcal{C}$, in order to check whether $\mathcal{C}$ is regular, we proceed as follows. We begin by forming the sets $X_0, X_1, \ldots, X_{m-1}$, and check whether $|X_{k+1} \setminus X_k| = 1$, for $k = 0, 1, \ldots, m-2$. If this is not the case, we conclude immediately that $\mathcal{C}$ is not regular. Otherwise, we extract the symbols $c_0, c_1, \ldots, c_{m-1}$ such that $c_{k+1} \in X_{k+1} \setminus X_k$, for $0 \leq k \leq m-2$, and $c_0 \in X_0$, and then check whether $C_n = \{c_1, \ldots, c_{m-1}, w\}$, for some symbol w distinct from c_0. If this is not the case, we conclude again that $\mathcal{C}$ is not regular; otherwise we form the voicings $V = c_0 c_1 \cdots c_{m-1}$ and $W = c_1 c_2 \cdots c_{m-1} w$, and run the algorithm ALGO1 with inputs $\mathcal{C}$, V, and W to search for a connected voice leading of $\mathcal{C}$ from V to W. The resulting algorithm, named ALGO3, is presented in Figure 5. Plainly, the time complexity of ALGO3 is $\mathcal{O}(nm^2)$ (at least in case in which sets are represented as linear arrays.)

However, by using the bit-parallelism technique, thus representing as usual sets as bit masks, and voicings as arrays of bit masks, we can obtain an efficient variant of the ALGO3 algorithm, called ALGO4. The algorithm ALGO4, shown in Figure 6, uses the algorithm ALGO2 (the variant of ALGO1 based on bit-parallelism) as a subroutine. It assumes that the input chord progression $\mathcal{C}$ is given as an array $\boldsymbol{\mathcal{C}}$ of bit masks representing the chords in $\mathcal{C}$. By a simple inspection, it is easy to see that

```
ALGO4(𝓒, m)
n := length(𝓒) − 1
for i := 0 to n − 1 do
      if 𝓒[i] = 𝓒[i + 1] then
            return false
X := 𝓒[0]
for k := m − 2 down to 0 do
      Y := X ∧ 𝓒[m − k − 1]
      if Y ≠ 0^σ and Y ≠ X then
            V[k + 1] := W[k] := X∧ ∼ Y
            X := Y
      else
            return false
V[0] := X
if X ∧ 𝓒[n] = 0^σ and (𝓒[n] ∧ 𝓒[0]) ∨ X = 𝓒[0] then
      W[m − 1] := 𝓒[n]∧ ∼ 𝓒[0]
      return ALGO2(𝓒, V, W)
else
      return false
```

Figure 6. An optimized variant with bit-parallelism of the ALGO3 algorithm

the ALGO4 algorithm has an $\mathcal{O}(n+m)$-running time and requires only $\mathcal{O}(m)$-extra space.

3.1 Further questions on the connectivity of chords and voicings

We discuss next a few further questions concerning the connectivity of chords and voicings. We begin by observing that

Property 10. Any two chords of the same size can always be connected by a voice leading. □

Indeed, let C' and C'' be two chords of size m, and let V_0 be any voicing of C'. We define a connected voice leading $\mathcal{V} = \langle V_0, V_1, \ldots, V_m \rangle$, in such a way that

- $V_{i+1}[1 \mathinner{..} m-1] = V_i[0 \mathinner{..} m-2]$, and
- $V_{i+1}[0]$ is any symbol in $C'' \setminus Set(V_{i+1}[1 \mathinner{..} m-1])$,

for $i = 0, 1, \ldots, m-2$.

Since V_m is a voicing of C'', it follows that $\mathcal{V}$ is indeed a voice leading connecting C' to C''.

We observe that in the above construction the voicing V_0 of C' has been selected arbitrarily. Therefore, we can conclude that the following property holds too:

Property 11. Any given chord progression $\mathcal{C} = \langle C_0, C_1, \ldots, C_n \rangle$ can always be embedded into a *connected* chord progression $\mathcal{C}' = \langle C'_0, C'_1, \ldots, C'_p \rangle$, in the sense that $C_i = C'_{k_i}$, for some strictly increasing sequence of indices $0 \leq k_i \leq p$, for $i = 0, 1, \ldots, n$. □

An interesting problem is then the following:

Open Problem 12. Find a minimal connected chord progression which extends a given chord progression. □

The connectivity relation between voicings depends on the richness of the alphabet. For instance, let us consider the voicings $V = abcd$ and $W = abdc$ of the same chord $C = \{a, b, c, d\}$. If we try to connect V to W by using only symbols of the alphabet $\Sigma = \{a, b, c, d\}$, then we end up with the periodic voice leading

$$\mathcal{V} = \langle V_1, V_2, V_3, V_4,\ V_1, V_2, V_3, V_4,\ V_1, V_2, V_3, V_4, \ldots \rangle ,$$

where $V_1 = V = abcd$, $V_2 = dabc$, $V_3 = cdab$ and $V_4 = bcda$, proving that V can not be connected to W with respect to the alphabet Σ.

However, if we are allowed to use a new symbol, say x, then it is immediate to see that

$$\langle abcd, xabc, cxab, dcxa, bdcx, abdc \rangle$$

is a voice leading which connects V to W (with respect to the alphabet $\Sigma \cup \{x\}$).

An immediate consequence of Theorem 16 in Appendix A is the following connectability test for voicings:

> Given any two voicings V and W of the same length over an alphabet Σ, if $Set(V) \neq \Sigma$ or $Set(W) \neq \Sigma$, then V can be connected to W with respect to Σ, otherwise V can be connected to W if and only if W is a substring of $V_{\bullet}V$.

But despite the simplicity of the above test, the related optimization problem does not seem to possess a simple and efficient algorithmic solution:

Open Problem 13. Given two voicings V and W of the same length over an alphabet Σ, determine a shortest voice leading connecting V to W. □

Notice that Open Problems 12 and 13 above may have practical applications in various musical situations, as for instance in the case in which one wants to compose a chord progression by using certain fixed or preferred chords or chord-voicings, eventually interspacing them by some other chords, and the length of the chord progression is constrained so as to fit within a given maximum number of available bars.

Another interesting question related to the connectivity relation between voicings, with applications in music composition, is the following. When a composer is engaged in assembling a harmonic progression, sometimes he or she has at hand only a limited number of available tones to form the various chords of the progression; this is the case, for instance, when the notes must belong to a particular scale, such as a pentatonic scale, or a diatonic scale, or similar. In this case, the above cited Theorem 16 has the consequence that if V and W are voicings of two distinct chords, then no additional tone is required to connect V to W. However, the theorem does not say anything on the fact that a voice leading $\mathcal{V}$ which connects V to W have to satisfy the additional property that any two or more consecutive voicings of $\mathcal{V}$ must have distinct base chords.

The ability of creating harmonic progressions with a certain degree of "dissimilarity" between consecutive chords is an important issue in order for a harmonic progression not to result too monotonous or uninteresting. From Corollary 18 in Appendix A, it follows that two extra symbols suffice to allow any two voicings V and W of the same length to be connected by a voice leading $\mathcal{V} = \langle V_0, V_1, \ldots, V_n \rangle$ such that $Set(V_i) \neq Set(V_{i+1})$, for $i = 0, 1, \ldots, n-1$. This implies also that given a chord progression $\mathcal{C}$, we can always extend $\mathcal{C}$ to a regular chord progression by adding at most two new symbols. An interesting question is then the following:

Open Problem 14. Given a chord progression $\mathcal{C} = \langle C_0, C_1, \ldots C_n \rangle$ and a fixed bound $k > n$, determine the minimum number of new symbols we need to add in order that $\mathcal{C}$ can be extended to a regular chord progression of length at most k. $\square$

4 Conclusions

We have presented some combinatorial problems on strings which arise in the fields of music processing and music analysis. We have also provided algorithms to solve some of these problems, whereas some others have been raised but left unsolved (at least in the sense that no efficient algorithm has been provided). We plan to address in more details such problems in the future and to provide also efficient algorithmic solutions to them.

References

1. R. Baeza-Yates and G. H. Gonnet: *A new approach to text searching.* Communications of the ACM, 35(10) 1992, pp. 74–82.
2. E. Cambouropoulos, M. Crochemore, C. S. Iliopoulos, L. Mouchard, and Y. J. Pinzon: *Algorithms for computing approximate repetitions in musical sequences*, in Proc. of the 10th Australasian Workshop on Combinatorial Algorithms, R. Raman and J. Simpson, eds., Perth, WA, Australia, 1999, pp. 129–144.
3. T. Crawford, C. Iliopoulos, and R. Raman: *String matching techniques for musical similarity and melodic recognition.* Computing in Musicology, 11 1998, pp. 71–100.
4. M. Crochemore, C. S. Iliopoulos, T. Lecroq, and Y. J. Pinzon: *Approximate string matching in musical sequences*, in Proc. of the Prague Stringology Conference '01, M. Balík and M. Šimánek, eds., Prague, Czech Republic, Annual Report DC–2001–06, 2001, pp. 26–36.
5. M. Crochemore, C. S. Iliopoulos, Y. J. Pinzon, and G. Navarro: *A bit-parallel suffix automaton approach for (δ, γ)-matching in music retrieval*, in Proc. of the 10th International Symposium on String Processing and Information Retrieval (SPIRE'03), E. S. D. Moura and A. L. Oliveira, eds., vol. 2857 of Lect. Notes in Comp. Sci., Springer-Verlag, 2003, pp. 211–223.
6. A. Forte: *The Structure of Atonal Music*, Yale University Press, New Heaven, 1973.
7. D. Lewin: *Generalized Musical Intervals and Transformations*, Yale University Press, New Heaven, 1987.
8. G. Mazzola: *The Topos of Music*, Basel: Birkhäuser Verlag, 2003.
9. C. MIDI Manufacturers Association, Los Angeles: *The Complete Detailed MIDI 1.0 Specification*, 1996.
10. D. Tymoczko: *Scale Theory, Serial Theory, and Voice Leading*, Princeton University, 2006.

A Some theoretical results on voicings and connected chord progressions

In this appendix we present in details some theoretical results that have been already referenced in Section 3.

We need first some further definitions. Let Σ be an alphabet, and let X and Y be strings over Σ. We say that X is a prefix of Y, and write $X \sqsubset Y$, if $Y = X_{\bullet}Z$ for some string Z. If s is a symbol of Σ and $0 \leq k < |X|$, we denote by $X(k:s)$ the string obtained from X by replacing its $(k+1)$-th symbol by s, so that the following equality holds

$$X(k:s) = X[0\mathinner{..}k-1]_{\bullet}s_{\bullet}X[k\mathinner{..}|X|-1]\,.$$

Moreover, we put also $X(k:s) = X$ when $k \geq |X|$.

Lemma 15. *Let V be a voicing of length m over the alphabet Σ, and let $s \in \Sigma \setminus Set(V)$. Then, for each $k \geq 0$, $V(k:s)$ is a voicing and $V \Longrightarrow V(k:s)$.*

Proof. Plainly, $V(k:s)$ is a voicing, since $s \notin Set(V)$. To show that $V \Longrightarrow V(k:s)$, we set $S = V(k:s)_{\bullet}V$ and $V_i = S[|S|-m-i\mathinner{..}|S|-1-i]$, for $i = 0,1,\ldots,|S|-m$. Then it is easy to verify that $\langle V_0, V_1, \ldots, V_{|S|-m}\rangle$ is a voice leading connecting V to $V(k:s)$. □

Theorem 16. *Any two voicings of the same length m over an alphabet Σ of size larger than m are connectable in Σ.*

Proof. Let V and W be two voicings of length m over an alphabet Σ of size $\sigma > m$. We will show that $V \Longrightarrow W$, by proving by induction that for each $\ell = 0,1,\ldots,m$ there is a voicing Z of length m such that $V \Longrightarrow Z$ and $W[0\mathinner{..}\ell-1] \sqsubset Z$.

For $\ell = 0$, it is enough to take $Z = V$, since $V \Longrightarrow V$ (by (1) of Lemma 1), and $W[0\mathinner{..}\ell-1] = \varepsilon \sqsubset V$.

For the inductive step, let $1 \leq \ell \leq m-1$, and let us assume that there is a voicing U (of length m) such that $V \Longrightarrow U$ and $W[0\mathinner{..}\ell-1] \sqsubset U$. Since $|U| < \sigma$, the set $\Sigma \setminus Set(U)$ is nonempty, so we can pick an $s \in \Sigma \setminus Set(U)$. Let

$$\begin{aligned} k &= \mathbf{min}(\{0 \leq i < m : U[i] = W[\ell]\} \cup \{m\})\,, \\ \widehat{U} &= U(k:s)\,, \\ Z &= \widehat{U}(\ell : W[\ell])\,. \end{aligned}$$

Then, by Lemma 15, the strings $\widehat{U}$ and Z are voicings of length m and, additionally, $U \Longrightarrow \widehat{U}$ and $\widehat{U} \Longrightarrow Z$ hold. By the transitivity of the connectivity relation "$\Longrightarrow$" (cf. Lemma 1) we have $V \Longrightarrow Z$. To conclude the proof of the inductive step, we have only to observe that $W[0\mathinner{..}\ell] \sqsubset Z$ plainly holds. □

Remark 17. The proof of Theorem 16 suggests an algorithm to effectively construct a voice leading $\mathcal{V}$ which connects any two given voicings V and W of the same length m, over an alphabet Σ of size larger than m. Observe, however, that the voice leading $\mathcal{V}$ so constructed is not, in general, the smallest possible. □

Corollary 18. *Let V and W be voicings of length m over an alphabet Σ of size at least $m+2$. Then there is a connected voice leading $\langle V_0, V_1, \ldots, V_n\rangle$, which connects V to W with respect to Σ,such that $Set(V_i) \neq Set(V_{i+1})$, for $i = 0,1,\ldots,n-1$.*

Proof. As usual, let $\sigma = |\Sigma|$. Since $|Set(V)| = |Set(W)| < \sigma$, there exist $a, b \in \Sigma$ such that $a \notin Set(V)$ and $b \notin Set(W)$. Let us put $V' = V_\bullet a$ and $W' = W_\bullet b$. Then V' and W' are voicings of length $m+1$, and since $\sigma > m+1$, by Theorem 16, there exists a connected voice leading $\mathcal{U} = \langle U_0, U_1, \ldots, U_n\rangle$ such that $U_0 = V'$ and $U_n = W'$.

Next, we define a voice leading $\mathcal{V} = \langle V_0, V_1, \ldots, V_n\rangle$, by putting $V_i = U[0\,..\,m-1]$, for $i = 0, 1, \ldots, n$. To begin with, notice that $V_0 = V$ and $V_n = W$. Moreover, since $V_{i+1} = U_{i+1}[0]_\bullet V_i[0\,..\,m-2]$, for $i = 0, 1, \ldots, n-1$, it follows that $\mathcal{V}$ is a voice leading which connects V to W, with respect to the alphabet Σ. It only remains to show that $Set(V_i) \neq Set(V_{i+1})$, for $i = 0, 1, \ldots, n-1$, which we do as follows. By way of contradiction, let us assume that $Set(V_j) = Set(V_{j+1})$, for some $j \in \{0, 1, \ldots, n-1\}$. Then we would have $V_j[m-1] = V_{j+1}[0]$, as $V_j \longrightarrow V_{j+1}$. But $V_j[m-1] = U_j[m-1] = U_{j+1}[m]$, as $U_j \longrightarrow U_{j+1}$, and $V_{j+1}[0] = U_{j+1}[0]$. Therefore, $U_{j+1}[m] = U_{j+1}[0]$, which is a contradiction, since U_{j+1} is a voicing. □

Theorem 19. *Let $\mathcal{C} = \langle C_0, C_1, \ldots, C_{m-1}\rangle$ be a connected chord progression, with $m = size(C_0) \geq 2$, such that $C_i \neq C_{i+1}$, for $i = 0, 1, \ldots, m-2$, and let $\langle V_0, V_1, \ldots, V_{m-1}\rangle$ be a connected voice leading of $\mathcal{C}$. Then*

$$\bigcap_{i=0}^{k} C_i = Set(V_0[0\,..\,m-k-1])\,,$$

for $k = 0, 1, \ldots, m-1$.

Proof. We begin by showing that

$$V_0[0\,..\,m-k-1] = V_i[i\,..\,i+m-k-1]\,, \quad \text{for } 0 \leq i \leq k\,, \tag{1}$$

by induction on $k = 0, 1, \ldots, m-1$.

For $k = 0$, (1) reduces to $V_0 = V_0$, which is trivially true.

For the inductive step, let us suppose that (1) holds for some k such that $0 \leq k \leq m-2$. Then, in order to prove that (1) holds also for $k+1$, we need to verify that $V_0[0\,..\,m-k-2] = V_{k+1}[k+1\,..\,m-1]$.

Since $V_k \longrightarrow V_{k+1}$, we have $V_k[k\,..\,m-2] = V_{k+1}[k+1\,..\,m-1]$. In addition, by the inductive hypothesis, we have $V_0[0\,..\,m-k-1] = V_k[k\,..\,m-1]$, which plainly implies $V_0[0\,..\,m-k-2] = V_k[k\,..\,m-2]$, by dropping the last symbol in both voicings. Therefore we have $V_0[0\,..\,m-k-2] = V_{k+1}[k+1\,..\,m-1]$, completing the proof of (1).

From (1), it follows that

$$Set(V_0[0\,..\,m-k-1]) \;\subseteq\; \bigcap_{i=0}^{k} Set(V_i) \;=\; \bigcap_{i=0}^{k} C_i\,,$$

so that we are only left with proving the converse inclusion

$$\bigcap_{i=0}^{k} C_i \;\subseteq\; Set(V_0[0\,..\,m-k-1])\,. \tag{2}$$

Since $\bigcap_{i=0}^{k} C_i \subseteq C_0 = Set(V_0)$, to show (2) it is enough to prove that

$$Set(V_0[m-k\,..\,m-1]) \cap \bigcap_{i=0}^{k} C_i \;=\; \emptyset\,, \tag{3}$$

which we do as follows. Let $m-k \leq h \leq m-1$. Then $0 < m-h \leq k$, so that $\bigcap_{i=0}^{k} C_i \subseteq C_{m-h-1} \cap C_{m-h} = Set(V_{m-h-1}) \cap Set(V_{m-h})$. Since $V_{m-h-1} \longrightarrow V_{m-h}$, we have $Set(V_{m-h-1}[0\,..\,m-2]) = Set(V_{m-h}[1\,..\,m-1])$. But $C_{m-h-1} \neq C_{m-h}$, hence $V_0[h] = V_{m-h-1}[m-1] \notin C_{m-h}$, which implies $V_0[h] \notin \bigcap_{i=0}^{k} C_i$. Therefore (3) holds, which in turn implies (2), concluding the proof of the theorem. □

On the Problem of Deciding If a Polyomino Tiles the Plane by Translation*

Srečko Brlek and Xavier Provençal

Laboratoire de Combinatoire et d'Informatique Mathématique,
Université du Québec à Montréal,
CP 8888, Succ. Centre-ville, Montréal (QC) Canada H3C3P8
[brlek,provenca]@lacim.uqam.ca

Abstract. The words that tile the plane by translation are characterized by the Beauquier-Nivat condition. By using the constant time algorithms for computing the longest common extensions in two words, we provide a linear time algorithm in the case of pseudo-square polyominoes, improving the previous quadratic algorithm of Gambini and Vuillon. For pseudo-hexagon polyominoes not containing arbitrarily large square factors we also have a linear algorithm.

Keywords: tiling polyominoes, plane tesselation, longest common extensions

1 Introduction

The way of tiling planar surfaces takes its roots in the ancient times for decorative purposes. More recently, connections were established with computational theory, mathematical logic and discrete geometry, where tilings are often regarded as basic objects for proving undecidability results for planar problems. Tilings have been also used in physics, as powerful tools for studying quasi-crystal structures: in particular these structures can be better understood by representing them as rigid tilings decorated by atoms in a uniform fashion. Their long-range order can consequently be investigated in a purely geometrical framework, after assigning to every tiling a structural energy.

A classical result of Berger [2] states that given a set of tiles, it is not decidable whether there exists a tiling of the plane which involves all its elements. This result has been achieved by constructing an aperiodic set of tiles, and it has been strengthened afterwards by Gurevich and Koriakov [12] to the periodic case.

It was therefore natural to seek manageable problems, and polyominoes appeared as good candidates. Invented by Golomb [10] who coined the term *polyomino*, these objects, also called n-ominoes or lattice animals, gained some interest after being popularized by Gardner in mathematical games [9]. In statistical physics they appear as models for percolation theory and their combinatorial properties have been extensively studied. These nowadays well studied combinatorial objects are still related to many challenging problems, such as tiling problems [5,11], games [9], among many others (see Weisstein [18] for more pointers). Their enumeration is also an open problem despite the fact that restricted classes have been fully described.

There are different types of polyominoes and here we consider a *polyomino* as a finite union of unit lattice closed squares (pixels) in the discrete plane whose boundary consists of a simple closed polygonal path using 4-connectedness (Figure 2(a)). In particular, polyominoes are simply connected (contain no holes), and have no multiple points (Figure 1(a)).

* with the support of NSERC (Canada)

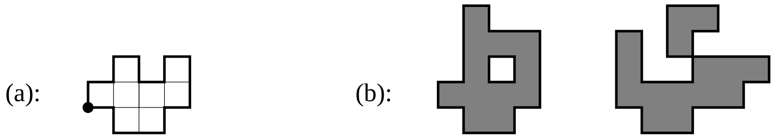


Figure 1. (a) a polyomino; (b) not a polyomino

The problem of deciding if a given polyomino tiles the plane by translation goes back to Wisjhoff and Van Leeuven [19] who coined the term *exact polyomino* for these, and also provided a polynomial $\mathcal{O}(n^4)$ algorithm for solving the problem. Polyominoes may be coded by words on a 4-letter alphabet $\Sigma = \{a, \overline{a}, b, \overline{b}\}$, also known as the Freeman chain codes [6,7], coding their boundaries (see [3] for further reading). For

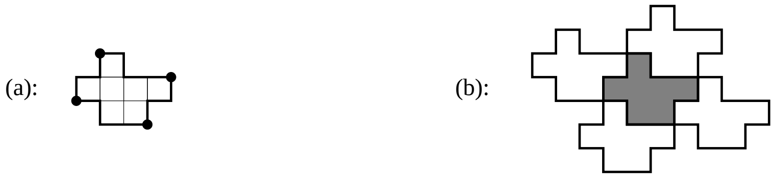


Figure 2. (a) an exact polyomino; (b) the associated tiling

instance, the boundary $\mathbf{b}(P)$ of the polyomino in Figure 2 (a), in a counterclockwise manner, is coded by the word $w = a\,\overline{b}\,a\,a\,b\,a\,b\,b\,\overline{a}\,\overline{b}\,\overline{a}\,b\,\overline{a}\,\overline{b}\,\overline{a}\,\overline{b}$.

Observe that we may consider the words as circular which avoids a fixed origin. The *perimeter* of a polyomino P is the length of its boundary word $\mathbf{b}(P)$ and is of even length $2n$. Beauquier and Nivat [1] gave a characterization stating that the boundary of such a polyomino P may be factorized (not necessarily in a unique way) as

$$\mathbf{b}(P) = A \cdot B \cdot C \cdot \widehat{A} \cdot \widehat{B} \cdot \widehat{C} \tag{1}$$

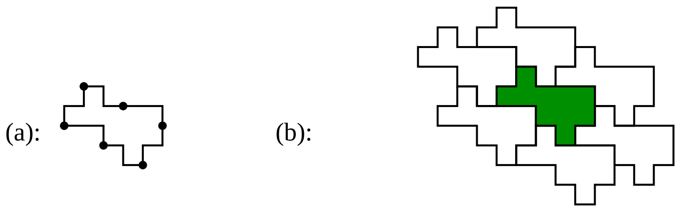
where at most one of the variables is possibly empty. The operation $\widehat{(\ )}$ appearing in (1) is defined by $\widehat{U} = \overline{\widetilde{U}}$, where $\widetilde{(\ )}$ is the usual reversal operation and $\overline{(\ )}$ the complement on $\Sigma = \{a, \overline{a}, b, \overline{b}\}$. For instance, the exact polyomino in Figure 2 (b) is coded by the circular word

$$w = a\,a\,b\,a\,b\,\overline{a}\,\overline{a}\,b\,\overline{a}\,\overline{b}\,\overline{a}\,\overline{b}\,a\,\overline{b},$$

its semi-perimeter is 7, and its boundary may be factorized as

$$\mathbf{b}(P) = A \cdot B \cdot \widehat{A} \cdot \widehat{B} = a\,\overline{b}\,a\,a \cdot b\,a\,b \cdot \overline{a}\,\overline{a}\,b\,\overline{a} \cdot \overline{b}\,\overline{a}\,\overline{b},$$

and for the exact polyomino P' below



the boundary may be factorized as

$$\mathbf{b}(P') \equiv a\,a\,\overline{b} \cdot a\,\overline{b}\,a \cdot b\,a\,b \cdot b\,\overline{a}\,\overline{a} \cdot \overline{a}\,b\,\overline{a} \cdot \overline{b}\,\overline{a}\,\overline{b}.$$

Determining if a given word $w \in \Sigma^n$ is the boundary of a polyomino is computed in $\mathcal{O}(n)$. Therefore the problem reduces to finding a factorization satisfying the Beauquier and Nivat condition. Recently, Gambini and Vuillon [8] improved the Wisjhoof-van Leeuven bound by designing an $\mathcal{O}(n^2)$ algorithm that checks the Beauquier-Nivat condition (1).

The underlying idea of our approach is to search efficiently the pairs of homologue factors $X, \widehat{X}$. Our algorithms borrow from Lothaire [16] (for instance) that the *Longest-Common-Factor*, the *Longest-Common-Prefix* and the *Longest-Common-Suffix* in two words may be computed in linear time. The approach is also inspired by the linear algorithm of Gusfield and Stoye [14] for detecting *tandem repeats* in a word, and by the linear algorithm used to detect repetitions with gaps, as shown in Lothaire [16]. More precisely, the computation of the *Longest-Common-Left-Extension* (LCLE(u, v)) and *Longest-Common-Right-Extension* (LCRE(u, v)) is achieved in constant time, provided a linear pre-processing is performed on u and v, by a clever utilization of suffix-trees (see Gusfield [13]).

Taking advantage of these algorithms we provide a linear algorithm, with respect to the length of words, for pseudo-square polyominoes (Theorem 8). We establish a first step in order to provide a linear algorithm for pseudo-hexagons as well. Indeed, for boundary words not having two large square repetitions there is a linear algorithm to decide whether a polyomino tiles the plane by translation or not (Theorem 11).

2 Preliminaries

Let Σ be a finite alphabet whose elements are called *letters*. Finite words are sequences of letters, that is, functions $w : [0..n-1] \longrightarrow \Sigma$, and the set of words of length n is denoted Σ^n. The free monoid $\Sigma^* = \cup_{n=0}^{\infty} \Sigma^n$ is the set of all finite words and the empty word is denoted ϵ.

A morphism is a function $\sigma : \Sigma^* \longrightarrow \Sigma^*$ such that $\sigma(uv) = \sigma(u)\sigma(v)$. Clearly a morphism is defined by the image of the letters. A *factor* f of w is a word $f \in \Sigma^*$ satisfying

$$\exists x \in \Sigma^*, y \in \Sigma^*, w = xfy.$$

If $x = \epsilon$ (resp. $y = \epsilon$) then f is called *prefix* (resp. *suffix*). The set of all factors of w is denoted by $F(w)$, and those of length n is $F_n(w) = F(w) \cap \Sigma^n$. Finally $\mathrm{Pref}(w)$ denotes the set of all prefixes of w. The length of a word w is $|w|$, and the number of occurrences of a factor $f \in \Sigma^*$ is $|w|_f$. A word is said to be *primitive* if it is not a power of another word. If $w = pu$, and $|w| = n, |p| = k$, then $p^{-1}w = w[k+1]..w[n-1] = u$ is the word obtained by erasing p. As a special case when $|p| = 1$ we have the shift operator σ defined by $\sigma(w) = w[1..(n-1)]$. Another useful operator is the circular permutation ρ defined by $\rho(w) = w[1..(n-1)] \cdot w[0]$.

Two words u and v are *conjugate* when there are words x, y such that $u = xy$ and $v = yx$. Equivalently, u and v are conjugate if and only if there exists an index k such that $v = \rho^k(v)$. Conjugation is an equivalence relation written $u \equiv v$. The *reversal* $\widetilde{u}$ of $u = u_1u_2 \cdots u_n \in \Sigma^n$ is the word $\widetilde{u} = u_nu_{n-1} \cdots u_1$. A *palindrome* is a word p such that $p = \widetilde{p}$, and for a language $L \subseteq \Sigma^\infty$, we denote by $\mathrm{Pal}(L)$ the set of its palindromic factors.

Paths on the square lattice $\mathbb{Z} \times \mathbb{Z}$ are encoded on the alphabet $\Sigma = \{a, \overline{a}, b, \overline{b}\}$ identified with the unit steps $\{\rightarrow, \leftarrow, \uparrow, \downarrow\}$. Parallel paths always define a translation and we say that two words are *homologue* when the corresponding paths define a translation. More precisely, two words u and v are said to be *homologue* when either

(i) $u = v$, or
(ii) $u = \widehat{v}$.

An exact polyomino P whose boundary is $\mathbf{b}(P) = A \cdot B \cdot C \cdot \widehat{A} \cdot \widehat{B} \cdot \widehat{C}$ is called a *pseudo-hexagon* if none of the variables is empty and a *pseudo-square* otherwise. In this factorization A (resp. B, C) and $\widehat{A}$ (resp. $\widehat{B}, \widehat{C}$) are homologue and define the respective translations. For instance, the translations defined by the homologue sides of the pseudo-square polyomino

$$\mathbf{b}(P) = A \cdot B \cdot \widehat{A} \cdot \widehat{B} = a\,\overline{b}\,a\,a \cdot b\,a\,b \cdot \overline{a}\,\overline{a}\,b\,\overline{a} \cdot \overline{b}\,\overline{a}\,\overline{b}$$

are shown in Figure 3 (a). In the case of a pseudo-hexagon, as in Figure 3(b), the

Figure 3. Translations defined by homologue sides of a polyomino tile

translations are related by the relation $t_3 = t_1 + t_2$. Moreover, the relative positions of the starting and ending point of any path is completely determined by the sum of the unit vectors corresponding to each letter. By abuse of notation we write for a path $w : [0..n-1] \rightarrow \Sigma$

$$\overrightarrow{w} = \sum_{k=0}^{n-1} \overrightarrow{w_k}.$$

Note that $\overrightarrow{w} = 0$ if and only if w is a closed path, and that $\overrightarrow{u} = -\overrightarrow{\widehat{u}}$.

3 Searching the homologue factors

Since polyominoes are coded by circular words w, in order to find the homologue factors it is convenient to work with $w \cdot w$ since a pair of homologue factors might be split, depending on the starting point.

Therefore, finding the homologue factors amounts to look for the *longest common factor* of ww and $\widehat{ww}$ denoted $\mathrm{LCF}(ww, \widehat{ww})$.

For instance the longest common factors of the polyomino-tile P in Figure 2 (b) are

$$\mathrm{LCF}(ww, \widehat{ww}) = \{a\,\overline{b}\,a\,a, \overline{a}\,\overline{a}\,b\,\overline{a}\}$$

and they are necessarily homologue sides(!). Indeed, since we know the positions i and j of $a\,b\,a\,a$ and $\overline{a}\,\overline{a}\,b\overline{a}$ in w, this is easy to check in linear time. Clearly the boundary of P may be written as

$$\mathbf{b}(P) = w = a\,\overline{b}\,a\,a \cdot u \cdot \overline{a}\,\overline{a}\,b\,\overline{a} \,\cdot\, v$$

and then one easily checks that $v = \widehat{u}$. Unfortunately the situation is not always that good. Indeed, let $w = a\,a\,b\,b\,b\,a\,a\,b\,\overline{a}\,\overline{a}\,b\,\overline{a}\,\overline{b}\,\overline{b}\,\overline{b}\,\overline{a}\,\overline{a}\,\overline{b}\,a\,\overline{b}$. Then the longest homologue factors of w are (see Figure 4)

$$\mathrm{LCF}(ww, \widehat{ww}) = \{a\,a\,b\,b\,b\,a, \overline{a}\,\overline{b}\,\overline{b}\,\overline{b}\,\overline{a}\,\overline{a}\},$$

but $w = a\,a\,b\,b\,b\,a \cdot a\,b\,\overline{a}\,\overline{a}\,b \cdot \overline{a}\,\overline{b}\,\overline{b}\,\overline{b}\,\overline{a}\,\overline{a} \cdot \overline{b}\,a\,\overline{b}$ does not satisfy the Beauquier-Nivat condition. A good factorization is $w \equiv b\,b \cdot b\,a\,a\,b \cdot \overline{a}\,\overline{a}\,b\,\overline{a} \cdot \overline{b}\,\overline{b} \cdot \overline{b}\,\overline{a}\,\overline{a}\,\overline{b} \cdot a\,\overline{b}\,a\,a$. This means

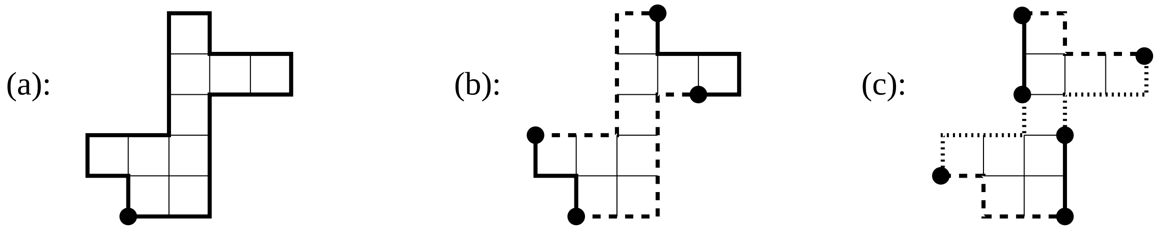


Figure 4. (b) longest homologue factors; (c) a good factorization

that not all the homologue factors provide a factorization, and good candidates are those separated by factors of same length.

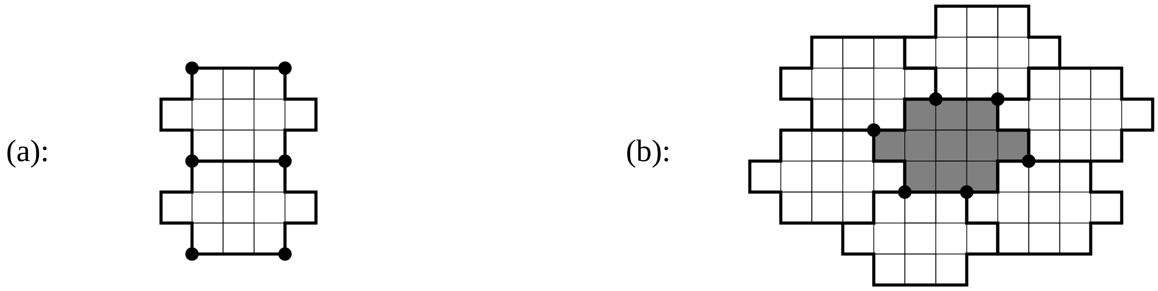
Definition 1. *Let $w \equiv \mathbf{b}(P)$ be the boundary word of a polyomino P. A factor A of w is* admissible *if*

(i) $w \equiv Ax\widehat{A}y$, *for some x, y such that $|x| = |y|$;*
(ii) *A is* saturated*, that is, $x_0 \neq \overline{x_{k-1}}$ and $y_0 \neq \overline{y_{k-1}}$ where $k = |x| = |y|$.*

Nevertheless, admissibility is ensured for words that code the boundary of polyominoes. Indeed, Gambini and Vuillon established the following property ([8], section 3.1) by using a geometric result of Daurat and Nivat [5].

Lemma 2. *Let $w \equiv ABC\widehat{A}\widehat{B}\widehat{C}$ be a Beauquier-Nivat factorization of the boundary $\mathbf{b}(P)$ of an exact polyomino P. Then A, B and C are admissible.*

Conversely, not all admissible factors lead to a Beauquier-Nivat factorization. For instance, in the polyomino $w \equiv a\,a\,a\,b\,a\,b\,\overline{a}\,b\,\overline{a}\,\overline{a}\,\overline{a}\,\overline{b}\,\overline{a}\,\overline{b}\,a\,\overline{b}$ shown below the factor aaa



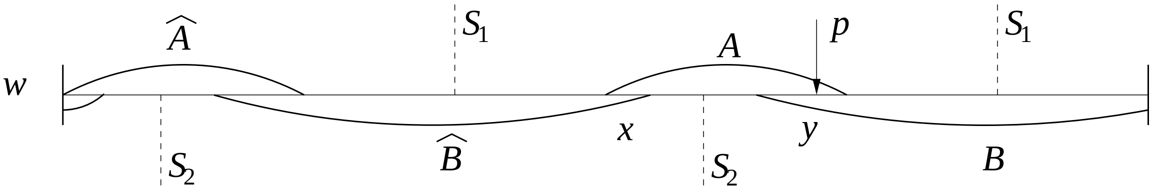
is admissible but does not provide a correct factorization of w. Indeed, $A = aa$ is the admissible factor ($w = Ax\widehat{A}y$ with $x = abab\overline{a}b$, $y = \overline{a}\overline{b}\overline{a}\overline{b}a\overline{b}$) yielding a correct factorization with $B = aba$ and $C = b\overline{a}b$:

$$w \equiv a\,a \cdot a\,b\,a \cdot b\,\overline{a}\,b \cdot \overline{a}\,\overline{a} \cdot \overline{a}\,\overline{b}\,\overline{a} \cdot \overline{b}\,a\,\overline{b}.$$

The following proposition establishes a useful property.

Proposition 3. *Let $w = \mathbf{b}(P) \in \Sigma^{2n}$ be the contour of a polyomino P and let p be any fixed position in w. Let X be the set of all admissible factors overlapping the position p and $\widehat{X}$ be the set of their respective homologue factors. Then, there exist at least one position in w that is not covered by any element of $X \cup \widehat{X}$.*

Proof. By contradiction, assume that there is no such point. Let $A \in X$ be the factor that starts at the leftmost position and $B \in X$ be the one that ends at the righmost position as shown below. The homologue factors $A, \widehat{A}$ and $B, \widehat{B}$ always define two

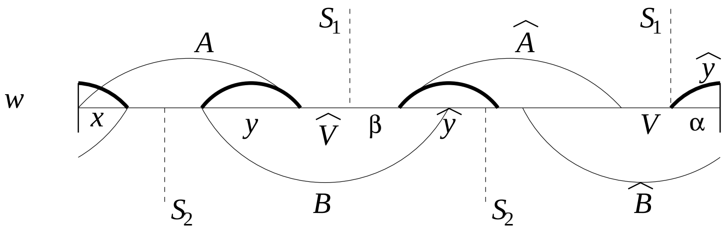


symmetries denoted respectively by S_1 and S_2. Let x be the overlap between A and $\widehat{B}$, and y be the overlap between A and B. Without loss of generality we may consider that $|y| \geq |x|$. If $|x| = |y|$ the symmetry implies that $x = \widehat{y}$ and the factorization is

$$w \equiv \mathbf{x}\,U\,\widehat{x}\,V\,x\,\widehat{U}\,\widehat{x}\,\widehat{V}\,. \tag{2}$$

We use a property proved in Brlek et all. ([4], DLT2005) that simplifies a result of Daurat and Nivat ([5], IWCIA'03) on the number of salient and reentrant points of discrete sets: indeed, the number of right turns minus the number of left turns in a closed and non-intersecting path on a square lattice is 4. In equation 2, notice that all turns in a factor are cancelled by those of its homologue. Therefore we only have to consider the turns between consecutive factors. Reading, the word w from left to right, we see that each pair of consecutive factors is cancelled by its homologue: $\mathbf{x}\,U$ is cancelled by $\widehat{U}\,\widehat{x}$, $U\,\widehat{x}$ by $x\,\widehat{U}$, $\widehat{x}\,V$ by $\widehat{V}\mathbf{x}$ (the word w is circular), and $V\,x$ by $\widehat{x}\,\widehat{V}$. Hence the difference between right and left turns is 0, and w is self intersecting. Contradiction.

If $|x| \neq |y|$ we have the following situation where the factor y (thick line) propa-



gates as shown by using the symmetries S_1 and S_2. In this case $\widehat{y}$ does not overlap $\widehat{A}$ in $\widehat{B}$, so let V be the factor between $\widehat{A}$ and $\widehat{y}$. We have the following factorization

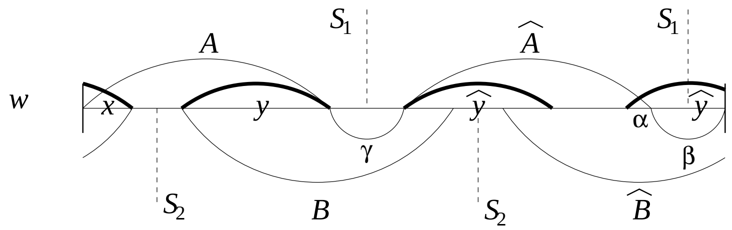
$$w \equiv A\,\widehat{V}\,\beta\,\widehat{A}\,V\,\alpha\,.$$

Passing to vectors, and using commutativity of addition, we have

$$\overrightarrow{w} = \overrightarrow{A} + \overrightarrow{\widehat{A}} + \overrightarrow{V} + \overrightarrow{\widehat{V}} + \overrightarrow{\beta} + \overrightarrow{\alpha} = \overrightarrow{\beta} + \overrightarrow{\alpha} = \overrightarrow{0}.$$

But $\widehat{y} = \alpha x$, so that β is followed by α in w. Therefore $\beta\alpha$ is a nonempty closed path on the boundary of P. Contradiction.

In the case where $\widehat{y}$ does overlap $\widehat{A}$ in $\widehat{B}$ we have the following situation where



$\overrightarrow{\gamma} + \overrightarrow{\beta} = 0$ (by closure property $\overrightarrow{w} = 0 = \overrightarrow{A} + \overrightarrow{\gamma} + \overrightarrow{\widehat{A}} + \overrightarrow{\beta}$). Moreover, $\widehat{y} = \alpha\beta x$, so that $y\,\gamma\,\widehat{y}$ contains the nonempty factor $\widehat{\alpha}\,\gamma\,\alpha\,\beta$ corresponding to a closed path. Contradiction. $\square$

Proposition 3 specializes for pseudo-squares as follows. Assume that a pseudo-square P has two factorizations

$$w = \mathbf{b}(P) \equiv AB\widehat{A}\widehat{B} \equiv XY\widehat{X}\widehat{Y}$$

where $A = sXt$. Then, by using the same argument as in the proof above, the boundary of P contains a loop yielding a contradiction.

Corollary 4. *If $w = \mathbf{b}(P) \equiv AB\widehat{A}\widehat{B} \equiv XY\widehat{X}\widehat{Y}$ are two distinct factorizations of the boundary of a pseudo-square P, then there exist α, β, γ such that $A = \alpha\beta$ and $X = \beta\gamma$.*

As an exemple we have the following pseudo-square

$$aba \cdot \overline{b}a\overline{b} \cdot \overline{a}\overline{b}\overline{a} \cdot b\overline{a}b \equiv bab \cdot a\overline{b}a \cdot \overline{b}\overline{a}\overline{b} \cdot \overline{a}b\overline{a},$$

showing two distinct factorizations. The problem of enumerating all the factorizations of a given pseudo-square will be addressed in a forthcoming paper.

3.1 A linear time algorithm for detecting pseudo-squares

The main idea used to achieve linear time factorization, is to choose a position p in w and then list all the *admissible factors* A that overlap this fixed position. The following auxiliary functions are useful. The *Longest-Common-Right-Extension* (LCRE) and *Longest-Common-Left-Extension* (LCLE) of two words u and v at positions respectively m and n are partial functions

$$\text{LCRE}, \text{LCLE} : \Sigma^* \times \Sigma^* \times \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$$

defined as follows. For $u, v \in \Sigma^*$, let m and n be such that $0 \leq m \leq |u|$ and $0 \leq n \leq |v|$, then

$$\mathrm{LCRE}(u, v, m, n) = \mathrm{LCP}(\rho^m(u), \rho^n(v))$$
$$\mathrm{LCLE}(u, v, m, n) = \mathrm{LCS}(\rho^{|u|-m}(u), \rho^{|v|-n}(v))$$

Remark 5. It is clear from the definition above that LCRE and LCLE may be computed in linear time. Their computation may also be performed directly by the following formulas. Since we use circular words w, denote $\underline{m} = m \mod |w|$. If $u[\underline{m}] = v[\underline{n}]$ then

(i) $\mathrm{LCRE}(u, v, \underline{m}, \underline{n}) = \mathbf{max}\{k \in \mathbb{N} \mid u[\underline{m}..(\underline{m} + \underline{k})] = v[\underline{n}..(\underline{n} + \underline{k})]\} + 1$,
(ii) $\mathrm{LCLE}(u, v, \underline{m}, \underline{n}) = \mathbf{max}\{k \in \mathbb{N} \mid u[(\underline{m} - \underline{k})..\underline{m}] = v[(\underline{n} - \underline{k})..\underline{n}]\} + 1$,

and, otherwise, $\mathrm{LCRE}(u, v, \underline{m}, \underline{n}) = \mathrm{LCLE}(u, v, \underline{m}, \underline{n}) = 0$.

For example, if $u = aabbbaabaabababababa$, $v = babaabbbaabbababab b$, $i = 4$ and $j = 7$ then (note that the words all starts at position 0) we have

$$u = \underline{a\,a\,b\,b}\,\mathbf{b}\,\underline{a\,a\,b}\,a\,a\,b\,a\,b\,a\,b\,a\,b\,a,$$
$$v = b\,a\,b\,\underline{a\,a\,b\,b}\,\mathbf{b}\,\underline{a\,a\,b}\,b\,a\,b\,a\,b\,a\,b\,b,$$

and $\mathrm{LCRE}(u, v, 4, 7) = 4, \mathrm{LCLE}(u, v, 4, 7) = 5$. On the other hand LCRE(u,v,4,1) = LCLE(u,v,4,1)=0.

Later we will need to perform these computations $\mathcal{O}(n)$ times. Fortunately, the computation of LCLE and LCRE is achieved in constant time, provided a linear pre-processing is performed on u and v, by a clever utilization of suffix-trees (see Gusfield [13], section 9.1, or Gusfield and Stoye [14], page 531).

Lemma 6. *Let $w = \mathbf{b}(P)$ be the boundary of P. For each occurrence of A in w and each occurrence of A in $\widehat{w}$, whether A is admissible or not is decidable in constant time.*

Proof. Given an occurrence of A in $\widehat{w}$, one computes in constant time the corresponding position of $\widehat{A}$ in w. If $\widehat{A}$ overlaps A in w is decidable in constant time. If $\widehat{A}$ and A do not overlap then, $u \equiv Ax\widehat{A}y$ and A is an admissible factor, by definition, if and only if the three following conditions are verified : $|x| = |y|$, $x_0 \neq \overline{x_{k-1}}$ and $y_0 \neq \overline{y_{k-1}}$ where $k = |x| = |y|$. □

Lemma 7. *Let $w = \mathbf{b}(P) \in \Sigma^{2n}$ be the boundary of P. For any position p in w, listing all the admissible factors overlapping p is computed in linear time.*

Proof. The following algorithm lists all admissible factors containing the p-th letter w. Since the longest common right and left extension problem can be solved in constant time after linear time pre-processing.

Algorithm 1
Input: $w = \mathbf{b}(P) \in \Sigma^{2n}$
Pre-process w and $\widehat{w}$ so that LCLE *and* LCRE *take constant time*
For $i := 0$ *to* $2n - 1$ **do**
    $l := \mathrm{LCLE}(w, \widehat{w}, p, i) - 1$
    $r := \mathrm{LCRE}(w, \widehat{w}, p, i) - 1$

4 : $\quad\quad A := w[p-l,\ldots,p+r]$
5 : $\quad\quad$ **If** $w \equiv Ax\widehat{A}y$ *with* $|x| = |y|$ **then**
6 : $\quad\quad\quad$ *Add* A *to the list of admissible factors.*
7 : $\quad\quad$ **end if**
8 : $\quad$ **end for**

Using the modulo in managing the positions is superfluous because we may assume, without loss of generality (since w is a circular word) that $p = n$. Note that, by definition of LCRE and LCLE, the factor A in this algorithm is necessarily saturated. As shown in Lemma 6, the condition can be tested in constant time by direct computation of positions in w. Finally, the loop is performed exactly $2n$ times. □

This lemma implies that the number of admissible factors in a word is linear. To determine a precise upper bound remains an open problem which is similar to the problem of determining a tight upper bound for the number of distinct squares in a word (see for instance Lothaire [16] or Ilie [15]).

Theorem 8. *Let* $w = \mathbf{b}(P) \in \Sigma^{2n}$ *be the boundary of* P. *Determining if* w *codes a* pseudo-square *is decidable in linear time.*

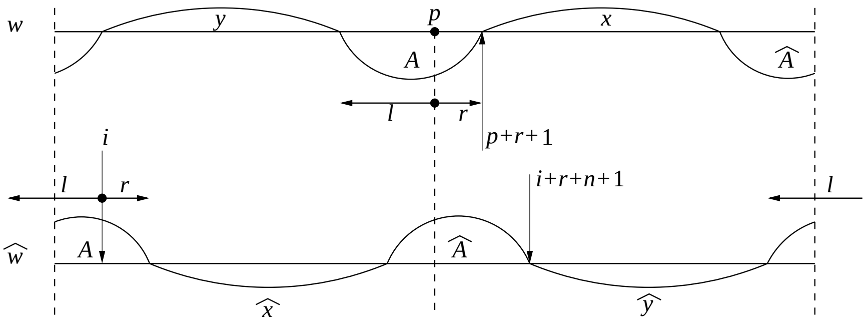


Figure 5. An admissible factor A in w and $\widehat{w}$

Proof. If w encodes an exact polyomino, any position belongs to some admissible factor of the Beauquier-Nivat factorization. Therefore, it suffices to apply Lemma 7 to an arbitrary position p. Then, Algorithm 1 provides the list of all admissible factors overlapping the position p, and it only remains to check, for each admissible factor, if $x = \widehat{y}$. Lemma 2 ensures that if $w \equiv AB\widehat{A}\widehat{B}$ then B is saturated. As shown in Figure 5, it suffices now to replace step 6 in Algorithm 1 by:

$6a$: $\quad$ **If** $\mathrm{LCRE}(w, \widehat{w}, p+r+1, \underline{i+r+n+1}) = |x|$ **then**
$6b$: $\quad\quad P$ is a *pseudo-square.*
$6c$: $\quad$ **End if**

Since LCRE is computed in constant time, the overall algorithm is linear. □

3.2 A linear algorithm for k-square-free pseudo-hexagons

Let $w \equiv \mathbf{b}(P)$ be the boundary word of an exact polyomino P. A factor f of a word w is called a *square* if $f = xx$ for some $x \in \Sigma^+$. The set of squares of a word w is $\mathrm{Squares}(w)$.

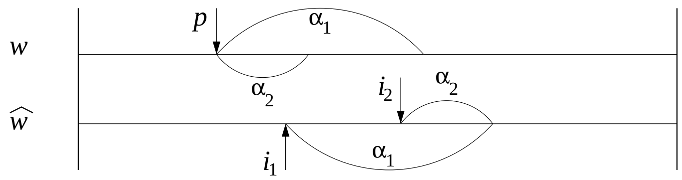
Definition 9. *A word w is k-square-free $\iff$ $\boldsymbol{max}\{|f| : f \in \text{Squares}(w)\} < k$.*

The following technical lemma is useful.

Lemma 10. *Let $w = \mathbf{b}(P) \in \Sigma^{2n}$ a k-square-free word, and let p be any position in w. Then, the number of admissible factors that overlap the position p in w is bounded by $4k + 2\log(n)$.*

Proof. Let $A_1, A_2, \ldots, A_k$ be all the admissible factors that overlaps p in w. Since $w \equiv A_i x \widehat{A_i} y$ with $|x| = |y|$ for all $1 \le i \le k$, all their homologue occurrences $\widehat{A_i}$ overlaps the position $p' = p + n$. Thus, there is a position q such that all A_i overlap q in $\widehat{w}$. In Algorithm 1, all the admissible factors overlapping p in w are listed through a loop such that each iteration can detect at most one of them. Let i_1, i_2 be such that $0 \le i_1 < i_2 < q$ and assume that admissible factors are detected when $i = i_1$ and $i = i_2$.

Let $\alpha_1 = \widehat{w}[i_1, \ldots, q]$ and $\alpha_2 = \widehat{w}[i_2, \ldots, q]$. By definition of *common extension* we also have that $\alpha_1 = w[p, \ldots, p + |\alpha_1| - 1]$ and $\alpha_2 = w[p, \ldots, p + |\alpha_2| - 1]$, as shown below.



This implies that α_2 is prefix and suffix of α_1, so that Lothaire's Proposition 1.3.4 [17] applies. It follows that there are two words $u, v \in \Sigma^*$ such that $\alpha_1 = (uv)^m u$, for some integer m with

$$m(|\alpha_1| - |\alpha_2|) \ge |\alpha_2| \ge (m-1)(|\alpha_1| - |\alpha_2|). \quad (3)$$

If $|\alpha_1| < 2|\alpha_2|$, equation (3) requires m to be greater than 1 and thus α_1 contains a square of length at least $\frac{1}{2}|\alpha_1|$. So, in Algorithm 1, as i goes from 0 to $2n-1$, the number of admissible factors detected is bounded by :

- $\log n$ for i from 0 to $q - 2k$.
- $4k$ for i from $q - 2k + 1$ to $q + 2k - 1$.
- $\log n$ for i from $q + 2k$ to $2n - 1$.

Summing up all these provides the bound. □

Theorem 11. *Let $w = \mathbf{b}(P) \in \Sigma^{2n}$ be a k-square-free word, with $k \in \mathcal{O}(\sqrt{n})$. Determining if w codes a* pseudo-hexagon *is decidable in linear time.*

Proof. The idea is to construct convenient, and not too long, lists of admissible factors and then to use the constant time LCRE function.

Algorithm 2
Input : $w = \mathbf{b}(P) \in \Sigma^{2n}$
  Build L_1 : *list of all admissible factors that overlap position p in w;*
  $m :=$ *position of the rightmost letter of w included in a factor of L_1;*
  Build L_2 : *list of all admissible factors that overlap position $(m+1)$ in w.*
  For *all $A \in L_1$* **do**
    For *all $B \in L_2$* **do**

If $w \equiv ABx\widehat{A}\widehat{B}y$ **then**
  Compute i : *position of* x *in* w;
  Compute j : *position of* $\widehat{y}$ *in* $\widehat{w}$;
  If $\mathrm{LCRE}(w, \widehat{w}, i, j) = |x|$ **then**
    P *is a* pseudo-hexagon;
  End if
End if
End for
End for

Since w is k-square-free, Lemma 10 ensures that L_1 and L_2 each contain less than $4k+2\log n$ elements. The nested loops perform at most $(4k+2\log n)^2$ iterations, and thus, the overall complexity is $\mathcal{O}(n + (4k+2\log n)^2) = \mathcal{O}(n)$. □

4 Concluding remarks

The results above generalize to more general tilings. Indeed, since the Beauquier-Nivat factorization involves path properties, there is no need for a tile to be a polyomino. For instance, the tile T in Figure 6

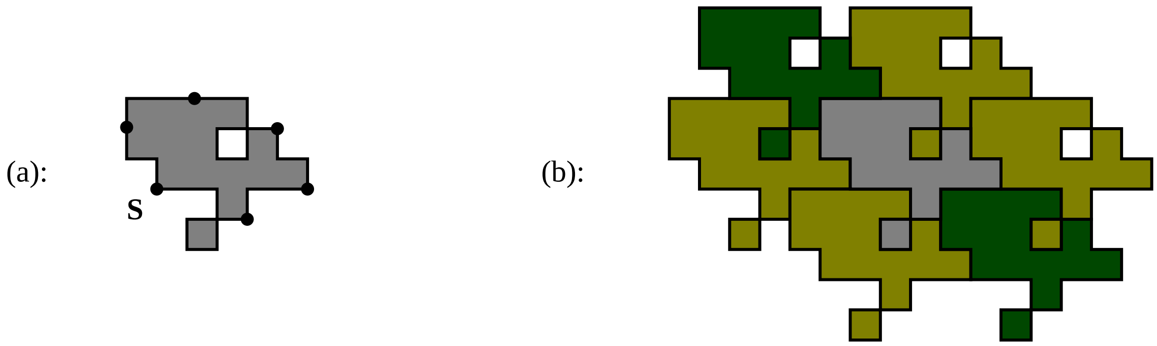


Figure 6. (a) An hexagonal tile with (b) the associated tiling

is hexagonal and its Beauquier-Nivat factorization is (starting from **S**)

$$\begin{aligned}\mathbf{b}(T) &= X \cdot Y \cdot Z \cdot \widehat{X} \cdot \widehat{Y} \cdot \widehat{Z} \\ &= a\,a\,\overline{b}\,\overline{a}\,\overline{b}\,a\,b\,a \cdot b\,a\,a \cdot b\,\overline{a}\,b \cdot \overline{a}\,\overline{b}\,\overline{a}\,b\,a\,b\,\overline{a}\,\overline{a} \cdot \overline{a}\,\overline{a}\,\overline{b} \cdot \overline{b}\,a\,\overline{b}\,.\end{aligned}$$

The contour path is non-crossing, instead of self-avoiding as in the case of polyominoes, and provides an instance of an 8-connected set of cells that tiles the plane by translation. This leads naturally to the problem of characterizing the 8-connected sets of cells that tile the plane by translation. On the other hand there is still a gap to fill. A deeper analysis is needed to lift the condition on the number of square factors in the contour word, in order to provide an optimal algorithm for deciding if a polyomino tiles the plane by translation.
Acknowledgements The authors are grateful to the anonymous referees for their careful reading and valuable comments.

References

1. D. Beauquier and M. Nivat: *On translating one polyomino to tile the plane*. Discrete Comput. Geom., 6 1991, pp. 575–592.
2. R. Berger: *The undecidability of the domino problem*. Mem. Amer. Math. Soc., 66 1966.
3. J.-P. Braquelaire and A. Vialard: *Euclidean paths: A new representation of boundary of discrete regions*. Graphical Models and Image Processing, 61 1999, pp. 16–43.
4. S. Brlek, G. Labelle, and A. Lacasse: *A note on a result of Daurat and Nivat*, in Proc. DLT 2005, 9-th International Conference on Developments in Language Theory, C. de Felice and A. Restivo, eds., no. 3572 in LNCS, Palermo, Italia, 4–8 July 2005, Springer-Verlag, pp. 189–198.
5. A. Daurat and M. Nivat: *Salient and reentrant points of discrete sets*, in Proc. IWCIA'03, International Workshop on Combinatorial Image Analysis, A. del Lungo, V. di Gesu, and A. Kuba, eds., Electronic Notes in Discrete Mathematics, Palermo, Italia, 14–16 May 2003, Elsevier Science.
6. H. Freeman: *On the encoding of arbitrary geometric configurations*. IRE Trans. Electronic Computer, 10 1961, pp. 260–268.
7. H. Freeman: *Boundary encoding and processing*, in Picture Processing and Psychopictorics, B. Lipkin and A. Rosenfeld, eds., Academic Press, New York, 1970, pp. 241–266.
8. I. Gambini and L. Vuillon: *An algorithm for deciding if a polyomino tiles the plane by translations*, tech. rep., LAMA, 2003.
9. M. Gardner: *Mathematical games*. Scientific American, 1958, Sept. pp. 182–192, Nov. pp. 136–142.
10. S. W. Golomb: *Checker boards and polyominoes*. Amer. Math. Monthly, 61 1954, pp. 675–682.
11. S. W. Golomb: *Polyominoes: Puzzles, Patterns, Problems, and Packings*, Princeton Academic Press, 1996.
12. Y. Gurevich and I. Koriakov: *A remark on Berger's paper on the domino problem*. Siberian Journal of Mathematics, 13 1972, pp. 459–463, (in Russian).
13. D. Gusfield: *Algorithms on Strings, Trees and Sequences*, Cambridge University Press, Cambridge (UK), 1997.
14. D. Gusfield and J. Stoye: *Linear time algorithms for finding and representing all the tandem repeats in a string*. Journal of Computer and System Sciences, 69 2004, pp. 525–546.
15. L. Ilie: *A note on the number of distinct squares in a word*, in Proc. Words2005, 5-th International Conference on Words, S. Brlek and C. Reutenauer, eds., vol. 36, Montreal, Canada, 13–17 Sept. 2005, Publications du LaCIM, pp. 289–294.
16. M. Lothaire: *Applied Combinatorics on Words*, Cambridge University Press, Cambridge (UK), 2005.
17. M. Lothaire: *Combinatorics on Words*, Cambridge University Press, Cambridge (UK), 2005.
18. E. Weisstein: *Polyomino, from Wolfram Mathworld*. Available electronically at http://mathworld.wolfram.com/Polyomino.html, 2006.
19. H. A. G. Wijshoff and J. van Leeuven: *Arbitrary versus periodic storage schemes and tesselations of the plane using one type of polyomino*. Inform. Control, 62 1984, pp. 1–25.

2D Context-Free Grammars: Mathematical Formulae Recognition

Daniel Průša and Václav Hlaváč

Czech Technical University, Faculty of Electrical Engineering
Department for Cybernetics, Center for Machine Perception
121 35 Prague 2, Karlovo náměstí 13
{prusa,hlavac}@cmp.felk.cvut.cz, http://cmp.felk.cvut.cz

Abstract. This contribution advocates that two-dimensional context-free grammars can be successfully used in the analysis of images containing objects that exhibit structural relations. The idea of structural construction is further developed. The approach can be made computationally efficient, practical and be able to cope with noise. We have developed and tested the method on a pilot study aiming at recognition of off-line mathematical formulae. The other novelty is not treating symbol segmentation in the image and structural analysis as two separate processes. This allows the system to recover from errors made in initial symbol segmentation.

1 Motivation and Taxonomy of Approaches

The paper serves two main purposes. First, it intends to point the reader's attention to the theory of two-dimensional (2D) languages. It focuses on context-free grammars having the potential to cope with structural relations in images. Second, the paper demonstrates on a pilot study concerning recognition of off-line hand written mathematical formulae that the 2D context-free grammars have the potential to deal with real-life noisy images.

The enthusiasm for grammar-based methods in pattern recognition from the 1970's [6] has gradually faded down due to inability to cope with errors and noise. Even mathematical linguistics, in which the formal grammar approach was pioneered [4], has tended to statistical methods since the 1990s.

M.I. Schlesinger from the Ukrainian Academy of Sciences in Kiev has been developing the 2D grammar-based pattern recognition theory in the context of engineering drawings analysis since the late 1970s. His theory was explicated in the 10th chapter of the monograph [17] in English for the first time.

The first author of this paper studied independently the theoretical limits of 2D grammars [14] and proved them to be rather restrictive.

The main motivation of the authors of the reported work is to discover to what extent the 2D grammars are applicable to practical image analysis.

This paper provides insight into an ongoing work on a pilot study aiming at off-line recognition of mathematical formulae. We have chosen this application domain because there is a clear structure in formulae and works of others exist which can be used for comparison.

Let us categorize the approaches to mathematical formulae recognition along two directions:

– *on-line* recognition (the timing of the pen strokes is available) versus *off-line* recognition (only an image is available).

– *printed* versus *hand-written* formulae.

We deal with off-line recognition of hand-written formulae in this contribution. Of course, the approach can be also applied to printed formulae.

2 State-of-the-Art

Formulae recognition has been a widely studied task. Several approaches from pattern recognition were adopted as well as new methods were motivated and designed by this particular task. A taxonomy of the methods is given in [3]. There are only a few commercial products performing formulae recognition. The most prominent is probably xMathJournal software [19] for Tablet PCs which uses on-line recognition. This software serves as a sophisticated calculator allowing inputs to be written by hand.

Most of the known methods follow the following two-phase procedure:

– Detection of individual symbols by image segmentation and labelling symbols using pattern recognition techniques.
– Structural analysis of relations among labelled symbols.

Classical approaches known from Optical Character Recognition were adopted to perform the symbols recognition phase. Images are segmented and a classifier is used to assign symbols to segments. There are also works performing symbol detection, segmentation and labelling during a single simultaneous process using Hidden Markov Models [18].

Formalisms related to structural analysis include geometric grammars, graph grammars, finding a minimal spanning tree, etc. [7,12,5].

Criticism of the commonly used methods concerns the image segmentation which is done without any knowledge of the formulae structure. It is hardly possible to recover from errors made during symbol segmentation phase. There have been attempts to employ additional error corrections schemes. However, this postprocessing corrections do not fit naturally into the pattern recognition process.

The approach based on the two-dimensional context-free (2D CF) grammars and a general structural construction tries to solve this problem [17]. The group from Kiev has applied their approach to images of musical scores [16] or electrical circuits [11]. Let us note that the notion of 2D CF grammars has appeared in works of other authors, e.g., [13].

3 Theory of Two-dimensional Languages

The theory of two-dimensional languages studies generalizations of formal languages to two dimensions. These generalizations can be done in many different ways. Automata working over a two-dimensional tape were firstly introduced by M. Blum and C. Hewitt, already in 1967. Since then, several formal models recognizing or generating two-dimensional objects have been proposed in the literature.

The most common two-dimensional object is a *picture* which is a matrix of symbols taken from a finite alphabet Σ. The set of all pictures over Σ is denoted by Σ^{**}. Each subset $L \subseteq \Sigma^{**}$ is called a *two-dimensional language*. Note that it is also possible to consider objects of more general shapes, e.g. connected arrays, but we will work only with pictures.

One of the important tasks the theory of two-dimensional languages deals with is to search for suitable generalizations of the Chomsky hierarchy, especially of its first two levels, i.e. regular and context-free languages. Several formalisms were adopted or developed to fulfil this task, however, the more complex topology of pictures causes that the properties of the proposed classes usually differ to those known from regular, resp. context-free languages, despite the fact that the classes are defined via models resembling finite-state automata, resp. context-free grammars.

In the following sections, we describe two models of 2D finite-state devices and a 2D generalization of CF grammars. We also list basic properties of recognized (generated) languages.

3.1 Finite-state Devices

A *two-dimensional finite-state automaton* (2FSA) [15] is a natural generalization of the one-dimensional two-way automaton (which of recognition power equals the power of 1D finite-state automaton). It is equipped by a two-dimensional tape and allowed to move the head in four directions – left, right, up and down. This is the reason why it is also called a *four-way automaton* by some authors.

Let 2DFSA denote a deterministic 2FSA and $L(2FSA)$, resp. $L(2DFSA)$ the class of 2D languages recognizable by a 2FSA, resp. 2DFSA. The classes of recognized languages are characterized by the following facts:

- $L(2DFSA)$ is a proper subset of $L(2FSA)$.
- $L(2FSA)$ is not closed under concatenation (neither row or column), complement and projection.
- The emptiness problem is not decidable even for 2DFSA's.

Another kind of a finite-state device, so called *two-dimensional on-line tessellation automaton* (2OTA), was introduced by K. Inoue and A. Nakamura [9] in 1977. It is a kind of a bounded 2D cellular automaton. In comparison to the cellular automata, computations are performed in a restricted way – cells do not make transitions at every time-step, but rather a 'transition wave' passes once diagonally across them. Each cell changes its state depending on two neighbors – the top one and the left one. The result of a computation is determined by the state the bottom-right cell finishes in.

Again, let 2DOTA denote a deterministic 2OTA and $L(2OTA)$, resp. $L(2DOTA)$ be the classes of languages recognizable by the models. The most important results on the automata follow:

- $L(2DOTA)$ is a proper subset of $L(2OTA)$.
- $L(2OTA)$ is closed under row and column concatenation, union and intersection.
- $L(OTA)$ is not closed under complement while $L(DOTA)$ is closed under complement.
- $L(DOTA)$ and $L(FSA)$ are incomparable.
- $L(DFSA)$ is a proper subset of $L(DOTA)$.

D. Giammarresi and A. Restivo [8] present the class of languages recognizable by this device as the ground level class of the two-dimensional theory, prior to languages recognizable by two-dimensional finite-state automata. They argue that the proposed class fulfills more natural requirements on such a generalization. Moreover, it is possible to use several formalisms to define the class. Except tessellation automata, they include tiling systems or monadic second order logic, thus the definition is robust as in the case of regular languages. On the other hand, 2OTA's are quite strong, since some NP-complete languages can be recognized by them. This result speaks against the promotion of the class to the 2D ground level class.

3.2 Two-dimensional Context-free Grammars

In this section we present a proposal of 2D CF grammars and results on them as they were given in [14]. The grammars are a generalization of 2D CF grammars introduced in [17].

Let $[a_{i,j}]_{m,n}$ denote the matrix

$$\begin{matrix} a_{1,1} & \dots & a_{1,n} \\ \vdots & \ddots & \vdots \\ a_{m,1} & \dots & a_{m,n} \end{matrix}$$

For $P \in \Sigma^{**}$, let rows(P), resp. cols(P) denote the *number of rows*, resp. *columns* of P. We consider there is the only picture consisting of 0 rows and 0 columns. This picture, denoted Λ, is called the *empty picture*, analogously to the empty word λ.

We define two binary operations, the *row* and *column concatenation*. Let $A = [a_{i,j}]_{k,l}$ and $B = [b_{i,j}]_{m,n}$ be non-empty pictures over Σ. The column concatenation $A \varobar B$, resp. row concatenation $A \ominus B$ is defined if $k = m$, resp. $l = n$. The products of the operations are given by the following schemes:

$$A \varobar B = \begin{matrix} a_{1,1} & \dots & a_{1,l} & b_{1,1} & \dots & b_{1,n} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ a_{k,1} & \dots & a_{k,l} & b_{m,1} & \dots & b_{m,n} \end{matrix} \qquad A \ominus B = \begin{matrix} a_{1,1} & \dots & a_{1,l} \\ \vdots & \ddots & \vdots \\ a_{k,1} & \dots & a_{k,l} \\ b_{1,1} & \dots & b_{1,n} \\ \vdots & \ddots & \vdots \\ b_{m,1} & \dots & b_{m,n} \end{matrix}$$

It means $A \varobar B = [c_{i,j}]_{k,l+n}$, where

$$c_{i,j} = \begin{cases} a_{i,j} & \text{if } j \le l \\ b_{i,j-l} & \text{otherwise} \end{cases}$$

and similarly, $A \ominus B = [d_{i,j}]_{k+m,l}$, where

$$d_{i,j} = \begin{cases} a_{i,j} & \text{if } i \le k \\ b_{i-k,j} & \text{otherwise} \end{cases}$$

The *generalized concatenation* is an unary operation $\bigoplus$ defined on a set of matrixes of elements that are pictures over some alphabet. For $i = 1, \dots, m$; $j = 1, \dots, n$, let $P_{i,j}$ be pictures over Σ. $\bigoplus[P_{i,j}]_{m,n}$ is defined if

$$\begin{aligned} \forall i \in \{1, \dots, m\} \ \text{rows}(P_{i,1}) = \text{rows}(P_{i,2}) = \dots = \text{rows}(P_{i,n}) \\ \forall j \in \{1, \dots, n\} \ \text{cols}(P_{1,j}) = \text{cols}(P_{2,j}) = \dots = \text{cols}(P_{m,j}) \end{aligned}$$

The result of the operation is $P_1 \ominus P_2 \ominus \dots \ominus P_m$, where each $P_k = P_{k,1} \varobar P_{k,2} \varobar \dots \varobar P_{k,n}$. See Figure 1 for an illustrative example.

$P_{1,1}$	$P_{1,2}$	$P_{1,3}$
$P_{2,1}$	$P_{2,2}$	$P_{2,3}$

Figure 1. Scheme for the result of $\bigoplus[P_{i,j}]_{2,3}$ operation

Definition 1. A two-dimensional context-free grammar *(2CFG) is a tuple* $(V_N, V_T, S_0, \mathcal{P})$, *where*

- V_N *is a finite set of nonterminals*
- V_T *is a finite set of terminals*
- $S_0 \in V_N$ *is the initial nonterminal*
- $\mathcal{P}$ *is a finite set of productions of the form* $N \to W$, *where* $N \in V_N$ *and* $W \in (V_N \cup V_T)^{**} \setminus \{\Lambda\}$. *In addition,* $\mathcal{P}$ *may contain production* $S_0 \to \Lambda$. *In this case, no production in* $\mathcal{P}$ *contains* S_0 *as a part of its right-hand side.*

Definition 2. *Let* $G = (V_N, V_T, S_0, \mathcal{P})$ *be a 2CFG. We define a picture language* $L(G, N)$ *over* V_T *for every* $N \in V_N$. *The definition is given by the following recurrent rules:*

I) If $N \to W$ *is a production in* $\mathcal{P}$ *and* $W \in {V_T}^{**}$, *then* W *is in* $L(G, N)$.
II) Let $N \to [A_{i,j}]_{m,n}$ *be a production in* $\mathcal{P}$, *different to* $S_0 \to \Lambda$, *and* $P_{i,j}$ *(*$i = 1, \ldots, n$; $j = 1, \ldots, m$*) be pictures such that*
- *if* $A_{i,j}$ *is a terminal, then* $P_{i,j} = A_{i,j}$
- *if* $A_{i,j}$ *is a nonterminal, then* $P_{i,j} \in L(G, A_{i,j})$

Then, if $\bigoplus[P_{i,j}]_{m,n}$ *is defined,* $\bigoplus[P_{i,j}]_{m,n}$ *is in* $L(G, N)$.

The set $L(G, N)$ *consists of pictures that can be obtained by applying a finite sequence of rules I and II. The language* $L(G)$ *generated by the grammar G is defined to be* $L(G) = L(G, S_0)$.

To illustrate the presented definition, let us show a simple example of a $2CFG$ that generates the set of all non-empty square pictures over $\Sigma = \{a\}$.

Example 3. Let $G = (V_N, V_T, S_0, \mathcal{P})$ be a $2CFG$, where $V_T = \{a\}$, $V_N = \{V, H, S_0\}$ and $\mathcal{P}$ consists of the following productions:

$$1)\quad H \to a\,, \qquad 2)\quad H \to a\,H\,, \qquad 3)\quad V \to a\,, \qquad 4)\quad V \to \begin{matrix} a \\ V \end{matrix}\,,$$

$$5)\quad S_0 \to a\,, \qquad 6)\quad S_0 \to \begin{matrix} a & H \\ V & S_0 \end{matrix}\,.$$

Productions 1), 2) are one-dimensional, thus it should be clear that $L(G, H)$ contains exactly all non-empty rows of a's. And really, by applying rule I) on production 1), we have $a \in L(G, H)$. Furthermore, if $a^k \in L(G, H)$, then rule II) applied on production 2) gives $a^{k+1} \in L(G, H)$. Similarly, $L(G, V)$ contains non-empty columns of a's.

By applying rule I) on production 5), the square 1×1 is generated by G. Since $a \in L(G, S_0) \cap L(G, H) \cap L(G, V)$, rule II) applied on production 6) gives that the square 2×2 is also in $L(G, S_0)$. The row, resp. column of length 2 is generated by

H, resp. V, thus rule II) can be applied again to produce the square 3×3, etc. By induction on the size, we can show that each non-empty square picture over $\{a\}$ can be generated and that there is no way to generate any non-square picture.

If we restrict right-hand sides of productions to be composed of at most two elements we obtain grammars from [17]. Let 2SCFG denote such a grammar and let us summarize the allowed types of productions:

P1. (column concatenation) $N \to A\,B$

P2. (row concatenation) $N \to \begin{matrix} A \\ B \end{matrix}$

P3. (renaming) $N \to A$

N is a nonterminal, A and B are terminals or nonterminals.

A characterization of 2CFG's follows:

- $L(2CFG)$ is not comparable to $L(2FSA)$, neither to $L(2OTA)$.
- $L(2SCFG)$ is a proper subset of $L(2CFG)$.
- There is no analogy to the Chomsky normal form of productions.
- Time complexity of recognition depends on size of production's right-hand sides.
- The emptiness problem is not decidable.

We define two languages to demonstrate the incomparability between $L(2FSA)$ and $L(2CFG)$.

1. $L_1 = \{P \mid P \in \{a\}^{**} \wedge \mathrm{cols}(P) = \mathrm{rows}^2(P)\}$
2. Let L_2 be a language over $\{a, b\}$, consisting of square pictures, where each row and each column contains exactly one symbol b.

L_1 can be generated by a 2CFG, but it cannot be recognized by any 2FSA. On the other hand, L_2 is recognizable by a 2FSA (even by a 2DFSA), but it cannot be generated by any 2CFG.

The well known Cocke-Younger-Kasami algorithm [10,20,1] for recognition of languages generated by one-dimensional context-free grammars can be generalized on 2SCFG's [17]. Time complexity of the algorithm is

$$\mathcal{O}\left(m^2 n^2 (m + n)\right) ,$$

where m, resp. n denote the number of rows, resp. columns of the picture. It is also possible to generalize the algorithm on languages generated by 2CFG's, but time complexity depends on sizes of production's right-hand sides in this case. Let $G = (V_T, V_N, S_0, \mathcal{P})$ be a 2CFG and

$$p = \mathbf{max}\left\{\mathrm{rows}(W) \mid N \to W \in \mathcal{P}\right\} , \qquad q = \mathbf{max}\left\{\mathrm{cols}(W) \mid N \to W \in \mathcal{P}\right\} .$$

Now, time complexity of the algorithm is

$$\mathcal{O}\left(m^{p+1} n^{q+1}\right) .$$

4 General Idea of Structural Construction

Principles related to the structural construction are explained in this section. The recognition process works with regions of the input image. A *region* is a connected set of pixels having some common property. The region is assigned a *label* determining which structure was recognized to be represented by the region (e.g., a fraction line as a part of a formula) and also a penalty giving the cost of derivation of the region. We consider two finite sets of labels: V_T is a set of *terminals* and V_N is a set of *nonterminals*. There is also one distinguished label $S_0 \in V_N$ corresponding to the whole structure we want to recognize.

At the beginning, there are so called terminal regions only, labelled by terminals. These regions can correspond to single pixels of the input image or to regions detected by an external tool. Usually, the property of regions is that they decompose an image into disjoint components. We relax this requirement and allow that regions to share some pixels.

A set of *rules* specifying how labelled regions can be combined to produce larger regions (their unions) is defined. A rule $N \to N_1, \dots, N_k$ is interpreted as follows. If there are regions $R_1, \dots, R_k$ labelled by $N_1, \dots, N_k$, their sizes and positions fulfil some constraints connected to the rule then their union $R = \bigcup_{i=1}^{k} R_i$ with the label N can be derived. The *penalty* of this derivation is computed from penalties of particular regions R_i. The rules are applied during an iterative process to derive larger and larger regions. The process ends when all possible regions have been derived. If the whole image was assigned by S_0 and the penalty of related derivation fits into some limit then the desired structure in the image was successfully recognized.

The described recognition process is too general and would be highly complex in time and space. Because of this, we need to seek some convenient unifications of rules and region shapes that will lead to an acceptable implementation. The formalisms resulting in these unifications can be based on 2D CF grammars we have already presented. Namely 2SCFG's are considered in [17]. In this case, permitted regions are only rectangles to suppress the complexity of derivations. Rules (or productions – to be consistent with the grammar terminology) are of types P1, P2 and P3.

Two observations can be made for the usage of the grammars: The complexity is still high and the constructs supported by the grammars are not rich enough to model relationships among symbols in mathematical formulae. We will address these two issues in the following section by introducing a suitable extension of the grammars.

5 Formulae Recognition Based on the Structural Construction

The main ideas taken from the structural construction we follow in our approach can be expressed in the following way:

Perform 'rough segmentation' of the input image. For each possible elementary symbol, find all occurrences of it and let the structural analysis decide, structure of which formula fits the input image best and how does the image segmentation looks like.

5.1 Specification and Goals of the Pilot Study

The pilot study was expected to support elements and constructs as numbers, variables, brackets, subscripts, superscripts, basic unary and binary operators, power to operations, fractions, sums, integrals and square roots.

We consider the inputs to be black and white images, however, the used method can be easily adopted to gray-scale images as well since it is general enough and does not depend on this assumption.

Our main goal was to investigate whether the structural construction can be successfully applied to off-line formulae recognition. In particular, we were interested how the method can deal with the following situations.

a) Symbols touching vertically or horizontally.
b) Symbols split into several components.
c) Ambiguities.
d) Misplaced symbols.
e) Noise.

Examples of these situations are given in Figure 2. Cases a) and b) demostrate standard formulae. Case c) illustrates a fraction line and a minus sign represented by the same image, the meaning is being given by the context. And finally, case d) includes an additional symbol A that is by mistake placed into the formula. We require our method to exclude such a misplaced symbol and recognize the formula composed of the other symbols.

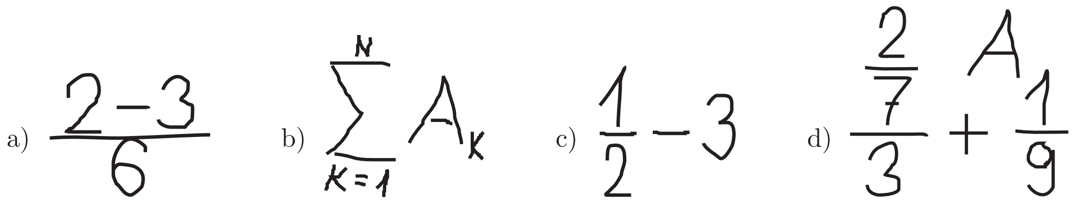


Figure 2. Corner cases we would like to handle

5.2 Extension of 2D Context-free Grammars

We use an extension of 2D CF grammars to model relationships among mathematical symbols. In this section, we give a description of the productions form and their application.

Let $N \to A \oplus B$ denote a production of our 2D CF grammar extension. The interpretation is similar to the interpretations of productions $N \to A|B$ and $N \to \frac{A}{B}$, regions labelled A and B can be united, producing a region labelled N. But this time we do not require the regions to touch each other. Permitted mutual positions of the regions are defined by a constraint that is connected to the *production*. The form of the constraint is depicted in Figure 3.

R and S are two regions in the image, F is a *feature point* of S (the center of left border in this particular case). C is a dashed rectangle the size and position of which is given relative to the size and position of R. It represents the mentioned constraint. The considered production can be applied when the following conditions are fulfilled:

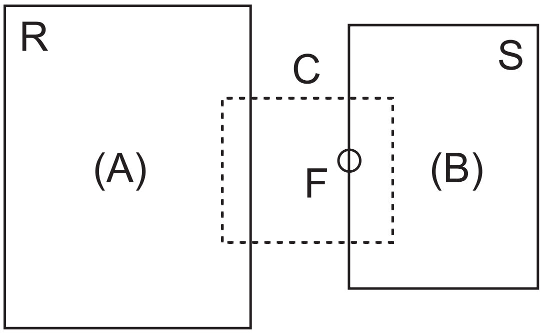


Figure 3. General production scheme of $N \rightarrow A \oplus B$

- R, resp. S can be labelled by A, resp. B.
- feature point F is located inside C.

The resulting region is the smallest rectangle containing R and S. Feature points are some specific points of a region. We usually use corners and centers of bounding edges. We also compute baseline (when a new region is derived) and take the intersection with left or right bounding edge.

Figure 4 shows usage of a production to model 'power to' relationship where the constraint is defined for the bottom-left corner of the region storing the symbol four.

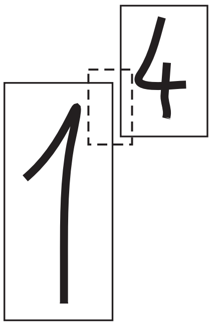


Figure 4. Example of the production usage

The penalty of the derived region is computed based on the following factors:

- Used production.
- Penalties of R and S.
- Percentage of black pixels in the new region that are neither in R nor S.
- Relative sizes and positions of R and S.

To finish the description we will make three remarks. First, note that R and S can overlap. Second, we have defined the constraint on the location of S, assuming we know the location and size of R. In our structural analysis, we will also need the opposite case, i.e., to look for S while knowing R. We slightly extend the production form by attaching one more constraint C' of the mentioned meaning. And finally, in some cases of formulae constructs, it is more convenient to compose a new region from three regions instead of two (consider, e.g., a variable with a subscript and superscript). This can be easily supported by introducing productions of the from $N \rightarrow A_1 \oplus A_2 \oplus A_3$.

5.3 Architecture

Our software consists of two independent layers. The first layer performs *terminals detection*, while the second one is responsible for *structural analysis*. An external OCR tool is used within the first layer. Structural analysis is driven by a grammar defining supported mathematical formulae. The parsing algorithm is of a general nature and it can be used to recognize another types of structures if supported by a proper grammar.

5.4 Terminals Detection

We have developed an OCR tool for classification of image regions. The tool is based on a simple extraction of features from the input picture. The k-nearest neighbor classifier is implemented to classify the extracted vectors.

We have tested two methods for terminal symbols detection. The first method works with rectangular scanning windows of some specific sizes. Each of the windows is being moved trough the input image. Whenever it is in a new position, its content is evaluated by the OCR tool (provided that the number of black pixels exceeds some defined threshold). The result of the evaluation is a set of terminals assigned by a penalty, where the penalty corresponds to the belief that the scanning window stores a particular symbol. Only the terminals with a sufficiently low penalty are included in the result. For example, we have a window to detect subscripts and another window to detect variables and numbers. Sizes of the windows are determined by expected sizes of symbols in the formulae which means the method requires tuning for typical inputs.

It would be ideal in theory if we could use views of all sizes to scan the image, however, this approach is time expensive. Limiting the sizes of used views results in an acceptable performance but can miss some symbols in the image when their size differs from the expectation. Because of this we have tested the second method based on preprocessing the content of the image by computing connected components. Selection of views that are evaluated by the OCR tool is driven by these components. The bounding rectangles of the following areas are chosen:

- the components themselves,
- divisions of the components – up to two splitting horizontal and vertical points are considered,
- combinations of neighboring components.

5.5 Parsing Algorithm

We describe the algorithm that is used for the structural analysis phase. To simplify the description, we do not explain how information needed to track feature points and derivation trees is being updated. Just note that whenever a region R is labelled by N, we record by which production this was derived.

1. Let $\mathcal{R}$ be a list of triples (R, N, p), where R is a region, N label assigned to R and p penalty of this assignment. Initialize $\mathcal{R}$ by results of the terminals detection phase.
2. Iterate through $\mathcal{R}$. Let $(R, L, p) \in \mathcal{R}$ be the current element. For each production $N \to A \oplus B$ such that $L = A$ or $L = B$, take the rectangular area C defined by the constraint of the production and find subset $\mathcal{S} \subseteq \mathcal{R}$, where for each $(R', L', p') \in \mathcal{S}$

the production can be applied on R and R'. Let $\overline{R}$ be the derived region, $\overline{p}$ penalty of the derivation. If $\overline{p}$ is greater than some threshold then continue by the next iteration, otherwise check whether $\overline{R}$ has been already labelled by N. If not then append $(\overline{R}, N, \overline{p})$ at the end of $\mathcal{R}$. If there is already $(\overline{R}, N, p_2)$ in $\mathcal{R}$ and $\overline{p} < p_2$ then remove $(\overline{R}, N, p_2)$ from $\mathcal{R}$ and append $(\overline{R}, N, \overline{p})$, otherwise ignore the new derivation.

3. A formula is successfully recognized when the bounding rectangle of the input is labelled by S_0. If not then we can look for the largest region in $\mathcal{R}$ labelled by S_0.

To make the algorithm fast, we use a data structure storing points of a plane, allowing it to effectively evaluate queries of the type: for a rectangle, return the points that are located inside the rectangle (so called *orthogonal range searching*). A suitable data structure allowing to search in time $\mathcal{O}(\log n)$ can be constructed [2].

Our conclusions on time complexity are based on empirical data, we do not give an exact formula since it depends on many factors, including the number of the terminals detected during the first phase. Compared to the generalized Cocke-Younger-Kasami algorithm, time complexity is lower because the algorithm does not process all rectangles in the input. It would be possible to derive some upper bound on time, but it does not give a good idea about expected time. Instead of doing it, we rather discuss the most problematic case in the section regarding results.

6 Results

We have implemented the pilot study in Java. It includes an user interface allowing to browse formulae images, run the recognition on them and display results. Except the results, the interface also provides information helping to understand and tune the process of structural analysis. For example, it is possible to query for all regions labelled by a specific nonterminal, for penalties of related derivations, etc.

The implementation has been tested on over 200 handwritten formulae. We can conclude that after tuning the grammar there were no problems with correctness of the structural analysis. The problems we have encountered are connected mainly to the terminals detection phase.

We have faced some limitations when working with rectangular regions. Not all symbols in a formula can be separated by rectangles. Figure 5 a) shows one of the simplest examples. The bounding box of symbol r contains a part of the subscript. It has an impact on recognition of r, which is assigned a bigger penalty in this case. In general, the recognition does not give good results, when the bounding boxes overlap too much. This is not usually case of printed formulae.

We were also forced to compose some elementary symbol from more components due to the mentioned limitation. A typical example is a square root which we consider to be formed of two parts (the square root argument can be treated as an additional, third part) – see Figure 5 b).

Other problems are connected to fraction lines. Continual subparts of a fraction line are also recognized as fraction lines. This leads to a large number of terminal symbols and possible combinations among them to be checked during the structural analysis. Figure 5 c) shows an image on which the problem starts to be visible. Recognition of this formulae takes about 20 seconds and grows fast for larger fractions (note that the recognition of each formula depicted in Figure 2 takes up to 2 seconds). We have implemented a preprocessing of detected fraction lines that reduces their

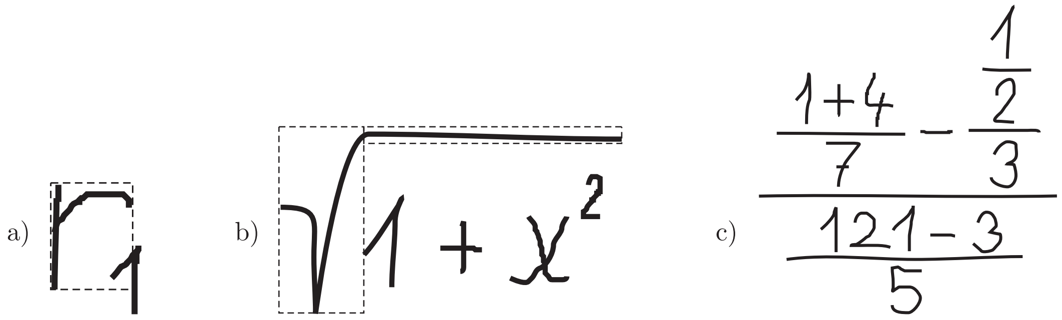


Figure 5. a) Bounding box of r contains a part of the subscript. b) Square root symbol composed of two parts. c) Formula with several fraction lines.

number. It would be also possible to implement a special method (separated from the OCR tool) for fraction lines detection.

The last problem we have encountered was the accuracy of the used OCR tool that was not as high as it should be, but this problem can be solved by choosing a better tool.

7 Conclusions

We have showed that the method of structural construction can be applied for off-line mathematical formulae recognition. Our main contribution to the area of formulae recognition are the following achievements:

- Segmentation of the image is done during structural analysis (no error corrections are needed). We took advantage of the rich formula structure which allows this approach.
- Structural analysis is robust. It is penalty oriented and searches for the formula structure that best matches the input image. It can easily deal with noises, including misplaced symbols.
- We have designed an extension of 2D CF grammars powerful enough to express the formulae structure. It can be also effectively parsed (thanks to constraints defined via rectangles and the usage of data structures for orthogonal range searching).

We would like to focus more on printed formulae in the upcoming work. Our future plans include the usage of learning methods. Provided that a sufficiently large set of formulae is collected, the methods can be applied on learning etalons of terminal symbols and productions parameters. The learned etalons can improve the terminals detection phase, while the learned productions parameters will improve tuning of the grammar for a concrete typesetting style of formulae (so far we have tuned these parameters manually).

References

1. A. Aho and J. Ullman: *The theory of parsing, translation, and compiling*, vol. 1 – *Parsing*, Prentice-Hall, Englewood Cliff, New Jersey, 1971.
2. M. Berg, O. Schwarzkopf, M. Kreveld, and M. Overmars: *Computational Geometry: Algorithms and Applications*, Springer-Verlag, 2000.
3. K.-F. Chan and D.-Y. Yeung: *Mathematical expression recognition: a survey.* IJDAR, 3(1) 2000, pp. 3–15.
4. N. Chomsky: *Syntactic Structures*, Mouton and Co, The Hague, 1957.
5. Y. Eto and M. Suzuki: *Mathematical formula recognition using virtual network*, in Proceedings of the ICDAR 2001, 2001, pp. 762–767.
6. K. Fu: *Syntactic Methods in Pattern Recognition*, Academic Press, New York, 1974.
7. P. Garcia and B. Coüasnon: *Using a generic document recognition method for mathematical formulae recognition*, in GREC '01: Selected Papers from the Fourth International Workshop on Graphics Recognition Algorithms and Applications, D. Blostein and Y.-B. Kwon, eds., vol. 2390 of LNCS, Berlin, Germany, 2002, Springer-Verlag, pp. 236–244.
8. D. Giammarresi and A. Restivo: *Recognizable picture languages*, in Int. J. of Pattern Recognition and Artificial Inteligence 6(2-3), 1992, pp. 32–45.
9. K. Inoue and A. Nakamura: *Some properties of two-dimensional on-line tessellation acceptors*, in Information Sciences, vol. 13, 1977, pp. 95–121.
10. T. Kasami: *An efficient recognition and syntax analysis algorithm for context-free languages*, Scientific report AFCLR-65-758, Air Force Cambridge Research Laboratory, Bedford, Mass., USA, 1965.
11. V. Kiyko: *Recognition of objects in images of paper based line drawings*, in Third International Conference on Document Analysis and Recognition, Montreal, 1995, pp. 970–973.
12. S. Lavirotte and L. Pottier: *Mathematical formula recognition using graph grammar*, in Proceedings of the SPIE 1998, vol. 3305, San Jose, CA, 1998, pp. 44–52.
13. E. Miller and P. Viola: *Ambiguity and constraint in mathematical expression recognition*, in AAAI/IAAI, 1998, pp. 784–791.
14. D. Průša: *Two-dimensional Languages*, PhD thesis, Faculty of Mathematics and Physics, Charles University, Prague, 2004.
15. A. Rosenfeld: *Picture Languages - Formal Models of Picture Recognition*, Academic Press, New York, 1979.
16. B. Savchynsky, M. Schlesinger, and M. Anochina: *Parsing and recognition of printed notes*, in Proceedings of the conference Control Systems and Computers, Kiev, Ukraine, 2003, pp. 30–38, in Russian, preprint in English available.
17. M. Schlesinger and V. Hlaváč: *Ten lectures on statistical and structural pattern recognition*, vol. 24 of Computational Imaging and Vision, Kluwer Academic Publishers, Dordrecht, The Netherlands, 2002.
18. H. Winkler and M. Lang: *Online symbol segmentation and recognition in handwritten mathematical expressions*, in Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing, vol. 4, Munich, Germany, 1997, pp. 3377–3380.
19. *Mathjournal 2.0*: a software for formulae recognition from the xThink company, 2006, http://www.xthink.com/MathJournal.html.
20. D. Younger: *Recognition of context-free languages in time* n^3. Information and Control, 10 1967, pp. 189–208.

A Concurrent Specification of Brzozowski's DFA Construction Algorithm

Tinus Strauss, Derrick G. Kourie, and Bruce W. Watson

FASTAR Research group
Department of Computer Science, University of Pretoria,
Pretoria, South Africa
{tstrauss, dkourie, bwatson}@cs.up.ac.za

Abstract. In this paper two concurrent versions of Brzozowski's deterministic finite automaton (DFA) construction algorithm are developed from first principles, the one being a slight refinement of the other. We rely on Hoare's CSP as our notation.

The specifications that are proposed of the Brzozowski algorithm are in terms of the concurrent composition of a number of top-level processes, each participating process itself composed of several other concurrent processes. After considering a number of alternatives, this particular overall architectural structure seemed like a natural and elegant mapping from the sequential algorithm's structure.

While we have carefully argued the reasons for constructing the concurrent versions as proposed in the paper, there are of course, a large range of alternative design choices that could be made. There might also be scope for a more fine-grained approach to updating sets or checking for similarity of regular expressions. At this stage, we have chosen to abstract away from these considerations, and leave their exploration to future research.

Keywords: automaton construction, concurrency, concurrent sequential processes, regular expressions

1 Introduction

This research is inspired by two contemporary trends. On the one hand, finite automaton technology is being applied to ever-larger applications. On the other hand, hardware is tending towards ever-increasing support for concurrent processing. Chip multiprocessors [6], for example, implement multiple CPU cores on a single die. Additionally, scale-out systems [1] – collections of interconnected low-cost computers working as a single entity – also provide parallel processing facilities. These hardware developments present the challenging task of producing quality concurrent software [5,7,8].

It seems, though, that relatively little thought has been given in the finite automaton research community to developing concurrent versions of the various sequential algorithms that are widely in use. The only parallel algorithm that converts a regular expression into an automaton of which we are aware has been described by Ziadi and Champarnaud [9].

Here, two concurrent versions of Brzozowski's deterministic finite automaton (DFA) construction algorithm are developed from first principles, the one being a slight refinement of the other. This will be the theme of section 3. However, before developing the concurrent algorithm, we provide a brief overview of the sequential version in section 2. A brief reflection on this work is given in section 6.

2 Sequential Algorithm

Brzozowski's DFA construction algorithm [2] employs the notion of derivatives of regular expressions to construct a DFA. The algorithm takes a regular expression E as input and constructs an automaton which accepts the language represented by E.

The automaton is represented using the normal five-tuple notation $(D, \Sigma, \delta, S, F)$ where D is the set of states; Σ the alphabet; δ the transition relation mapping a state and an alphabet symbol to a state; and $S, F \subseteq D$ are the start and final states, respectively. $\mathcal{L}$ is an overloaded function giving the language of a finite automaton or a regular expression.

Since each regular expression in D is represented by a node in the automaton, we will sometimes refer to an element in a set as a regular expression, and at other times we will refer to it as a node.

The well-known sequential version of the algorithm is given in Dijkstra's guarded command language in figure 1. The notation assumes that the set operations ensure "uniqueness" of the elements at the level of similarity, i.e. $a \in A$ implies that there is no $b \in A$ such that a and b are similar regular expressions.

func $Brz(E, \Sigma) \rightarrow$
  $\delta, S, F := \varnothing, \{E\}, \varnothing;$
  $D, T := \varnothing, S;$

  do $(T \neq \varnothing) \rightarrow$
    let q be some state such that $q \in T$
    $D, T := D \cup \{q\}, T \setminus \{q\};$
    for $(i : \Sigma) \rightarrow$
      $d := \frac{d}{di}q;$
      if $d \notin (D \cup T) \rightarrow T := T \cup \{d\}$
      $[\!]$ $d \in (D \cup T) \rightarrow$ **skip**
      fi;
      $\delta(q, i) := d;$
    rof;
    if $\varepsilon \in \mathcal{L}(q) \rightarrow F := F \cup \{q\}$
    $[\!]$ $\varepsilon \notin \mathcal{L}(q) \rightarrow$ **skip**
    fi;
  od;
  return $(D, \Sigma, \delta, S, F);$

Figure 1. Brzozowski's DFA construction algorithm

Walking through this sequential algorithm, it will be seen that it relies on two sets: a set T containing the nodes (regular expressions) for which derivatives need to be calculated; and another set D containing the nodes for which derivatives have been found already.

The algorithm then works through all the nodes $q \in T$, finding derivatives with respect to all the alphabet symbols and depositing these nodes (regular expressions) into T in those cases where no equivalent regular expression has already been deposited into $T \cup D$. Each node, q, dealt with in this fashion from T is then removed from T and added into D.

In each iteration of the inner `for` loop (i.e. for each alphabet symbol), the δ relation is updated to contain the mapping from node q to its derivative with respect to the relevant alphabet symbol.

Finally if q is nullable, it is added to the set of final states F.

In the forthcoming sections we present a concurrent specification of the algorithm in which we attempt to allow as much concurrency as possible.

3 Concurrent specification

We present here an approach to parallelising the algorithm. Of the many process algebras have been developed to concisely and accurately model concurrent systems, we have selected CSP [4,3] as a fairly simple and easy to use notation. It is arguably better known and more widely used than most other process algebras. Below, we provide a brief introduction to the CSP operators that are used, and indicate some of the assumptions we make in regard to atomicity of operations.

3.1 Introductory Remarks

CSP is concerned with specifying a system of concurrent sequential processes (hence the CSP acronym) in terms of sequences of events, called traces. In fact, the semantics of a concurrent system is seen as being precisely described by the set of all possible traces that characterise such as system. A fundamental assumption is that events are instantaneous and atomic—i.e. they cannot occur concurrently. Various operators are available to describe the sequence in which events may occur, as well as to connect processes. Table 1 briefly outlines the main operators used in this article.

$a \to P$	event a then process Q
$a \to P \vert b \to Q$	a then P choice b then Q
$x : A \to P(x)$	choice of x from set A then $P(x)$
$P \parallel Q$	P in parallel with Q
	Synchronize on common events in the alphabet of P and Q
$b!e$	on channel b output event e
$b?x$	from channel b input to variable x
$P \nleftarrow C \nrightarrow Q$	if C then process P else process Q
$P;Q$	process P followed by process Q
$P \square Q$	process P choice process Q

Table 1. Selected CSP Notation

Full details of the operator semantics and laws for their manipulation are available in [4,3]. Note that *SKIP* designates a special process that engages in no further event, but that simply terminates successfully. Parallel synchronization of processes means that if $A \cap B \neq \emptyset$, then process $(x : A \to P(x)) \parallel (y : B \to Q(y))$ engages in some nondeterministically chosen event $z \in A \cap B$ and then behaves as the process $P(z) \parallel Q(z)$. However, if $A \cap B = \emptyset$ then deadlock results. A special case of such parallel synchronization is the process $(b!e \to P) \parallel (b?x \to Q(x))$. This should be viewed a process that engages in the event $b.e$ and thereafter behaves as the process $P \parallel Q(e)$.

3.2 Atomicity Assumptions

In deploying CSP, we have made the following assumptions relating to atomic execution.

Firstly, if an event maps to a function call, then that function is assumed to be a sequence of code in the original sequential algorithm which runs uninterruptedly to completion on some processor.

Furthermore, in the interest of conciseness and without loss of generality, it will sometimes be convenient to subsume certain assignment operations of the sequential program into the actual parameter list of a process invocation. For example, instead of specifying some recursive parameterised process $P(D)$ as $P(D) = \cdots (D := D \cup \{q\}); P(D)$, we will regard the specification $P(D) = \cdots P(D \cup \{q\})$ as equivalent. This means that operations that are needed to compute the actual parameters for a process invocation are regarded as taking place atomically—i.e. they cannot be interrupted by any other process's activity.

Similarly, where the CSP syntax for a conditional is used, as in $P \not< C \not> Q$, it will be assumed that the computation of the condition, C, takes place atomically and prior to the activation of any first event possible in the constituent processes, P and Q. This is specifically the case where similarity between regular expressions as implied in the Boolean expression $d \notin (D \cup T)$ has to be computed.

These instances of atomic activity are highlighted, not because they deviate from CSP syntax, but because they represent potential opportunities for a more fine-grained specification of the algorithm than what will be proposed below. However, deeper consideration of whether such a more fine-grained specification would be desirable or possible was deemed to be outside the scope of this present endeavour.

4 The *BRZ* process

The specification that is proposed of the Brzozowski algorithm is in terms of the concurrent composition of three top-level processes, each participating process itself composed of several other concurrent processes. After considering a number of alternatives, this particular overall architectural structure seemed like a natural and elegant mapping from the sequential algorithm's structure.

The first of these three processes is called *OUTER*. It corresponds to the actions of the outer loop of the sequential program. Another process called *DERIVE* caters for the computation of derivatives in the inner loop of the sequential version. Finally, an *UPDATE* process caters for the determination of which derived regular expressions should be used to update the "to do" set T, and also for updating the transition function, δ. The concurrent specification of the Brzozowski algorithm is thus:

$$BRZ(D,T) = OUTER(D,T) \parallel DERIVE \parallel UPDATE$$

Note the sets D and T are required as parameters for the *OUTER* process, because they are explicitly altered within this process. However, we will assume that these sets are globally available for read-only purposes within the other two processes, *DERIVE* and *UPDATE*. It will be convenient to regard the alphabet Σ as well as the sets F

and δ as being a globally available to all sub-processes of the concurrent version of *BRZ*.

Furthermore, we assume that the first statement of the sequential algorithm—$\delta, S, F := \varnothing, \{E\}, \varnothing$;—takes place before the concurrent algorithm starts off. Given a regular expression, E, the concurrent process $BRZ(\varnothing, \{E\})$ is equivalent to the sequential algorithm $BRZ(\{E\}, \Sigma)$.

In the subsequent sections these constituent processes of *BRZ* are explored and defined in greater detail. Figure 2 provides a graphical representation of the structure of *BRZ*. However, it also includes a refinement to this model that incorporates buffers for greater efficiency. This refinement is described in subsection 5.2.

4.1 The *OUTER* process

This process corresponds to the iterations of the outer loop in the sequential algorithm, in that it selects the next q to be processed, and caters for the updating of the two sets T and D.

The *OUTER* process has these two sets as parameters. As in the sequential case, D contains all the regular expressions for which derivatives have been found and will become nodes in the automaton. T is the set of regular expressions for which derivatives are still to be found.

The process is responsible for extracting an arbitrary node from T and then passing it on to the *DERIVE* process. It also updates the sets D, T, and F.

The process is defined in terms of a choice between two sub-processes. This is indicated by the CSP process choice operator, $\square$. The first sub-process operand in the choice is initiated by engaging in an event that consists of selecting one of the regular expressions in T. The selected regular expression is called q. Thereafter, the *OUTER* process behaves as the parallel composition of two processes that take q as a parameter: *EXTRACT* and *FINAL*. This parallel composition has to run to completion before the *OUTER* process repeats, now with q added to D and removed from T.

Before considering the detail of the processes *EXTRACT* and *FINAL* that constitute the parallel composition, consider the second sub-process operand of the process choice operator in *OUTER*. It monitors a channel that is called *insert*, inputting a regular expression represented by the variable q from the channel whenever such an input becomes available. Thereafter *OUTER* repeats with q added to T. Again, we assume that this set union operation is atomic. This corresponds to the part in the sequential algorithm where the derivative d is added to T inside the inner loop.

$$\begin{aligned} OUTER(D,T) = {} & (q : T \rightarrow (EXTRACT(q) \parallel FINAL(q)); OUTER(D \cup \{q\}, T \setminus \{q\})) \\ & \square \\ & (insert?q \rightarrow OUTER(D, T \cup \{q\})) \end{aligned}$$

Now consider the two processes within *OUTER* which are to execute as a parallel composition: *EXTRACT* and *FINAL*.

We begin with *EXTRACT*. Our task here is to express the fact that its parameter q should be broadcast to a set of processes that will independently compute the derivative of q, each with respect to a different symbol in the alphabet. To this end, *EXTRACT* is regarded as the parallel composition of a set of processes, designated

$EXTRACT_i$ for each i in the alphabet Σ. Each $EXTRACT_i$ process passes its parameter, q, along its own channel, dIn_i, and then terminates successfully. The CSP specification to express the above is as follows:

$$EXTRACT(q) = \|_{i:\Sigma}\ EXTRACT_i(q) \quad \text{where}$$

$$EXTRACT_i(q) = (dIn_i!q \rightarrow SKIP)$$

As will be seen a little later, there is a $DERIVE_i$ process for each alphabet symbol i in Σ. Each of these processes will receive the outputted q on the associated dIn_i channel.

The *FINAL* process checks whether q is nullable and then adds q to the set of final states and then terminates successfully; otherwise *FINAL* terminates successfully without engaging in any action.

$$FINAL(q) = F := F \cup \{q\} \not\leq \varepsilon \in \mathcal{L}(q) \not\geq SKIP$$

Note that the set of events that take place in *EXTRACT* and *FINAL* are disjoint. The parallel composition of these two processes can therefore be described by the arbitrary interleaving of their respective event trace sets.

4.2 The *DERIVE* Process

The *DERIVE* process finds, in parallel, the derivatives of a regular expression with respect to all the symbols $i \in \Sigma$. The objective is to define *DERIVE* in such a way that each of its sub-processes can resume computing yet another derivative for a given alphabet symbol as soon as its task is complete, independently of the progress of its peer sub-processes. Here, a first order definition of *DERIVE* is given that does not fully meet this objective. This can be achieved by a simple refinement of the overall specification, as will be discussed later.

A sub-process that is designated $DERIVE_i$ receives input on channel dIn_i and outputs results of its computation to $dOut_i$. The channels have the same alphabet, namely, the set of all possible derivatives of regular expressions that can be constructed from Σ. Each $DERIVE_i$ process repeatedly accepts some arbitrary regular expression, and then emits the associated derivative with respect to i.

The parent *DERIVE* is therefore the parallel composition of all $DERIVE_i$ processes. Thus:

$$DERIVE = \|_{i:\Sigma} DERIVE_i \quad \text{where}$$

$$DERIVE_i = dIn_i?q \rightarrow dOut_i!(q, \frac{d}{di}q) \rightarrow DERIVE_i$$

Recall that data is put onto the dIn_i channel by process $EXTRACT_i$. Thus, in principle, the sub-processes $DERIVE_i$ and $EXTRACT_i$ can synchronise independently on events on channel dIn_i and run ahead of a pair of their peer sub-processes, say $DERIVE_j$ and $EXTRACT_j$. Unfortunately, the parent process of the $EXTRACT_i$ processes, namely *EXTRACT*, can only complete once *all* its constituent sub-processes have completed. And a fresh regular expression, q, can only be offered to $DERIVE_i$ via $EXTRACT_i$ once *EXTRACT* has completed, since only then can the recursive call to *OUTER* take place. This deficiency will be corrected in subsection 5.1, where a buffer will be placed on each channel.

At this stage, operations for updating δ and feeding the derivatives back to T are discussed.

4.3 The *UPDATE* Process

The *UPDATE* process is designed to receive a regular expression and its derivative with respect to i as a pair (q, d) from each $dOut_i$ channel. This is to happen independently of the state of readiness to receive some other regular expression and derivative pair on an alternative channel, say $dOut_j$. In each case, the pair is passed on for updating δ and the derivative is considered for possible updating of T. The process is formed by the parallel composition of $DERIVE_i$ processes for each i in Σ. This can be expressed as follows:

$$UPDATE = \|_{i:\Sigma}\ UPDATE_i$$

After receiving the regular expression and derivative pair on the $dOut_i$ channel, each $UPDATE_i$ process behaves as the parallel composition of two sub-processes. One, called UPT_i, corresponds to the action of conditionally adding the derivative to T. The other, called UPD_i, corresponds to the action of unconditionally updating δ.

$$UPDATE_i = (dOut_i?(q, d) \rightarrow (\, UPT_i(d) \parallel UPD_i(q, d))\ );\, UPDATE_i$$

$UPT_i(d)$ establishes whether or not d is in $D \cup T$. If it is, then UPT_i simply terminates successfully. Otherwise, it outputs d on the *insert* channel, thus feeding d back to *OUTER* where d is added to T. After this, the sub-process terminates successfully.

$$UPT_i(d) = insert!d \rightarrow SKIP \nleftarrow d \notin (D \cup T) \nrightarrow SKIP$$

UPD_i unconditionally updates δ and then terminates. The relation is updated by adding an entry into δ that represents a transition from q to d as a result of symbol i. Because each such update will always be with respect to a different (q, i) pair, there is no need to protect the data structure used to represent δ from write conflicts. How such concurrent access to the relevant data structure can actually be achieved is left as an implementation issue.

$$UPD_i(d) = \delta(q, i) := d$$

Note that $UPDATE_i$ starts again after the two sub-processes terminate successfully. Only then will a given $UPDATE_i$ sub-process be ready to input another (q, d) from its respective channel. Once more, there is scope modelling each of the $dOut_i$ channels as a buffer. This would ensure that any holdup in the execution of sub-processes UPT_i and UPD_i (in particular, the computation of the Boolean result of the condition in UPT_i) will not delay the supplier of data on the $dOut_i$ channel. However, in the interests of simplicity, this will not be modelled here. Instead, we illustrate below how the idea of buffering can be included between the $DERIVE_i$ and $EXTRACT_i$ processes, as previously suggested.

5 The *BRZBUFF* process

The $DERIVE_i$ and $EXTRACT_i$ processes are connected by synchronous channels. A process outputting a message onto a channel can only proceed when the receiving process inputs the message. This implies that *EXTRACT* will only terminate once q has been read by all the $DERIVE_i$ processes. This will in turn prevent *OUTER* from producing another q. So if, for example, there is a very slow $DERIVE_i$ process, all

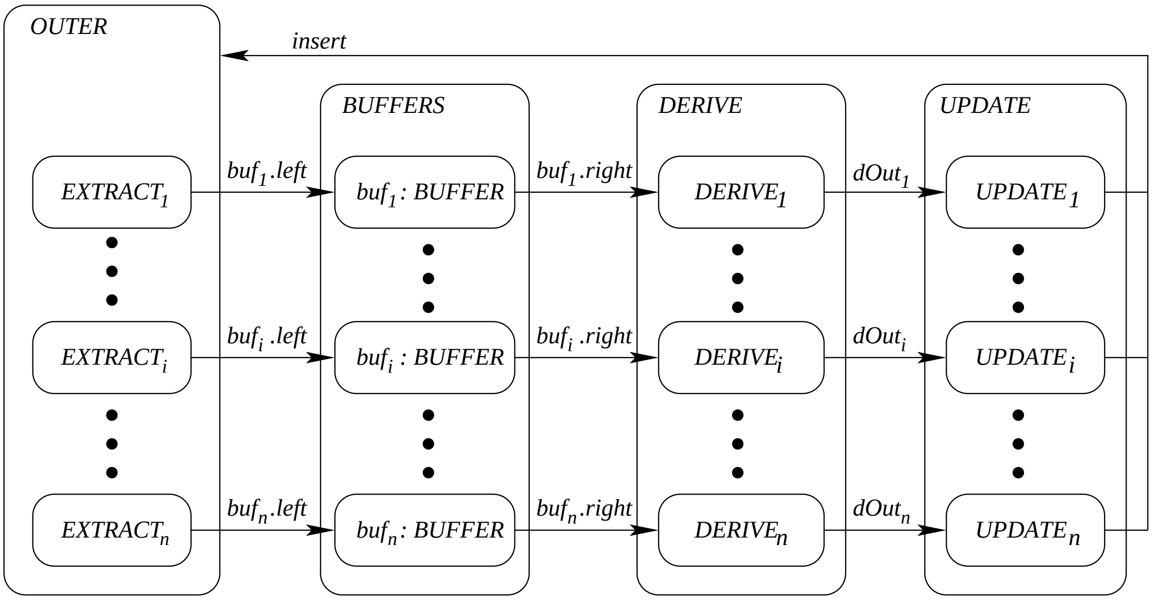


Figure 2. Graphical representation of the *BRZBUFF* process

the others will have to wait for it to complete before continuing. This is clearly not desirable. Work for any processes should ideally be produced at least as fast as it can consume the work.

For the above reason it was decided to connect the $DERIVE_i$ and $EXTRACT_i$ processes through buffers. New q's can then be placed into the buffers without having to wait for all the $DERIVE_i$ processes to complete.

5.1 The *BUFFERS* process

As suggested in [4], a buffer may be modelled using a process called *BUFFER*. It behaves like a queue—messages enter at the right and exit on the left in the same order that they arrived.

$$\begin{aligned} BUFFER &= P_{\langle\rangle} \quad \text{with} \\ P_{\langle\rangle} &= left?x \rightarrow P_{\langle x\rangle} \\ P_{\langle x\rangle\frown s} &= (left?y \rightarrow P_{\langle x\rangle\frown s\frown\langle y\rangle} \\ & \qquad right!x \rightarrow P_s) \end{aligned}$$

Since each pair of processes, $EXTRACT_i$ and $DERIVE_i$, need to be connected through a buffer, multiple labelled copies of the *BUFFER* process are required. As a matter of convenience we will define a process called *BUFFERS* as the parallel composition of these *BUFFER* processes:

$$BUFFERS = \|_{i:\Sigma}(buf_i : BUFFER)$$

The only remaining step is to modify the respective definitions of the $DERIVE_i$ and $EXTRACT_i$ processes so that they interact through these buffers. This is necessary since the alphabet of each of these processes should contain the alphabet of the corresponding buffer process. Thus, each $EXTRACT_i(q)$ sub-process enters data on the left channel of its associated buffer, and may thus be defined as

$$EXTRACT_i(q) = (\underline{buf_i.left!q} \rightarrow SKIP)$$

Each corresponding $DERIVE_i$ sub-process inputs data from the right channel of its associated buffer. Its definition therefore changes to

$$DERIVE_i = \underline{buf_i.right?q} \rightarrow dOut_i!(q, \frac{d}{di}q) \rightarrow DERIVE_i$$

5.2 Putting everything together

A complete system that can be built from the preceding processes is designated $BRZBUFF(D,T)$, because it provides for the buffering just discussed. It is defined as follows:

$$BRZBUFF(D,T) = OUTER(D,T) \parallel BUFFERS \parallel DERIVE \parallel UPDATE$$

Thus, $BRZBUFF(\varnothing, \{E\})$ is a more efficient alternative to $BRZ(\varnothing, \{E\})$. It, too, is therefore a concurrent specification of the sequential algorithm given in figure 1.

Figure 2 depicts the major constituent processes of *BRZBUFF*. It should be clear from the diagram that each $EXTRACT_i$,$DERIVE_i$ pair is now connected via a buffer. The arrows in the diagram indicate the direction of information flow on the channels that connect the processes.

6 Conclusion

Although CSP has proved to be a convenient paradigm and notation for unravelling and articulating the concurrency inherent in the sequential algorithm, it has proven to be deficient in one respect: there does not seem to be a convenient mechanism for gracefully terminating the concurrent specification. As given above, the algorithm terminates when T is empty and further synchronisation is expected on the *input* channel. This means that the *OUTER* process does not terminate in a *SKIP*, but instead awaits further input on this channel, which never appears. Notwithstanding this deficiency, the problem can be easily overcome at the implementation level.

While we have carefully argued the reasons for constructing concurrent version as proposed above, there are of course, a large range of alternative design choices that could be made. These relate not only to overall architectural issues, but also to the level of more or less granularity in the concurrency, and whether the number of processors available should be explicitly taken into account.

Thus, we have already pointed out the scope for including even more buffering than we have. There might also be scope for a more fine-grained approach to updating sets or checking for similarity of regular expressions. At this stage, we have chosen to abstract away from these considerations, and leave their exploration to future research.

From an implementation point of view, it would be relatively easy to use a threaded language such as Java to do the implementation on a single processor platform. However, this does not appear to be particularly interesting, since the context switching required would undoubtedly render the concurrent version less efficient than its sequential counterpart. Instead, we are interested in implementing the concurrent version proposed above on one or more multiprocessor platforms. We expect this to be the immediate focus of our future research.

References

1. T. Agerwala and M. Gupta: *Systems research challenges: A scale-out perspective.* IBM Journal of Research and Development, 50(2/3) March/May 2006, pp. 173–180.
2. J. A. Brzozowski: *Derivatives of regular expressions.* Journal of the ACM, 11(4) 1964, pp. 481–494.
3. C. A. R. Hoare: *Communicating sequential processes.* Communications of the ACM, 26(1) 1983, pp. 100–106.
4. C. A. R. Hoare: *Communicating sequential processes (electronic version)*, 2004, `http://www.usingcsp.com/cspbook.pdf`.
5. R. McDougall: *Extreme software scaling.* ACM Queue, 3(7) September 2005, pp. 36–46.
6. K. Olukoton and L. Hammond: *The future of microprocessors.* ACM Queue, 3(7) September 2005, pp. 26–29.
7. H. Sutter: *A fundamental turn toward concurrency in software.* Dr. Dobb's Journal, 30(3) March 2005, pp. 16–20,22.
8. H. Sutter and J. Larus: *Software and the concurrency revolution.* ACM Queue, 3(7) September 2005, pp. 54–62.
9. D. Ziadi and J.-M. Champarnaud: *An optimal parallel algorithm to convert a regular expression into its Glushkov automaton.* Theoretical Computer Science, 215(1-2) February 1999, pp. 69–87.

Efficient Automata Constructions and Approximate Automata

Bruce W. Watson[1,2,3,4], Derrick G. Kourie[1,3], Ernest Ketcha Ngassam[1,5], Tinus Strauss[1,3], and Loek Cleophas[1,6]

[1] FASTAR Research group
bruce@bruce-watson.com
[2] FST Labs Inc., Kelowna, Canada bruce@fst-labs.com
[3] Department of Computer Science, University of Pretoria, Pretoria, South Africa
{tstrauss, dkourie, bwatson}@cs.up.ac.za
[4] Sagantec Inc., Fremont, California, USA
bruce@sagantec.com
[5] School of Computing, University of South Africa Pretoria, South Africa
ngassek@cs.unisa.ac.za
[6] Faculty of Computing Science and Mathematics Eindhoven University of Technology, Eindhoven, Netherlands
loek@loekcleophas.com

Abstract. In this paper, we present data structures and algorithms for efficiently constructing *approximate automata*. An approximate automaton for a regular language L is one which accepts *at least* L. Such automata can be used in a variety of practical applications, including network security pattern matching, in which false-matches are only a performance nuisance. The construction algorithm is particularly efficient, and is tunable to yield more or less *exact* automata.

Keywords: Automaton construction, approximate automata, memory efficiency, hash functions

1 Introduction

In this paper, we present data structures and algorithms for efficiently constructing an 'approximate automaton' from a regular expression. Very large automata are finding application in areas as diverse as computational linguistic, network intrusion detection, text indexing, and silicon chip design. These problem domains intrinsically have very large amounts of data—both in the form of input strings being processed by finite automata, and also in the size of the automata themselves.

A great deal of effort has been invested in tuning algorithms for processing a string for acceptance by an automaton, or for pattern matching using the automaton[7]. Recently, much less research and implementation effort has been devoted to the efficiency during construction of very large automata[8]. In some of the newer application domains, the 'exactness' of the automata is proving to be less of an issue than

[7] Cf. the proceedings of conferences such as *Prague Stringology Workshop*, *Conference on Implementations and Applications of Automata* and *Combinatorial Pattern Matching*.

[8] In the last decades, asymptotic improvements were made by Champarnaud, Ponty, Ziadi, Chang, Paige, Antimirov, and Watson, among others.

the performance of the algorithms constructing and using the automata. In particular, in network intrusion detection, a pattern matching finite automaton may accept some 'extra' words without ill effects—the algorithm merely detects an additional matched pattern (corresponding to a network security attack), which is subsequently discarded during further vetting of the pattern[9]. In such an application, we can use an approximate automaton—one which accepts the intended language and perhaps some additional strings. Approximate automata have also been derived for a field of pattern matching, where they are known as *factor oracles* [1,3].

Section 2 gives a definition of the problem, along with discussion of some of the existing solutions. Section 3 gives the new algorithm, while Section 4 discusses choices of hash functions. Finally, Section 5 gives some discussion points, conclusions and future work. Throughout this paper, we use standard definitions of deterministic finite automata, which are not discussed or defined further in detail.

2 Problem statement

In this section, we give a brief overview of the problem and existing work on solutions. In Algorithm 2.1, we begin with Brzozowski's automata construction algorithm [2,10]. (Note that the **skip** statement does nothing — it is included for completeness so that the **if-fi** statement has two branches.) The algorithm takes as input a regular expression over alphabet Σ and directly produces a finite automaton. (This algorithm is arguably one of the simplest and most elegant algorithms, although it is not always efficiently implemented, giving the incorrect impression that some of the bitvector-based algorithms (such as those of Glushkov, McNaughton-Yamada, Berry-Sethi, among others) are intrinsically more efficient.)

The abstract states in Algorithm 2.1 have 'internal structure', meaning that they are in fact regular expressions. In practice, the regular expressions are mapped on-the-fly to integers to store the transition function δ in a lookup table and final state set F as a bitvector (indexed by state). That gives Algorithm 2.2, in which state set Q is replaced by a set of integers, the start state is state zero, the signature of δ is appropriately changed, and a 'remapping' data structure is used to map the abstract states to integers. In this algorithm, an abstract state is remapped (assigned an integer representation) when it is first encountered/created (as opposed to when its out-transitions are constructed), since the integer representation may be needed as a transition destination immediately. The worst-case running time of this algorithm (indeed, of all deterministic finite automata construction algorithms) is exponential in the size of the regular expression. However, we are more concerned with those parts of the algorithm and data structures which are tunable.

We make the following observations about Brzozowski's construction algorithm[10]:

1. The sets *done* and *todo* both contain abstract states, whereas they could just contain the integer representations of states, while using the inverse of *remap* to recover the abstract state (which is needed in building the out-transitions). We do not discuss this optimization further, as it is already used in most implementations.

[9] Such further vetting is characteristic of intrusion detection, in which network traffic is rapidly scanned for patterns; pattern 'hits' are subsequently examined for further characteristics before classifying them as a real network security attack.

[10] Some of these observations were previously made in [11,12]—though about another construction algorithm.

Algorithm 2.1 (Brzozowski's construction):

func $Brzconstr(E) \rightarrow$
    $Q, \delta, F := \emptyset, \emptyset, \emptyset;$
    $done, todo := \emptyset, \{E\};$
    do $todo \neq \emptyset \rightarrow$
        let p be some state such that $p \in todo$;
        $done, todo := done \cup \{p\}, todo \setminus \{p\};$
        $Q := Q \cup \{p\};$
        { build out-transitions from p on all alphabet symbols }
        for $a : \Sigma \rightarrow$
            { compute the left derivative of p with respect to a }
            $destination := a^{-1}p;$
            if $destination \notin done \cup todo \rightarrow$
                { *destination*'s out-transitions are still to be built }
                $todo := todo \cup \{destination\}$
            $[\!]$ $destination \in done \cup todo \rightarrow$ **skip**
            fi;
            { make a transition from p to *destination* on a }
            $\delta(p, a) := destination$
        rof;
        { if p is nullable, make it a final state }
        if $\varepsilon \in \mathcal{L}(p) \rightarrow$
            { p should be a final state }
            $F := F \cup \{p\}$
        $[\!]$ $\varepsilon \notin \mathcal{L}(p) \rightarrow$ **skip**
        fi
    od;
    { language of automaton $(Q, \Sigma, \delta, E, F)$ = language of regular expression E }
    return $(Q, \Sigma, \delta, E, F)$
cnuf

Algorithm 2.2 (Brzozowski's construction with remapping):

func $Brzconstr'(E) \rightarrow$
    $next, \delta, F, remap := 0, \emptyset, \emptyset, \emptyset;$
    $remap[E], next := next, next + 1;$
    $done, todo := \emptyset, \{E\};$
    do $todo \neq \emptyset \rightarrow$
        let p be some state such that $p \in todo$;
        $done, todo := done \cup \{p\}, todo \setminus \{p\};$
        $\{$ build out-transitions from p on all alphabet symbols $\}$
        for $a : \Sigma \rightarrow$
            $\{$ compute the left derivative of p with respect to a $\}$
            $destination := a^{-1}p;$
            if $destination \notin done \cup todo \rightarrow$
                $\{$ $destination$'s out-transitions are still to be built $\}$
                $todo := todo \cup \{destination\};$
                $\{$ give $destination$ an integer representation now though $\}$
                $remap[destination], next := next, next + 1$
            $[\!]$ $destination \in done \cup todo \rightarrow$ **skip**
            fi;
            $\{$ make a transition from p to $destination$ on a $\}$
            $\delta(remap[p], a) := remap[destination]$
        rof;
        $\{$ if p is nullable, make it a final state $\}$
        if $\varepsilon \in \mathcal{L}(p) \rightarrow$
            $\{$ p should be a final state $\}$
            $F := F \cup \{remap[p]\}$
        $[\!]$ $\varepsilon \notin \mathcal{L}(p) \rightarrow$ **skip**
        fi
    od;
    $\{$ language of automaton $(\{0, \ldots, next - 1\}, \Sigma, \delta, 0, F)$ = language of regular expression E $\}$
    return $(\{0, \ldots, next - 1\}, \Sigma, \delta, 0, F)$
cnuf

2. In pathological examples, the *todo* set (containing abstract states that still need to be constructed) can grow at one time during construction to contain all states which will ever be built (except for the current state p). The only solutions to this problem are domain-specific; that is, the size of *todo* is sometimes bounded by representing *todo* as a queue or as a stack (yielding, respectively, a depth-first or a breadth-first construction of the automaton's transition graph).
3. The performance of the algorithm depends heavily on the quality of the *remap* representation for fast lookups. There are numerous efficient implementations for *remap*, including hash tables, balanced trees, etc. This is not discussed further here.
4. During construction, the remapper (data structure *remap*) grows to include a mapping from *all* abstract states to their respective integer representations. The memory consumed by *remap* is therefore significant; worse-still, it is only freed after the entire automaton is constructed.

The most promising area for improvement is the last point, for which the following solutions exist:

1. [8,6,7,5] give a space-efficient data structure combining representations of regular expression E, all of the derivative regular expressions in *remap* (the set of states) and the automaton itself.
2. A reachability-based algorithm was presented in [11,12]. That algorithm limits *remap* to those states which are reachable from states still in *todo*.

We now turn to the new algorithm, which is combinable with these two pre-existing solutions.

3 New algorithm

One implementation of *remap* uses a hash table with hash function h, which maps regular expressions (the abstract states) to integers. In the event that two regular expressions hash-collide, a mechanism is normally used to check whether the regular expressions are indeed identical—typically using 'hash-buckets', rehashing, etc. (as is found in elementary data structure textbooks such as [4]). Unfortunately, all of those collision-resolution mechanisms involve representing the abstract states themselves. Our new algorithm eliminates this, thereby allowing hash collisions: two abstract states which hash to the same value are simply mapped to the same integer state. Indeed, the hashed values can be directly used as the states, as in Algorithm 3.1 where we reintroduce state set Q; state sets Q, F and *done* can now be implemented as sets of integers. Note that the test $destination \notin todo$ can also be efficiently implemented using h.

4 Hash functions and their implications

The primary difference between a normal automaton construction algorithm (such as Brzozowski's algorithm) and Algorithm 3.1 is that the latter will merge two states whenever there is a hash collision; such merging would not otherwise have occurred in any of the standard construction algorithms. The resulting automaton is an *approximate automaton* as it will accept additional words not in the language of regular expression E. Precisely how often additional words are accepted clearly depends on

Algorithm 3.1 (New hash-based construction):

```
func Brzconstr″(E) →
    Q, δ, F, remap := ∅, ∅, ∅, ∅;
    done, todo := ∅, {E};
    do todo ≠ ∅ →
        let p be some state such that p ∈ todo;
        done, todo := done ∪ {h(p)}, todo \ {p};
        Q := Q ∪ {h(p)};
        { build out-transitions from p on all alphabet symbols }
        for a : Σ →
            { compute the left derivative of p with respect to a }
            destination := a⁻¹p;
            if h(destination) ∉ done ∧ destination ∉ todo →
                { destination's out-transitions are still to be built }
                todo := todo ∪ {destination}
            ⫿ h(destination) ∈ done ∨ destination ∈ todo → skip
            fi;
            { make a transition from h(p) to h(destination) on a }
            δ(h(p), a) := h(destination)
        rof;
        { if p is nullable, make it a final state }
        if ε ∈ ℒ(p) →
            { p should be a final state }
            F := F ∪ {h(p)}
        ⫿ ε ∉ ℒ(p) → skip
        fi
    od;
    { language of automaton (Q, Σ, δ, h(E), F) = language of regular expression E }
    return (Q, Σ, δ, h(E), F)
cnuf
```

how often hash collisions occur, which depends in turn on the hash function h. The approximate automaton can be forced arbitrarily close to an *exact* automaton for E by increasing the number of bits in the range of h and making h appropriately more intricate[11].

While the design of h is crucial, our ongoing experiments give these guiding requirements:

- It should be structurally inductive on regular expressions.
- It should map to unsigned integers.
- The atomic 'letter' (single symbol) regular expressions should map to the character itself, e.g. $h(a) = \mathit{unsigned}(a)$.
- The remaining two atomic regular expressions (empty string and empty set) should map to infrequently or unused characters, such as -1 and -2.
- The hash of Kleene closure (the star operator) should set a high-bit in the hash of the star's operand's hash, e.g. $h(F^*) = h(F)\&(1 << (\mathit{numbits} - 1))$. In this case, the hash function can also be designed for idempotence of Kleene closure, i.e. $h(F^{**}) = h(F^*)$, etc. Such 'design-for-identities' allows us to specifically hash-collide states which look dissimilar, but are equivalent, thereby achieving a measure of *minimization*[12].
- The hash of a union/or regular expression should combine the two sub-hashes via an associative and commutative operator, such as exclusive-or.
- The hash of two concatenated regular expressions should combine the two sub-hashes while simultaneously being anti-symmetrical (as concatenation is), such as bitwise concatenation, or left-shifting the first sub-hash before exclusive-oring with the second sub-hash.

In short, the hash function should reflect the algebraic properties of the regular operators themselves.

4.1 Fixing the automaton size *a priori*

Interestingly, the use of a hash function to generate the state-set enables us to *a priori* choose the number of states in the final automaton. Rather than accumulating the hashed values in variable Q, we initially fix our state set as $\{0, \ldots, n\}$ (for some n). Subsequently, we always use $h(p) \mod n$ instead of $h(p)$, thereby bringing all states into the appropriate range. This can be particularly useful in cases where dynamic memory (re)allocation is costly while building the automaton.

5 Discussion and future work

We have presented an efficient new algorithm for constructing approximate deterministic finite automata. The 'approximateness' of the finite automata (that is, how few 'extra' words they accept over and above their intended language) can be controlled by choice of a hash function, some guidelines for which were presented. Unlike other automata construction algorithms, the number of states can be fixed *a priori*.

There remain a number of interesting research questions and tasks:

[11] When the number of bits is equal to the largest derivative of E, h can be used to directly encode each such derivative regular expression—giving perfect hashing.

[12] Other regular identities which may be specifically hash-collidable include $F \cdot \emptyset = \emptyset$, $F \cdot \varepsilon = F$, $\varepsilon^* = \varepsilon$, etc.

1. The new algorithm and a variety of hash functions should be benchmarked; this work is ongoing.
2. The effects of various hash functions should be tested in practice (for example, in intrusion detection) for how close the resulting automaton is to the desired language.
3. Some operations in the new algorithm can be parallelized. A parallelization of Brzozowki's algorithm is the subject of another paper submitted to PSC. It would be interesting to know whether further parallelization opportunities exist in the new algorithm.
4. In data structure design, the size of the hash tables are typically chosen to be a prime number (and therefore the hash key is reduced modulo this size), as this reduces the probability of collisions. It would be interesting to know whether a similar property exists in the new algorithm, with typical choices of hash functions.

References

1. C. Allauzen, M. Crochemore, and M. Raffinot: *Factor oracle: A new structure for pattern matching*, in Proceedings of SOFSEM, 1999, pp. 295–310.
2. J. A. Brzozowski: *Derivatives of regular expressions.* Journal of the ACM, 11(4) 1964, pp. 481–494.
3. L. Cleophas, B. W. Watson, and G. Zwaan: *Constructing factor oracles*, in Proceedings of the Eighth Prague Stringologic Conference, J. Holub, ed., Prague, Czech Republic, Sept. 2003, Czech Technical University.
4. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein: *Introduction to Algorithms*, The MIT Press, second ed., 2001.
5. M. Frishert: *Fire works & fire station: A finite automata and regular expression playground*, Master's thesis, Faculty of Computing Science, Eindhoven University of Technology, the Netherlands, 2005.
6. M. Frishert, L. Cleophas, and B. W. Watson: *FIRE station: An environment for manipulating finite automata and regular expression views*, in Salomaa [9].
7. M. Frishert and B. W. Watson: *Combining regular expressions with (near-)optimal Brzozowski automata*, in Salomaa [9].
8. M. Frishert, B. W. Watson, and L. Cleophas: *The effects of rewriting regular expressions on their accepting automata*, in Proceedings of the Eighth Conference on Implementations and Applications of Automata, O. Ibarra and D. Zhu, eds., Santa Barbara, USA, July 2003, Springer-Verlag, pp. 304–305.
9. K. Salomaa, ed., *Proceedings of the Ninth Conference on Implementations and Applications of Automata*, Kingston, Canada, July 2004, Springer-Verlag.
10. B. W. Watson: *Taxonomies and Toolkits of Regular Language Algorithms*, PhD thesis, Faculty of Computing Science, Eindhoven University of Technology, the Netherlands, Sept. 1995.
11. B. W. Watson: *An early-retirement plan for the states*, in Proceedings of the Third Prague Stringologic Workshop, J. Holub, ed., Prague, Czech Republic, Sept. 1998, Czech Technical University, pp. 119–124.
12. B. W. Watson: *Reducing memory requirements during finite automata construction.* Software — Practice & Experience, 34(3) 2004, pp. 239–248.

On Implementation and Performance of Table-Driven DFA-Based String Processors

Ernest Ketcha Ngassam[1], Derrick G. Kourie[2,3], and Bruce W. Watson[2,3]

[1] School of Computing, University of South Africa, Pretoria 0003, South Africa,
ngassek@unisa.ac.za
[2] Department of Computer Science, University of Pretoria, South Africa,
{dkourie,watson}@cs.up.ac.za
[3] FASTAR Research Group (www.fastar.org)
{eketcha,derrick,bruce}@fastar.org

Abstract. Table-driven (TD) DFA-based string processing algorithms are examined from a number of vantage points. Firstly, various strategies for implementing such algorithms in a cache-efficient manner are identified. The denotational semantics of such algorithms is encapsulated in a function whose various arguments are associated with each implementation strategy. This formal view of the implementation strategies suggests twelve different algorithms, each blending together the implementation strategies in a particular way. The performance of these algorithms is examined in against a set of artificially generated data. Results indicate a number of cases where the new algorithms outperform the traditional TD algorithm.

Keywords: deterministic finite automata, table-driven algorithms, performance, recognizer denotational semantic, string recognizer, string processor

1 Introduction

Most automata implementers rely on a well-known table-driven (TD) algorithm for string acceptance testing. The algorithm is a simple loop which accesses the transition table and scans through string's symbols in order to establish whether it is part of the language modelled by the DFA or not. To the best of our knowledge, not much has been done to explore alternative methods for implementing DFA-based recognizers that could outperform the conventional TD algorithm in specialised circumstances. Of course, the hardcoded approach suggested by Thompson in [8] has been studied, and experiments revealed that it outperforms TD for DFAs of relatively small size [2].

For each successive element of the input string being scanned, the TD algorithm has to access a row of the transition table. The overall performance of a TD recognizer is determined by the pattern of accesses into the transition table induced by the input string. If, for example, the string induces a fairly random pattern of accesses into the table, then there will be a relatively high probability of cache misses and a consequent degradation of performance [3]. It is thus of interest to investigate alternative table-driven approaches that somehow organize the transition table so as to minimize such effects. It should be noted at the outset that the conclusions to be expected from such an investigation must, of course, be tentative and probabilistically conditioned, since the characteristics of input strings are, in practice, at best only probabilistically known. Nevertheless, as the amount of hardware cache increases, and as finite automata technology is deployed in ever-larger applications, the need to explore effective cache utilization strategies is an important research theme.

In this paper, we propose three implementation strategies associated with the table-driven algorithm in order to minimize the overall latency of a recognizer. In each case, the revised algorithm outperforms the TD algorithm for an appropriate class of input strings. The first strategy, referred to as the dynamic state allocation (DSA) strategy, has already been suggested in [3] and was proven to outperform TD when a large-scale FA is used to recognise very long strings that tend to repeatedly visit the same set of states. The second strategy, referred to as the State pre-ordering (SpO) strategy, relies on a degree of prior knowledge about the order in which states are likely to be visited at runtime. It is shown that the associated algorithm outperforms its TD counterpart no matter the kind of string being processed. The last strategy, referred to as the Allocated Virtual Caching (AVC) strategy, reorders the transition table at run time and also leads to better performance when processing strings that visit a limited number of states.

The remaining part of the paper is organized as follows: in section 2 below, we provide a unified formalism to describe table-driven recognizers that are based on the above mentioned strategies. Section 3 discusses each of the algorithms. Then follows in section 4 discussion on experimental results. The conclusion and further direction to this contribution are given in section 5.

2 Characterization of Table-driven Recognizers

In this section, we describe the various table-driven string recognizers mentioned in the introduction in a formal fashion—specifically, in terms of mathematical functions—where arguments correspond to the strategy according to which the TD recognizer is implemented. The next section gives the pseudo-code for these algorithms.

Consider an automaton $\mathrm{M} = (\mathcal{Q}, \mathcal{V}, \delta, s_0, \mathcal{F})$, where: $\mathcal{Q}$ is the set of states; $\mathcal{V}$ is the set of the alphabet symbols; δ is the transition function; s_0 is the start state; and $\mathcal{F}$ is the set of final states of the automaton. We denote by $\mathcal{T} = \mathcal{P}(\mathcal{Q} \times \mathcal{V} \times \mathcal{Q})$ the power set of $\mathcal{Q} \times \mathcal{V} \times \mathcal{Q}$. Since $\mathcal{V}^*$ is the set of all strings over the alphabet, including the empty string; the language of M is denoted $\mathcal{L}(\mathrm{M}) \subseteq \mathcal{V}^*$. We also consider the set $\mathbb{B} = \{T, F\}$ of boolean.

Traditionally, FA-based string recognizers are implemented using the table-driven algorithm. In this case, the transition function is represented in the form of a table (two-dimensional array) whose columns represent the symbols of the alphabet and rows the states of the automaton. The table is therefore an implementation of the function $\delta(q, c_j)$ that is associated with the FA at issue[4].

The traditional TD algorithm does not account for the way in which the transition table is accessed. If the input string results in transitions to states that are arbitrarily located in the table, then there are likely to be many cache misses and consequent performance degradation. On the other hand, if transitions are to states that are contiguously stored in the transition table, then the cache utilisation will be optimal, with consequent performance gains. The following subsection discusses the strategies investigated to date aimed at exploiting this insight.

[4] For convenience, and without loss of generality, it will be assumed that states are integers in the ranges $[-1, |\mathcal{Q}|)$. State -1 corresponds to the sink state that indicates rejection, and need not be represented as a row in the table. Each remaining state, q, corresponds to the q^{th} table row. By convention, 0 corresponds to the start state.

2.1 Dynamic State Allocation (DSA)

Implementation of FA-based string processors that rely on the dynamic state allocation principle requires that a dynamically allocated space be created in memory which is used during acceptance testing. At runtime, as each state is encountered that falls for the first time within the *string path*[5], it is allocated a memory block into which the state's transition information (i.e. a row in the original transition table) is copied. Subsequent references to such a state's transitions are then made via this new piece of memory, rather than via the original transition table. Furthermore, the memory blocks allocated to states on the string path are contiguous, and arranged in the order in which the states are encountered. The DSA strategy was first introduced in [3] and further improvements on the algorithm were suggested in [4].

The TD algorithms based on various strategies are to be described as a mathematical function in subsection 2.4. In this function, we will rely on an argument to represent the DSA strategy. The argument, D is a natural number that indicates the extent to which the strategy has been adopted. This can range from not having been adopted at all, in which case the argument should be 0; to having been adopted for every possible state visited along the state path, in which case the argument should be set to $n = |\mathcal{Q}|$. Two scenarios are distinguished:

- In the *unbounded dynamic state allocation* scenario, the relevant strategy variable is equal to the maximum number of states (i.e. $D = n$). In this case, dynamic allocation occurs as new states that have not yet been dynamically allocated in the new memory space are encountered. In a worst case situation it may be necessary to have the size of the newly allocated memory equal to that of the originally used memory. All that has changed is that the state ordering in the newly allocated memory is organized in a contiguous fashion with respect to the sequence of states in the string's state path.
- In a *bounded dynamic state allocation* scenario, a relevant strategy variable is strictly less than the maximum number of states, but also greater than zero ($0 < D < n$). In this case, the algorithm only has a limited number of states to be allocated dynamically in memory. The restriction means that not all states need necessarily be represented in the new memory location when processing a string whose string path requires more states than those allocated.

Note that a bounded DSA strategy requires a *replacement policy*—i.e. a policy about whether and how to replace states in the dynamically allocated space. In this paper, we shall assume the *direct mapping* replacement policy. In terms of this policy, when the dynamic space is full and reference is made to a state that has not yet been visited, the new state is assigned an address in the dynamically allocated space based on the modulus operation used to identify the state to be removed from the dynamic space. Of course, there could be various other replacement policies such as: the least recently used (LRU) policy whereby, state in allocated memory is removed, replacing it with the least recently invoked state; or associative mapping and the set associative mapping [1,7]. Alternatively, we may simply chose not to do any replacement at all within the dynamically allocated space. However, these various policy options will not be further explored here.

In the next subsection, a new TD-based implementation strategy referred to as the state pre-ordering strategy is discussed.

[5] String path is construed to mean the set of visited states that are encountered during the processing of the input string.

2.2 State Pre-ordering (SpO)

It may sometimes happen that the percentage of visited states is far below the overall number of states that make up the automaton. Moreover, those frequently visited states may be scattered throughout the transition table. To optimize performance in such a situation, a mechanism is needed to reorganize the transition graph such that frequently accessed states are grouped together in memory, thus reducing the probability of cache misses. The SpO strategy addresses this issue. The strategy requires that a function be run before acceptance testing. The function reorders the automaton's original placement of states in the transition table, to correspond with some ordering of states that is provided by the user as input. Such input could, for example, be based on some *a priori* reasoning of the user, or on an empirical analysis of the history of state visits in prior runs of the FA.

As in the case of the DSA strategy, the reference to this strategy in the mathematical function of subsection 2.4 is in terms of a boolean argument, say P. It indicates whether the SpO strategy is used for implementing the FA or not. Therefore, when the variable evaluates to $true$, a preprocessing operation that reorders the automaton states is to be invoked before acceptance testing. As a result, state i transitions are not necessarily in the i^{th} row of the table. Thus, subsequent accesses to the transition table is done via an auxiliary array, say $p : [0..n)$, whose i^{th} entry is the *new* row number of state i in the transition table. However, if the variable evaluates to $false$, the strategy is not used at all.

The next subsection discusses yet another implementation strategy referred to as allocated virtual caching.

2.3 Allocated Virtual Caching (AVC)

The allocated virtual caching strategy treats a portion of the memory that holds the transition table as a kind of cache area. Typically, this portion of memory is the first V rows of the transition table, where V is some pre-specified value. The phrase *virtual cache* has been coined to differentiate this block of RAM memory from the conventional cache memory in hardware. We algorithmically enforce a type of cache behaviour in utilising this memory. The dedicated portion of the memory is referred to as the *allocated virtual cache*.

During acceptance testing, individual states are transferred into the cache as they are visited, in the hope of enhancing the cache's spatial and temporal locality of reference. The virtual cache will typically be limited in size, and may therefore not contain every single state required. As a result, when reference is made to a state that is not present in the cache, a replacement policy must be used to remove a state from the cache. Removing a state from the cache makes a *cache line* available, so that a new state's information can be placed in the empty cache line.

It is important to realise that the initial cache is regarded as empty, even though the actual cache is occupied by state transition information for the first V table entries. By this we mean that a pointer which keeps track of the portion of cache utilised will initially indicate 0 and progressively climb to V as more and more rows are swopped into cache. Eventually, this pointer will climb to V, whereafter the cache is regarded as full.

The AVC strategy differs from the DSA strategy in the following sense. In the DSA strategy, the state transition information is copied (dynamically) into an free portion of the memory. The AVC strategy, on the other hand, utilises the original transition table. It swops rows of the initial table, removing some state transition information from the cache into elsewhere in the transition table, and bringing in fresh state transition information into the cache in respective of the most recently encountered state (unless, of course, that most recently encountered state is already in cache, in which case no swopping is required).

Unlike the DSA strategy, the AVC strategy only makes sense if it is bounded. If it were unbounded, then the entire transition table would be regarded as the virtual cache—if a prefix of the string path references all states, then there would never be the need to change the contents of the virtual cache when inspecting the suffix of the string path.

For the present work, for deciding on a state to be removed from cache,we rely on the direct mapping policy that was mentioned in the discussion of the DSA strategy. Of course, we could also chose to have a policy of not swapping out states from the cache once it is full, but instead referencing the transition information from the original state position in the table.

In the mathematical description of the TD algorithm based on an AVC strategy, given in subsection 2.4, V (a natural number) is used as a function argument. If V is 0 then the AVC is not used at all. Alternatively if V has any value in the range $[0, n)$ this means that up to V states may be part of the cache.

The next subsection gives a mathematical function to summarise the above strategies to be built into TD recognizers, relating each implementation strategy to an argument of the function.

2.4 A Unified Formalism of TD algorithms

In general, an acceptor or a string recognizer of a finite automaton is an algorithm that relies on the finite automaton's transition function in order to determine whether a string is part of the language modelled by the FA or not. Therefore, given a input string s and a transition function, δ, the recognizer scans each symbol of the string and returns a boolean. Clearly, this way of describing a recognizer reflects the semantics of the core TD implementation. However, it places no restriction on how the the mapping of s and δ should operate. Since we have introduced three additional strategy arguments for describing the various ways in which a TD recognizer could be implemented, we may now consider a recognizer as a function that requires as arguments a transition function δ, the input string s, and the respective DSA, SpO and AVC strategy arguments, D, P and V. The function then returns a boolean as result. Let us call such a function ρ. It may be characterized as follows:

$$\rho : \mathcal{T} \times \mathbb{N} \times \mathbb{B} \times \mathbb{N} \times \mathcal{V}^* \nrightarrow \mathbb{B}$$

$$\rho(\Delta, D, P, V, s) = \begin{cases} \mathit{true} \text{ if } s \in \mathcal{L}(\mathrm{M}) \\ \mathit{false} \text{ if } s \notin \mathcal{L}(\mathrm{M}) \end{cases}$$

where $0 \leq D \leq |\mathcal{Q}|$, $P \in \mathbb{B}$, and $0 \leq V < |\mathcal{Q}|$.

In fact, ρ may be regarded as the *denotational semantics* of the string recognizer [6]. It specifies the "meaning" of the algorithm in functional terms, but hides details about how the algorithm that performs acceptance testing should actually work. There are, in fact, various ways in which the processing can take place, each corresponding to different instantiations of the strategy arguments. The next section gives algorithms that correspond to various instantiations.

3 The various TD algorithms

The strategy arguments of ρ may be instantiated in $3 \times 2 \times 2 = 12$ different ways: D can either be 0 (no DSA), a bounded value ($D \leq |Q|$), or an unbounded value ($D = |Q|$); P is one of two Boolean values; and V is either 0 (no AVC) or greater than 0. Each of these instantiations may be associated with a different implementation of a table-driven FA-based string recognizer.

Each row of table 1 depicts one of the 12 different algorithms that can be constructed, based on the combination of the values that are assigned to strategy arguments. For easy reference, the last column in the table informs the reader the subsection in the text where the algorithm is discussed. Note that for reasons of space economy, not all algorithms can be fully discussed here. In these cases, the column references a subsection where the algorithm is mentioned. The first column in the table are triplets of the form (D, P, V), indicating instances of the strategy variables that relate to this specific algorithm. The second column informs the reader of the strategies that are implemented in the construction of the algorithm.

The name given to each algorithm starts with the letter t followed by the concatenation of the numbers assigned to each active strategy as follows: the number 1 is assigned to the DSA strategy; the number 2 is assigned to the SpO strategy; and the number 3 is assigned to the AVC strategy. Since there are two variations to the DSA strategy, we chose to prefix its number with: u for the unbounded case, and b for the bounded case. For example, t_{u1} refers to the table-driven algorithm based on the unbounded DSA strategy, t_{b123} refers to the table-driven algorithm based on the bounded DSA strategy combined with the SpO and AVC strategies; and of course, t refers to the core table-driven algorithm.

Combination	Active strategy	Name	Reference
$(0, F, 0)$	None	t	3.1
$(d, F, 0)$	bounded DSA	t_{b1}	3.2
$(n, F, 0)$	unbounded DSA	t_{u1}	3.2*
$(0, T, 0)$	SpO	t_2	3.3
$(0, F, v)$	AVC	t_3	3.4
$(0, T, v)$	SpO and AVC	t_{23}	3.5
$(d, T, 0)$	bounded DSA and SpO	t_{b12}	3.5*
(d, T, v)	bounded DSA, SpO and AVC	t_{b123}	3.7
(d, F, v)	bounded DSA and AVC	t_{b13}	3.6
$(n, T, 0)$	unbounded DSA and SpO	t_{u12}	3.5*
(n, T, v)	unbounded DSA, SpO and AVC	t_{u123}	3.7*
(n, F, v)	unbounded DSA and AVC	t_{u13}	3.6*

Table 1. The Range of TD-based algorithms (those with a * are not discussed in details)

These various algorithms are now discussed in the subsections below.

3.1 The core TD algorithm

This well-known algorithm is rather simple: it takes as input the transition function δ, the string s to be processed, and returns a boolean.

Algorithm 3.1(The core table-driven recognizer)

proc $t(\delta, s)$
  $; q, j := 0, 0$
  do $(j < s.len) \wedge (q \geq 0) \rightarrow$
    $q,\ j := \delta(q, s_j),\ j + 1$
  od
  if $q < 0 \rightarrow$ {return false} $[\!]$ $q \geq 0 \rightarrow$ {return true} **fi**

3.2 The bounded TD-DSA algorithm

The algorithm is based on the DSA strategy and accounts for both bounded and unbounded case according to the nature of the strategy argument D. The unbounded case of the algorithm was discussed in [3] and here, we provide its bounded counterpart, which involves state replacement when the dynamically allocated space is full, and reference is made to a state that has not previously been visited.

In addition to the core TD algorithm inputs, TD-DSA also requires as input the number of states, D, that may be dynamically allocated; the block size, Z, of memory required for each newly allocated state; and the starting address, A from which memory is to be allocated. In the algorithm, The variable p serves as a counter of the number of states that have been dynamically allocated to date. If $p \geq 0$ is the q^{th} entry in array $m_{[0,n)}$ (which is initialized to -1) then this means that the q^{th} row of δ has been copied over to the p^{th} row of the table d that has been dynamically evolved in memory. The function $search(m, r)$ returns the index i in the array $m_{[0..n)}$ for which $m_i = r$.

When $m_q = -1$, where q is the current state, then a further test is made on p to find out whether the reserved dynamic portion of memory is full or not. If not full, the block referenced by variable d is expanded by Z bytes (starting from address B which initially is A), the state information is copied into d, and B is incremented to points to the next address where the next state information may be copied if the dynamic memory block is not full.

Algorithm 3.2(The bounded TD-DSA recognizer)

proc $t_{b1}(\delta, D, A, Z, s)$
  $; m_{[0..n)}, B, q, j, p := -1, A, 0, 0, 0$
  ;**do** $(j < s.len \wedge q \geq 0) \rightarrow$
    if $(m_q = -1) \rightarrow$ { state not dynamically allocated}
      if $(p < D) \rightarrow$
        $; m_q, d_p := p, malloc(B, Z)$
        $; d_{p,[0..a)}, p, B := \delta_{q,[0...a)}, p + 1, B + Z$
        $; q := d_{p,s_j}$
      $[\!]$ $(p \geq D) \rightarrow$
        $r := MOD(q, D)$ { remainder in the division of q by D}
        $; m_q := r$
        $; i := search(m, r)$
        $; m_i := -1$
        $; d_{r,[0..a)}, q := \delta_{q,[0..a)}, d_{r,s_j}$
      fi

$[\!]\ \ m_q \neq -1 \rightarrow$ **skip**{ state dynamically allocated}
fi
$; q, j := d_{m_q, s_j}, j + 1$
od
if $q < 0 \rightarrow$ {return false} $[\!]$ $q \geq 0 \rightarrow$ {return true} **fi**

3.3 The TD-SpO algorithm

Again, in addition to the input for the TD algorithm, the TD-SpO algorithm is provided with an auxiliary array $p_{[0..n)}$ such that p_i specifies to which new row of δ the i^{th} row of δ should be moved. At the start, the function $reorder(\delta, p)$ reorders the automaton's states according to p's entries. Then follows proper acceptance testing whereby, access to a state q information is made indirectly via p. Thus, $\delta(p_q, s_j)$ returns the transition triggered by the string's symbol s_j at state q.

Algorithm 3.3(The TD-SpO recognizer)

proc $t_2(\delta, p, s)$
$; reorder(\delta, p)$
$; q, j := 0, 0$
do $(j < s.len) \wedge (q \geq 0) \rightarrow$
$q, j := \delta(p_q, s_j), j + 1$
od
if $q < 0 \rightarrow$ {return false} $[\!]$ $q \geq 0 \rightarrow$ {return true} **fi**

3.4 The TD-AVC algorithm

For this algorithm, the size of the virtual cache, V, has to be provided. Auxiliary arrays $m_{[0..n)}$, $c_{[0..V)}$ and $i_{[0..n)}$ are used. $m_q = r$ means that state q transition information is held in row r of the table δ. The array $c_{[0..V)}$ is used to hold the states currently in the cache, and the array $i_{[0..n)}$ indicates whether a state is currently in the cache ($i_q = 0$) or not ($i_q = -1$). All entries of i are set to -1, indicating that states have not yet been visited although the first V states are initially in the cache by default. The variable l is used as cache line controller and helps establish whether the cache is full or not. The cache is said to be full when the number of different states visited thus far has reached V.

For every iteration of the main loop, a test is made to check whether the current state, q is in the cache or not. If it is ($i_q \neq -1$), then the normal transition code is executed. Otherwise a check is made to see whether the cache is full or not. If the cache is full ($l \geq V$) and q is not in the cache, then further acceptance testing may take place only *after* doing state replacement, as for the DSA strategy. Otherwise, (i.e. the cache is not full and q is not in cache) q's transition information is swapped with that of state l, then l is incremented and acceptance testing takes place. Of course, information of those states that are in the cache by default are not swopped provided that their state matches the state in the cache line (ie. $q = c_l$).

State information is swopped by the function $swd(\delta[m_q], \delta[m_p])$ that also interchanges the entry m_q of the current state q with the entry m_p of the state p currently in the cache line, and referenced in the algorithm by c_l.

In order to do state replacement when the cache is full, we use the modulo operation to determine the position in the cache to which q should be moved. Then follows implicit interchange of state information as previously described. Of course, whether the cache is full or not, a states's information in i is updated accordingly as depicted in the algorithm.

Algorithm 3.4(Table-driven based on allocated virtual caching)

proc $t_3(\delta, V, s)$
    $; q, j, p, l := 0, 0, 0, 0$
    $; m_{[0..n)}, c_{[0..V)}, i_{[0..n)} := [0..n), [0..V), -1$
    do $(j < s.len) \wedge (q \geq 0) \rightarrow$
        if $(i_q \neq -1) \rightarrow$ **skip**
        $[\!]$ $(i_q = -1) \wedge (l < V) \rightarrow$
            if $q = c_l \rightarrow$ **skip**
            $[\!]$ $q \neq c_l \rightarrow$
                $p := c_l$
                $; swd(\delta[m_q], \delta[m_p])$
                $; i_p, c_l := -1, q$
            fi
            $; i_q, l := 0, l + 1$
        $[\!]$ $(i_q = -1) \wedge (l \geq V) \rightarrow$
            $p := MOD(m_q, V)$
            $swd(\delta[m_q], \delta[m_{c_p}])$
            $i_q, i_{c_p}, c_p := 0, -1, q$
        fi
        $; q, j := \delta(m_q, s_j), j + 1$
    od
    if $q < 0 \rightarrow$ {return false} $[\!]$ $q \geq 0 \rightarrow$ {return true} **fi**

3.5 The TD-SpO-AVC algorithm

The algorithm relies on both SpO and AVC strategies. A naïve approach for its implementation would be to first reorder the automaton's states using the function $reorder(\delta, p)$, and then invoke (for every iteration of the main loop) a function $tdavc(\delta, p, m, c, i, l, V, j, q, s)$ that updates the next state q to be transited to, as well as the next index j of the string s currently being processed. This latter function also takes as parameters the arrays m, c, and i as well as the cache line controller, l, previously described in subsection 3.4. Moreover, access to states' original information is made via entries of the array p. The algorithm below depicts the pseudo-code of algorithm t_{23}.

Algorithm 3.5(The TD-SpO-AVC algorithm)

proc $t_{23}(\delta, p, V, s)$
    $; reorder(\delta, p)$
    $; q, j, p, l := 0, 0, 0, 0$
    $; m_{[0..n)}, c_{[0..V)}, i_{[0..n)} := [0..n), [0..V), -1$
    do $(q < s.len) \wedge (q \geq 0) \rightarrow$

$$
\begin{array}{l}
\qquad tdavc(\delta, p, m, c, i, l, V, j, q, s) \\
\quad \textbf{od} \\
\quad \textbf{if } \ q < 0 \rightarrow \{\text{return false}\} \ [\!] \ q \geq 0 \rightarrow \{\text{return true}\} \ \textbf{fi}
\end{array}
$$

The same principle applies for algorithms t_{u13} and t_{b13}, that also rely on the combination of the SpO strategy with the unbounded and bounded DSA strategies. All that is required is to change the function in the main loop with a function that implements the relevant strategy and updates the next state and the next string index.

3.6 The bounded and unbounded TD-DSA-AVC algorithm

These two algorithms combine the AVC strategy with either the bounded DSA strategy or the unbounded DSA strategy. We only discuss here the unbounded TD-DSA-AVC algorithm. For its implementation, the following simple policy may be adopted for an automaton of n states:

- The first k states of the automaton are cacheables. That is, they are processed within the virtual cache which holds up to V states, and the following holds: $0 < V < k < n$
- The remaining $n - k$ states are processed within the allocated dynamic memory space where each state occupies Z bytes, and the first state to be allocated dynamically is located at address A in the memory.

In the algorithm, for every iteration of the main loop, upon accessing a state q, a test is first made to determine whether the state is cacheable or not. If that is the case, the function $tdavc(\delta, m, c, i, l, V, j, q, s)$ is invoked that updates the next state q, the next symbol s_j to be processed, as well as all the other parameters involved in the function. However, if the state is not cacheable, it ought to be processed within the dynamically allocated space using the function $utddsa(\delta, A, Z, B, q, j, s)$ that updates the variables q and j, as well as the parameter B that holds the next address to be used for space allocation in the case the state currently being processed has not yet been visited. The pseudo-code of the algorithm is depicted below.

Algorithm 3.6(The unbounded TD-DSA-AVC algorithm)

$$
\begin{array}{l}
\textbf{proc } \ t_{u13}(\delta, k, A, Z, s) \\
\qquad ;q, j, p, l, B := 0, 0, 0, 0, A \\
\qquad ;m[0..n), c[0..V), i[0..n) := [0..n), [0..V), -1 \\
\qquad ;\textbf{do } \ (j < s.len \wedge q \geq 0) \rightarrow \\
\qquad\qquad \textbf{if } \ q < k \rightarrow tdavc(\delta, m, c, i, l, V, j, q, s) \\
\qquad\qquad [\!] \ \ q \geq k \rightarrow utddsa(\delta, A, Z, B, q, j, s) \\
\qquad\qquad \textbf{fi} \\
\qquad \textbf{od} \\
\qquad \textbf{if } \ q < 0 \rightarrow \{\text{return false}\} \ [\!] \ q \geq 0 \rightarrow \{\text{return true}\} \ \textbf{fi}
\end{array}
$$

3.7 The bounded and unbounded TD-DSA-SpO-AVC algorithm

The algorithms consist of the SpO strategy and a combination of the AVC strategy and either of the DSA strategies. For consistency, we briefly discuss the unbounded case. Based on previous discussions, the algorithm would first consists of a preprocessing phase whereby the function $reorder(\delta, p)$ is invoked. Then follows the processing phase similar to the unbounded TD-DSA-AVC algorithm described in the previous subsection. Of course the policy on the number of states to be cacheable and those to be dynamically allocated must be defined. We adopt the same principle as previously described, and the algorithm is depicted below.

Algorithm 3.7(The unbounded TD-DSA-SpO-AVC algorithm)

---

proc $t_{u123}(\delta, p, s, c, k, d, A, Z)$
  $; reorder(\delta, p)$
  $\{ \ Initializations \ \}$
  ;**do** $(q < s.len \wedge q \geq 0) \rightarrow$
    if $q < k \rightarrow tdavc(\delta, p, m, c, i, l, V, j, q, s)$
    $[\!]$ $q \geq k \rightarrow utddsa(\delta, p, A, Z, B, q, j, s)$
    fi
  od
  if $q < 0 \rightarrow$ {return false} $[\!]$ $q \geq 0 \rightarrow$ {return true} **fi**

The reader should notice the presence of the auxiliary array p as argument in the functions in the main loop. This simply emphasizes that, unlike algorithm t_{u13}, access to original states' information is made via entries of p.

4 Experimental Results

Various experiments were conducted in order to determine the point at which the derived implementation strategies outperform the core TD implementation. We relied on artificially generated data, in each case contrived to generate an input string that would demonstrate the strength of a relevant new algorithms in relation to the core TD algorithm.

The algorithms were implemented using the Netwide Assembly language (NASM), under the Linux OS, on an Intel Pentium IV machine. For each algorithm under investigation, 120 different automata were generated of size ranging from 100 to 12000 states, with increment of 100. Associated with each recognizer was an accepting string of length $4n$, where n is the number of states of the automaton. The strings generated were explicitly designed such that the first n symbols where randomly chosen among the automaton states, and those symbols where repeated 4 times.

For each algorithm involving the SpO strategy, the array of state positions $p_{[0..n)}$ was randomly generated.

Each recognizer was then run 50 times, and the minimum time in clock cycles (ccs) was recorded. In order to evaluate the extent to which the algorithms outperformed the core TD algorithm and vice-versa, the data collected for each algorithm was plotted against the data of the core TD algorithm.

For the bounded TD-DSA and the TD-AVC algorithms, we chose a bound of 50% on the number of states—that is, for an automaton of size n, up to $\lceil n/2 \rceil$ states were processed in the allocated dynamic memory or in the virtual cache, respectively.

The graphs in figure 1 depict the performance of the core TD algorithm against the bounded TD-DSA, the unbounded TD-DSA, the TD-SpO, and the TD-AVC algorithm respectively. The performance of the unbounded TD-DSA strategy was already discussed in [3]. As shown in the graphs, the TD-algorithm outperforms the bounded TD-DSA algorithm under the conditions discussed above. This suggests that the bounded nature of the algorithm requires frequent state replacement during acceptance testing when the dynamically allocated space is full. Therefore, the following scenarios merit further experimentation in respect of the bounded TD-DSA algorithm:

- *Replacement policy*: We have chosen to use the direct mapping policy in order to swap a state in and out of the allocated free memory space when no more space is available. Such policy may not always guarantee better cache placement and data organization since it may happen that a state is constantly swapped in and out of the cache, and hence poor performance of the algorithm. A policy such as the associative mapping or the LRU policy could perhaps be used to avoid such problem [7]. Alternatively, once the threshold for state allocation has been reached, we could avoid using any replacement policy. Instead, acceptance could be performed through the transition table whenever reference is made to a state out of the dynamic space.
- *Kind of string*: Previous experiments indicated that up to 70% of the automaton's states were accessed during acceptance testing [3], when processing the kind of randomly generated accepting strings that we used for the current experiment. Therefore, dedicating only 50% of those states to the dynamic memory space apparently does not guarantee sufficient improvement on data organization. Instead, for the particular strings generated, excessive overheads are incurred, with consequent poor performance.

In an attempt to take into account the fact that replacement policy is a performance bottleneck, the TD-AVC algorithm was implemented such that no replacement was made when the cache was full. This approach resulted in TD-AVC competing with it core TD counterpart as shown in figure 1-IV. Again, since up to 70% of the automaton's state are visited, we merely have 20% of the states that are processed out of cache whiles the remaining are processed in cache. This observation clearly shows that the strategy is competitive under appropriate circumstances. However, the range of data that will provide better cache utilization requires further investigation. Furthermore, even better performance of TD-AVC over the TD is expected if the cache size increases at about 70% of the total number of states, since most of the states would be contiguously organized. The graphs in figure 1-III also reveal that the TD-SpO strategy outperforms its TD counterpart. It should be noted that we did not take into account the time taken to reorder the states, since this is regarded as a once-off pre-processing activity. The results confirm that pre-ordering of the state rows can indeed bring about performance improvements. This will be at an optimal when pre-ordering maximizes spacial and temporal locality of reference.

Figure 2 depicts the graphs of the various algorithms obtained by combining the implementation strategies. As may be observed, the combination of the SpO strategy with the unbounded TD-DSA yields better performance over the core TD algorithm. This also applies to algorithms t_{23} (this version used direct mapping for replacement). Again, the pre-ordering of states apparently results in better data organization, and hence better cache utilization. However, for algorithm t_{u123}, the AVC strategy re-

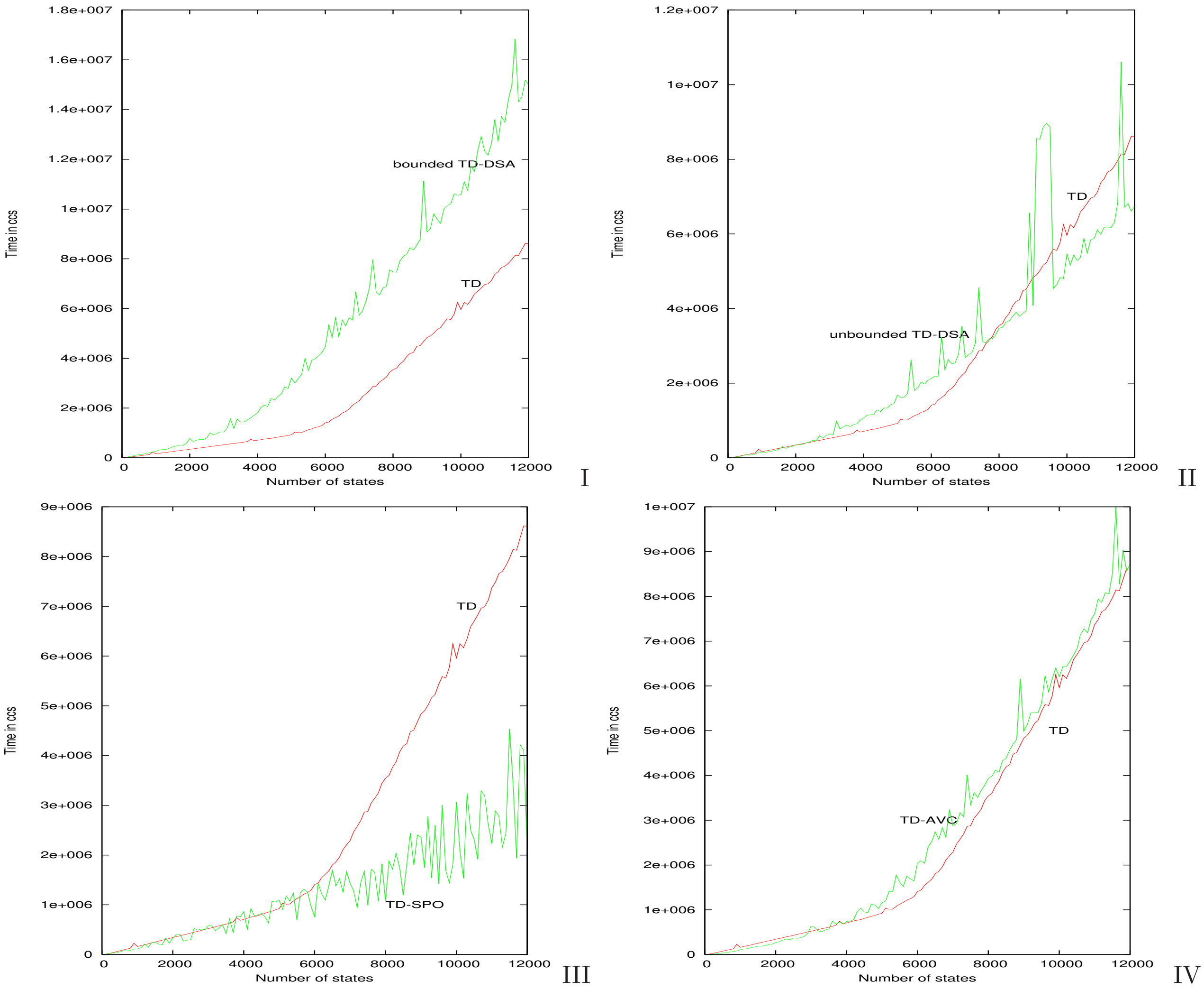


Figure 1. Performances of TD vs DSA (I & II), AVC (III), and SPO (IV)

lied on direct mapping for replacement; this resulted in overheads and therefore poor cache utilization, and hence poor performance. It is expected that by choosing suitable strings, a better performance would be observed. For algorithm t_{b123}, the bounded nature of both DSA and AVC strategies required replacement, resulting in various overheads; this explains its poor performance over the TD algorithm. Again, suitable data set would produce better result. The combination of both the DSA strategy and the AVC strategy also suffered from the overheads caused by the direct mapping replacement policy. Therefore, in order to improve the performance, a better replacement policy should be chosen or we may even avoid it totally. The conclusion and further direction to this contribution are depicted in the next section.

5 Conclusion and Future Work

In this paper, we have investigated various ways of improving performances of the conventional TD algorithm using various implementation strategies to which were associated parameter arguments. A 6-argument function provided the denotational semantics of various TD FA-based string recognizers. Alternative instantiations of

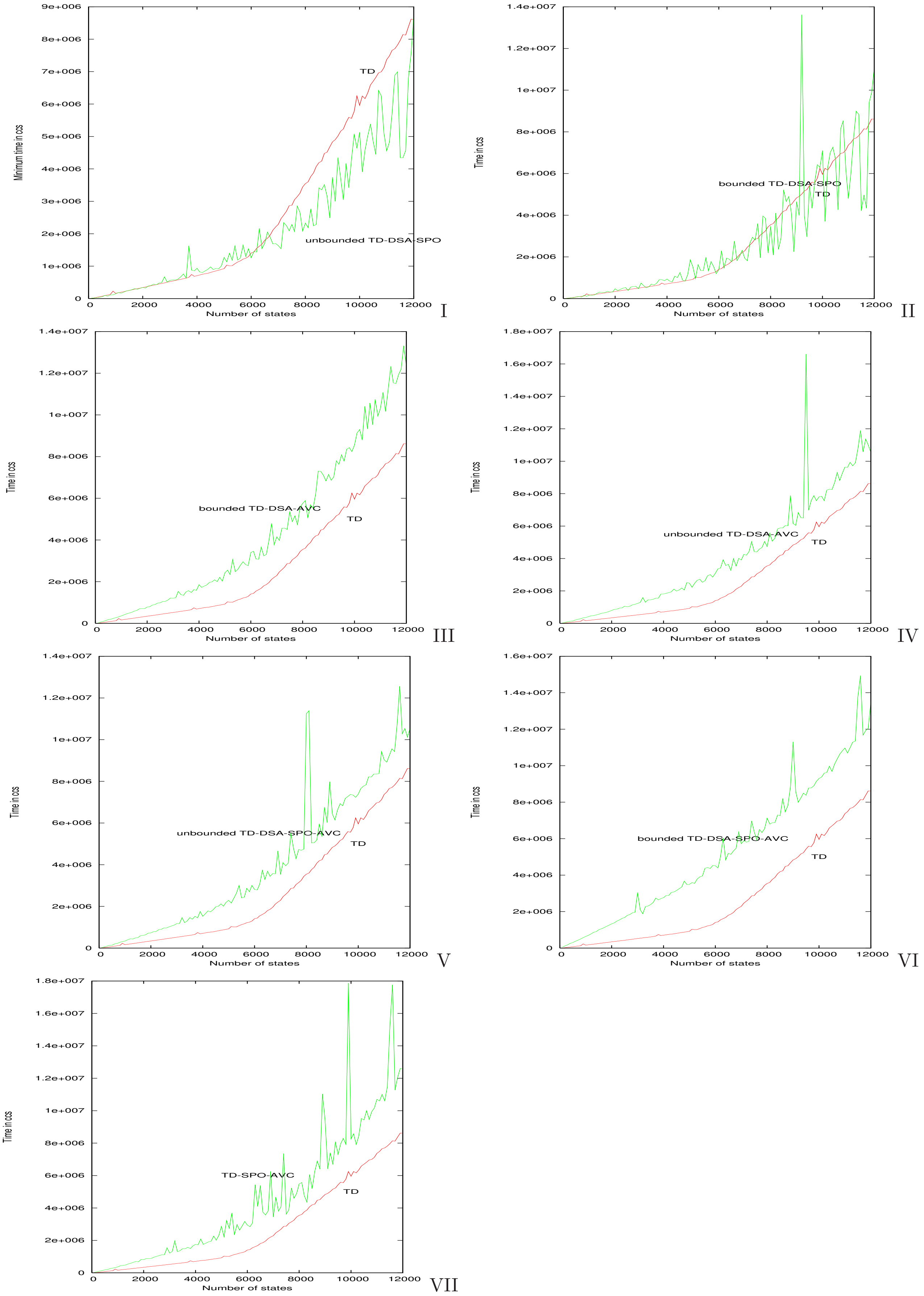


Figure 2. Performances of TD vs DSA-SPO (I & II), DSA-AVC (III & IV), DSA-SPO-AVC (V & VI), and SPO-AVC (VII)

these arguments represented new TD-based algorithms. The algorithms were then implemented and performance recorded. It was shown that, based on the strings made of long repeated sequences, some of the algorithms outperformed the traditional TD algorithm, but the others were not of interest due to the appropriateness of the kind of string considered and the policy used for replacement.

The algorithms were tested by relying on artificially generated data (strings and automata). Thus as a matter of future work, various experiments will be conducted on real life data such as genetic sequences, micro-satellites for tandem repeat detection, network intrusion detection, and the like. We also wish to further explore the string characteristics and appropriate sizes for dynamically allocated space (in the case of DSA) and virtual cache (in the case of AVC) respectively.

The algorithms presented in this work are part of a toolkit that is under construction for FA-based string processing, targeted at applications that use FA-based string recognition as part of their model solution.

References

1. J. P. Hayes: *Computer Architecture and Organization*, McGraw-Hill, third ed., 1998.
2. E. N. Ketcha: *Hardcoding finite automata*, Master's thesis, University of Pretoria, Department of computer Science, Pretoria 0002, South Africa, November 2003.
3. E. N. Ketcha, D. G. Kourie, and B. W. Watson: *Reordering finite automatata states for fast string recognition*, in In Proceeding of the Prague Stringology Conference, Prague, Czech Republic, August 2005, Czech Technical University.
4. E. N. Ketcha, D. G. Kourie, and B. W. Watson: *Dynamic allocation of finite automatata states for fast string recognition.* International Journal of Foundation of Computer science, 2006.
5. D. E. Knuth, J. H. Morris, Jr, and V. R. Pratt: *Fast pattern matching in strings.* SIAM Journal on Computing, 6(3) 1977, pp. 368–387.
6. B. Meyer: *Introduction to the Theory of Programming Languages*, Prentice Hall, c.a.r hoare series ed., 1990.
7. D. A. Patterson and J. L. Hennessy: *Computer Organization and Design*, Morgan Kaufmann, third ed., 2005.
8. K. Thompson: *Regular expression search algorithm.* Communications of the ACM, 11(6) 1968, pp. 323–350.
9. A. C. Yao: *The complexity of pattern matching for a random string.* SIAM Journal on Computing, 8(3) 1979, pp. 368–387.

A Markovian Approach for the Analysis of the Gene Structure

Christelle Melo de Lima[1], Laurent Guéguen[1], Christian Gautier[1] and Didier Piau[2]

[1]UMR 5558 CNRS Biométrie et Biologie Evolutive, Université Claude Bernard Lyon 1
43 boulevard du 11-Novembre-1918
69622 Villeurbanne Cedex 69622 – France.
melo@biomserv.univ-lyon1.fr
[2] Institut Camille Jordan UMR 5208, Université Claude Bernard Lyon 1
Domaine de Gerland, 50 avenue Tony-Garnier
69366 Lyon Cedex 07 – France.

Abstract. Hidden Markov models (HMMs) are effective tools to detect series of statistically homogeneous structures, but they are not well suited to analyse complex structures. Numerous methodological difficulties are encountered when using HMMs to segregate genes from transposons or retroviruses, or to determine the isochore classes of genes. The aim of this paper is to analyse these methodological difficulties, and to suggest new tools for the exploration of genome data. We show that HMMs can be used to analyse complex genes structures with bell-shaped distributed lengths, modelling them by macro-states. Our data processing method, based on discrimination between macro-states, allows to reveal several specific characteristics of intronless genes, and a break in the homogeneity of the initial coding exons. This potential use of markovian models to help in data exploration seems to have been underestimated until now, and one aim of our paper is to promote this use of Markov modelling.

Keywords: HMM, macro-state, gene structure, $G + C$ content

1 Introduction

The sequencing of the complete human genome led to the knowledge of a sequence of three billion pairs of nucleotides [19]. Such amounts of data make it impossible to analyse patterns or to provide a biological interpretation analysis unless one relies on automatic data-processing methods. For twenty years, mathematical and computational models have been widely developed in this setting. Numerous methodological efforts have been devoted to multicellular eukaryotes since a large proportion of their genome has no known function. For example, only 3% of the human genome is known to code for proteins. Another difficulty is that the statistical characteristics of the coding region vary dramatically from one species to the other, and even from one region in a given genome to the other. For example, vertebrate isochores ([29], [3]) exhibit such a variability in relation to their $G + C$ frequencies. Thus it is necessary to use different models for different regions if one seeks to detect patterns in genomes.

A classical way of modelling genomes uses hidden Markov Models (HMMs) ([22], [18], [23]). To each type of genomic region (exons, introns, etc.), one associates a state of the hidden process, and the distribution of the stay in a given state, that is, of the length of a region, is geometric. While this is indeed an acceptable constraint as far as intergenic regions and introns are concerned, the empirical distributions of the lengths of exons are clearly bell-shaped ([6], [2], [17]), hence they cannot be represented by geometrical distributions. Semi-Markov models are one option to overcome this problem [6]. These models are very versatile, since they allow to adjust the distribution of the duration of the stay in a given state directly to the empirical distribution. The trade off is a strong increase in the complexity of most algorithms implied by the estimation and the use of these models. For example, the complexities of the main algorithms (forward-backward and Viterbi) are quadratic in the worst case with respect to the length of the sequence for hidden semi-Markov chains and linear for HMMs ([6], [27], [15]). This may limit their range of application as far as the analysis of sequences with long homogeneous regions is concerned. Another difficulty is the multiplication of the number of parameters that are needed to describe the empirical distributions of the durations of the states, and which must be estimated, in addition to usual HMM parameters [27]. Thus the estimation problem is more difficult for these variable duration HMMs than for standard HMMs [27].In other words, semi-Markov models are efficient tools to detect protein genes, but they are much more complex than HMMs. We suggest to use HMM for modelling the exon length distribution by sum of geometric laws. To do this a state representing a region is replaced by a juxtaposition of states with the same emission probabilities. This juxtaposition of states is called macro-states.

The modelling of a gene may be used to annotated complete genomes, as Genscan [6] in Ensembl, but also to explore data in order to detect exceptional patterns and to help in their biological interpretation. Thus, the use of Markov models for the purpose of data exploration has been underestimated in genome analysis. This objective requires simple parameters and a relative small amount of computer resources, to be able to perform numerous analyses of the data. For this purpose, we show how to use macro-states HMMs models for complete genome analysis.

2 Materials

Gene sequences were extracted from Hovergen (Homologous Vertebrate Genes Database) [11] for the human genome. To ensure that the data concerning the intron/exon organisation was correct, we restricted our analysis to genes of which the RNA transcripts have been sequenced. To avoid distortion of the statistical analysis, redundancy was discarded. This procedure yielded a set of 5034 multi-exon genes and 817 single-exon (that is, intronless) genes. To simplify the model, UTRs (including their introns) were not separated from intergenic regions. As a consequence, in the present paper, the word "intron" means an intron which is located between two coding exons.

The statistical characteristics of the coding and noncoding regions of vertebrates differ dramatically between the different isochore classes [4]. The isochore has been classified as a "fundamental level of genome organisation" [13] and this concept has increased our appreciation of the complexity and variability of the composition of eukaryotic genomes [25]. Many important biological properties have been associated with the isochore structure of genomes. In particular, the density of genes has been shown to be higher in H- than in L isochores [24]). Genes in H isochores are more compact, with a smaller proportion of intronic sequences, and they code for shorter proteins than the genes in L isochores [12]. The amino-acid content of proteins is also constrained by the isochore class: amino acids encoded by $G + C$ rich codons (alanine, arginine ...) being more frequent in H isochores ([10], [9]). Moreover, the insertion process of repeated elements depends on the isochore regions. SINE (short-interspersed nuclear element) sequences, and particularly Alu sequences, tend to be found in H isochores, whereas LINE (long-interspersed nuclear element) sequences are preferentially found in L isochores [20]. Thus, we took into account the isochore organisation of the human genome. Three classes were defined based on the $G+C$ frequencies at the third codon position ($G+C_3$). The limits were set so that the three classes contained approximately the same number of genes. This yielded classes H=[100%, 72%], M=]56%,72%[and L=[0%,56%], which were used to build a training set. These classes were roughly the same as those used by other authors ([24], [30]). These sets were used to model the distributions of the lengths of the exons and the introns, and to analyse the structure of genes.

3 Methods

3.1 Estimation of the parameters

Estimation of emission probabilities: The DNA sequence is heterogeneous along the genome, but it consists of a succession of homogenous regions, such as coding and non-coding regions. HMMs are used to distinguish between these different types of regions.

Exons consist of a succession of codons, and each of the three possible positions in a codon (1, 2, 3) has characteristic statistical properties. This implies the need to divide exons into three states ([7], [5]). HMMs take into account the dependency between a base and its n preceding neighbours. In this case, the order of the model is n. For our study, n was taken to be equal to 5, as in the studies of Borodovsky and Burge ([7], [5]). The emission probabilities of the HMM were therefore estimated from the frequencies of 6-letter words in the different regions (intron, initial exon, internal exons and terminal exon) that made up the training set. Even if introns have not codon structure, the use of 6-letter words allow to improve the discrimination between coding and no-coding region. Therefore there is an HMM for each region.

Thus, the emission probabilities of the model were estimated by using the maximum likelihood method in order to highlight why some sequences are not correctly predicted although it is the case for other sequences of the same region. In other words, we relied on the error of predictions of the HMMs, rather than analyse somewhat blindly the genomes to do an exploration of the human genome.

Estimation of the structure of the macro-states: An alternative to the semi-Markov models is suggested to model the bell-shaped empirical length distributions of the exons. We propose to use sums of a variable number of geometric laws with equal or different parameters. Thus a "biological state" is represented by a HMM and not by a single Markov state. The emission of probabilities of every state in this HMM are the same. A key property of this macro-state approach is that the conditional independence assumptions within the process are preserved with respect to HMMs. Hence, the HMM algorithms to estimate the parameters and compute the most likely state sequences still apply [15]. The length distribution of the exons and introns was estimated from the training set (data set sequences are named $x_1 \dots x_n$). Each x_i was considered to be the realization of an independent variable of a given law. We tested the following laws:

- The sum of m geometric laws of same parameter p (*i.e.* a binomial negative law):

$$P[X=k] = C_{k-1}^{m} p^m (1-p)^{k-m}$$

– The sum of two geometric laws with different parameters $p_1 > p_2$:

$$P[X = k] = p_1 \times p_2 \times \frac{(1-p_2)^{k-1} - (1-p_1)^{k-1}}{p_1 - p_2}$$

– The sum of three geometric laws with different parameters $p_1 < p_2 < p_3$:

$$P[X=k] = \frac{p_1 \times p_2 \times p_3}{p_2 - p_3} \times \left(\frac{(1-p_1)^{k-1} - (1-p_3)^{k-1}}{p_3 - p_1} - \frac{(1-p_2)^{k-1} - (1-p_3)^{k-1}}{p_3 - p_2} \right)$$

To estimate the parameters of the different laws, we minimised the Kolmogorov-Smirnov distance for each law. The law which fits best with the empirical distribution is the law with the smallest Kolmogorov-Smirnov distance. However, the classical Newton or gradient algorithm cannot minimise for the Kolmogorov-Smirnov distance, because this distance cannot be differentiable. We therefore discretised the parameter space with a step of 10^{-5}, and fixed the minimum value. Parameter estimations were not based on the maximum likelihood, which would have matched the end of the exon length distribution thus neglecting many small exons (Figure 1 a). Indeed, that it is for a geometrical law or a convolution of geometrical laws, the parameter p is estimated by the reverse of the mean ($E[X] = 1/p$) by the method of the maximum likelihood. The extreme values thus tend to stretch the distribution towards the large ones. We therefore have preferred to use the Kolmogorov-Smirnov distance in order to obtain a better modelling of the human gene. Again, in order to provide simple but efficient models, equal transitions between states of a macro-state were favoured when it was possible.

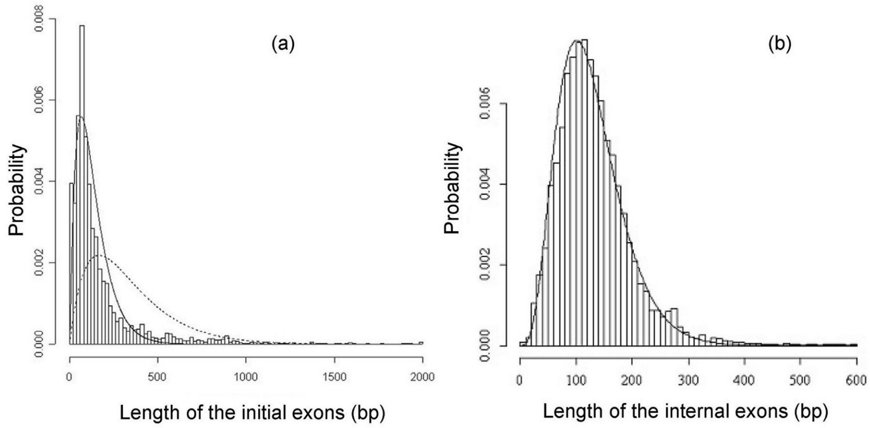


Figure 1. (a) The histogram represents the empirical distribution of the length of the initial exons in a multi-exons gene. The dotted line describes the theoretical distribution, obtained from the Kolmogorov-Smirnov distance. The continuous line characterises the binomial distribution, obtained by the method maximum likelihood. (b) The histogram represents the empirical distribution of the length of the internal exons. The dotted line describes the theoretical distribution, obtained from the Kolmogorov-Smirnov distance.

Thus, a region is represented by a hidden state of the HMM. If the length distribution of a region is fitted by a sum of geometric laws, the state representing the region is replaced by a juxtaposition of states with the same emission probabilities, thus leading to macros-states (Figure 2). The state duration is characterised by the parameters of the sum of these geometric laws. Various studies ([6], [28], [8]) have shown that the length distribution of the exons depend on their position in the gene. We took all exon types into account: initial coding exons, internal exons, terminal exons and single-exon genes.

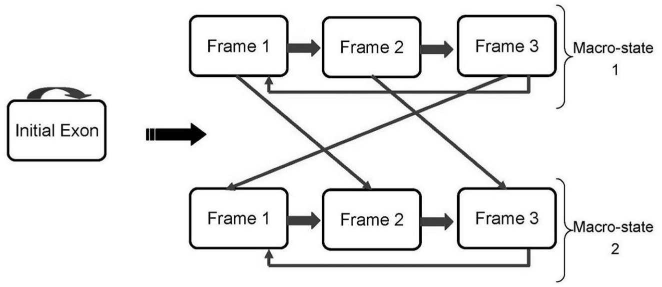


Figure 2. Figure representing initial exon HMM

3.2 Models selection

Algorithm of Models selection In order to measure the adequacy of a model with a genomic region, the theory of HMMs proposes two solutions: the probability of the observed sequence conditioned by the optimal trajectory in the hidden states (Viterbi) or the probability of the sequence x under the model M, $P[x|M]$.

The first method neglects the fact that many trajectories are biologically equivalent. The second method sums the probabilities corresponding to internal structures of a sequence, which were different. Thus, a model that predicts a bad internal structure can be associated to a high value of the probability. For example, these two techniques of selection of models in the context of HMMs were compared:

We consider a HMM of type $M1M0$ with 2 states, called A and B, and 2 observations, called 0 and 1. We assume that the transition probabilities from A to B and from B to A are both $t = 9.53643.10^{-7}$, and that A emits 0, respectively B emits 1, with probability p. We note $M_{0.9}$ the HMM with the probability $p = 0.9$ and $M_{0.6}$ the HMM with $p = 0.6$. We choose the sequence $x = 0^n1^n$ for given value of n=10, the aim is to choose $M_{0.9}$ and $M_{0.6}$.

If the maximisation of the probability of the sequence was used, it is needed to compute $P(x|M)$:

$$P(x|M_{0.9}) = 6.97.10^{-11} < P(x|M_{0.6}) = 1.27.10^{-6}.$$

In this case the model $M_{0.6}$ is better than the model $M_{0.9}$.

If the probability of the observed sequence conditioned by the optimal trajectory in the HMM was used, it is needed compute the probability given by the Viterbi algorithm: $P(x/s_{op}, M)$. For the models $M_{0.9}$ and $M_{0.6}$, the optimal sequence is composed of n states A followed by n states B. In general case ($0.5 < p < 1$), this probability is:

$$P(x|s_{op}, M_{0.9}) = 0.1215 > P(x|s_{op}, M_{0.6}) = 3.65.10^{-5}.$$

Thus, $M_{0.9}$ is better than $M_{0.6}$. This very schematic example shows opposite conclusions for the two methods and amphases the fact none of these approaches has a universal validity. On the other hand, if we consider HMMs that correspond to a macro-state, the situation is biologically clearer. All trajectories in a macro-state are biologically equivalent. The method of the optimal trajectories is therefore not adapted to this problem, while, the situation is well described by the probability of the sequence under the model. Thus, the probabilities that are summed correspond to the same biological structures.

Analysis of the gene structure using HMMs selection The use of HMMs for classifying sequences raises the question of the evaluation of their discriminating power. The method chosen here is to split the set of sequences of known nature into two sets: one for training and one to compare the different models.
All HMMs (introns, initial exons, internal exons and terminal exons models) are then compared pairwise for all the sequences in a given type of region (intron, initial exon, internal exon and terminal exon sequences) of the test set, in order to identify the model which is the most likely to represent the test sequence. This gives the discrimination measure D, with

$$D = P(S/HMM_1)/P(S/HMM_2),$$

where S is the sequence being tested, and HMM_1, HMM_2 are the two models tested. The computations were realized with the package SARMENT [16]. The best HMM for most of the sequences in a given region is used to characterise this region. Each model is finally characterised by the frequency with which it recognises the sequences. This approach allows to show the types of sequences that were not well recognised by their corresponding model. Finally, the analysis of the different types of exons was completed by a correspondence analysis.

4 Results – Discussion

4.1 Inclusion of explicit distributions of the durations of the states in HMMs

In order to model the bell-shaped empirical length distributions of exons (Figure 1), we have used sums of geometric distributions with equal or different parameters. The length of an exon depends on its position within the gene. Initial and terminal exons tend to be longer than internal ones (Table 1). The length of introns displays also a noticeable positional variability. The distributions of the lengths of internal and terminal introns are relatively similar, but these types of introns are both smaller than the initial introns (Table 1). As is well known, the lengths of exons and introns depend on their $G + C$ content [8]. Table 1 shows that the $G + C$ frequency at the third codon position is negatively correlated with the length of the introns, *i.e.*, high frequencies correspond to short introns, and vice versa. The initial exons are longer in $G+C$ rich regions (i.e. displays a significant Wilcoxon non-parametric test). However, the length of the internal and terminal exons does not vary with the class of isochores (*i.e.* displays a non significant Wilcoxon non-parametric test). The length of the exons displays clearly a bell-shaped pattern, for the three $G + C$ classes. Since the minimisation of the Kolmogorov-Smirnov distance yields a good fit with the empirical distribution of the length of the exons (Figure 1 and Table 1), we used it to model their length distribution by a sum of geometric laws and estimated the parameters of these laws (see Method for a comparison with the maximum likelihood approach).

Position in the gene	Length (bp) in class H		Length (bp) in class M		Length (bp) in class L	
	Mean	Median	Mean	Median	Mean	Median
Initial coding exon	223	123	176	102	160	87
Internal exon	144	126	143	125	144	120
Terminal exon	244	165	237	145	218	138
Initial intron	4027	3189	4139	3540	5315	4857
Internal intron	1461	958	1767	1310	2850	2433
Terminal intron	1394	884	1764	1282	2819	2415

Table 1. Length of the exons and of the introns according to their position in the gene and according to their $G+C$ frequency at third codon position

Laws	Parameters p	K-S distance
$G_2(p)$	0.0117	0.1084
$G_3(p)$	0.0185	0.16
$G_4(p)$	0.02634	0.1826
$G(p_1,p_2)$	0.0055-0.087	0.0447

Table 2. Parameters estimation of different laws obtained for initial exons of class H minimising Kolmogorov-Smirnov distance (K-S)

We define $G_n(D_1,...,D_n)$ as the distribution of the sum of n random variables of geometric distributions, each with expectation D_i and parameter $p_i = 1/D_i$. Thus the expectation of $G_n(D_1,...,D_n)$ is $D_1 + ... + D_n$. When $D_i = D$ for every i, this is called a negative binomial distribution with parameters (n,1/D), which we denote $G_n(1/p)$. Finally $G_n(D)$ is a geometric distribution with expectation D and parameter $p = 1/D$, which we write $G(D)$.

We show here only the results for the modelling of the distributions of the lengths in the H class. However, the distributions of the lengths in the classes M and L can be modelled by sums of geometric laws. The estimated distributions are $G_2(58.82, 74.07)$ for initial exons (Figure 1 a), $G_3(86.21, 181.81, 10)$ for terminal exons, $G_5(26.32)$ for internal exons (Figure 1 b), $G_3(351.11)$ for intronless genes, and the geometric distribution $G(111.11)$ for initial introns. Other types of introns are also modelled by a geometrical distribution.

The distributions of the lengths of the single exons (that is, the intronless genes) exhibit a clear bi-modality (Figure 3). By using the software Blast [1] to search regions of the human genome similar to our intronless genes, we have found that many small intronless genes are often repeated along the human genome. The comparison of all these repeated intronless genes to a database of pseudogenes [21] revealed that many small intronless genes are actually pseudogenes, *i.e.*, genes that have lost their function. After the elimination of these pseudogenes, the distribution of the lengths of the real intronless genes is bell-shaped, like the distributions for the other types of exons.

4.2 Evaluation of the models

The macro-states used for initial exons (M_E1), internal exons (M_Eint) and terminal exons (M_Eter) were evaluated. In order to compare the models two-by-two, the likelihoods of each sequence of a given type (initial exons, internal exon, etc.) with respect to the two models were compared and the model with the greater likelihood is voted for by this sequence (see Method). For example, the sequences of initial exons

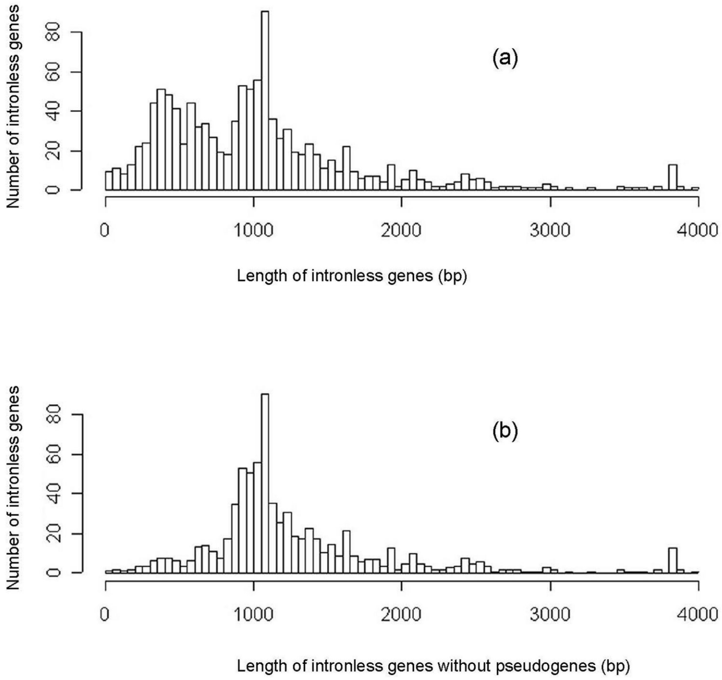


Figure 3. (a) The histogram represents the empirical distribution of the length of the intronless genes. (b) The histogram represents the empirical distribution of the length of the intronless genes without pseudogenes.

vote between the model for the initial exons (M_E1) and the model for the internal exons (M_Eint), assuming roughly equal proportions (Figure 4, Histogram 1). In conclusion, on the sequences of initial exons, the models M_E1 and M_Eint have similar predictive powers. Figure 4 gives results for the isochore class H. We stress the following points.

1. Internal exons and terminal exons share similar statistical properties. This is shown by the similar predictive powers of the models M_Eint and M_Eter (Figure 4, Histograms 4 and 6).
2. The initial exons are clearly discriminated from the other exons. This is shown by the smaller likelihood of the internal exons in M_E1 than in M_Eint (Figure 4, Histograms 3 and 5).
3. The modelling of the initial exons is inadequate. This is shown by the small likelihood of the initial exons in M_E1 (Figure 4, Histograms 1 and 2).

The specific statistical characteristics of the initial exons might result from the existence of signals located at, or covering, the beginning of the genes. To explore this hypothesis, we have split our HMM for the initial exons into two HMMs. The first one models the first n nucleotides of the initial exon, and the second the remaining part of the initial exon. This new initial exon model is called M_E1_n. Pairwise comparisons between the models M_E1_n for various values of n (Figure 5) show that the M_E1_{80} model yields the better discrimination. This suggests that the break of homogeneity in the initial exon happens around the 80th base. Finally, this separation provides a better discrimination between the models of the internal and initial exons on the one hand and the model of the initial exons on the other hand (49% to 61% in favour of

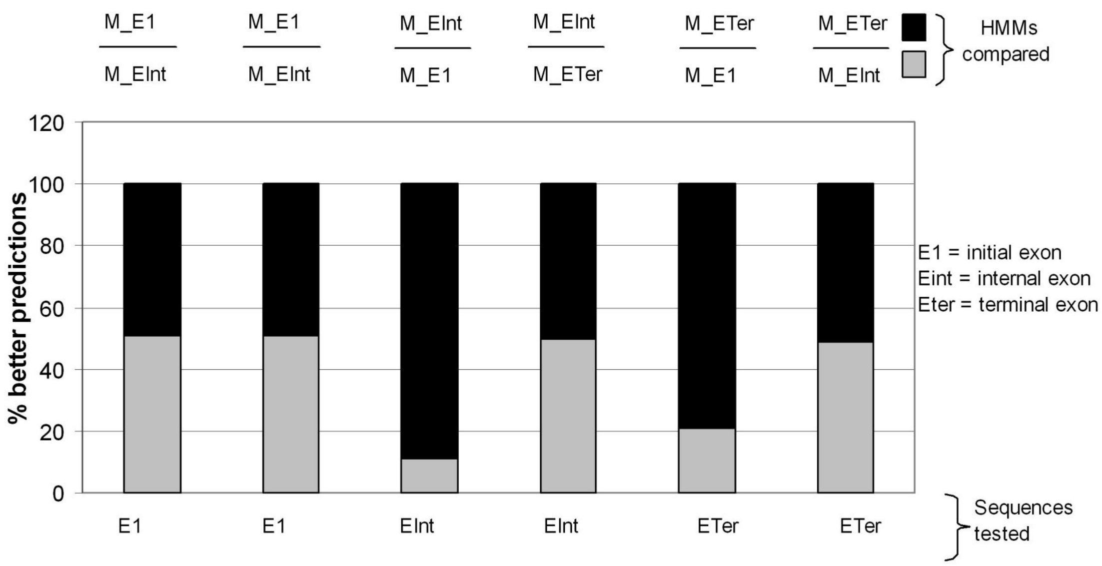


Figure 4. Models learned from different sequences (initial, internal and terminal exons) were compared pairwise using the sequences used to determine the best predictions. For instance, in histogram 1, the likelihood of each first exon was computed using models learnt on $E1$ and $Eint$. The black bar represents the percentage of first exons having a higher likelihood for the first exon model, and the grey bar those with a higher liklihood for the second exon model. Histograms 1-2: The models $E1$, $Eint$ and ET have same predictive power on initial exons. Histograms 3-5: The models $Eint$ and ET provide a good prediction of the internal and terminal exons compared to the $E1$ model (82% and 75%, respectively). Histograms 4-6: The models $Eint$ and ET have the same predictive power for initial and terminal exons.

the M_E1_{80} model [Figure 4, histogram 1 and Figure 5, histogram 7]) and from the internal exons (89% to 92% in favor of M_Eint, not shown in the Figure). Similar results were found for the terminal exons.

The break in the homogeneity of the first exon could be explained by the presence of a signal peptide. The first exons which contain a signal peptide are better recognised by the first HMM of the M_E1_{80} model than by the second one in 75% of the cases. These results were also compared with those obtained by SignalP [26]. The initial exons which, according to SignalP, contain a signal peptide, were more accurately recognised by the M_E1_{80} model than by the internal exon model in 70% of the cases. When SignalP does not predict a signal peptide, the M_E1_80 and the internal exon models yield similar results.

The significance of the modelling of isochores is highlighted by the results described in the previous paragraph, which show the effect of the distributions of the lengths of exons and introns. This claim was confirmed by our study of the influence of the isochore class on the words frequencies, in the different types of regions. For every type of exons (*i.e.*, initial, internal and terminal), the model trained with a specific isochore class performed better on this class than the others (Figure 6). The situation as concerns the classes of introns is somewhat different. The introns from classes H and M are better predicted by our HMMs H and M, respectively (Figure 7, Histograms 1 to 4), whereas the three models H, L, and M, are more or

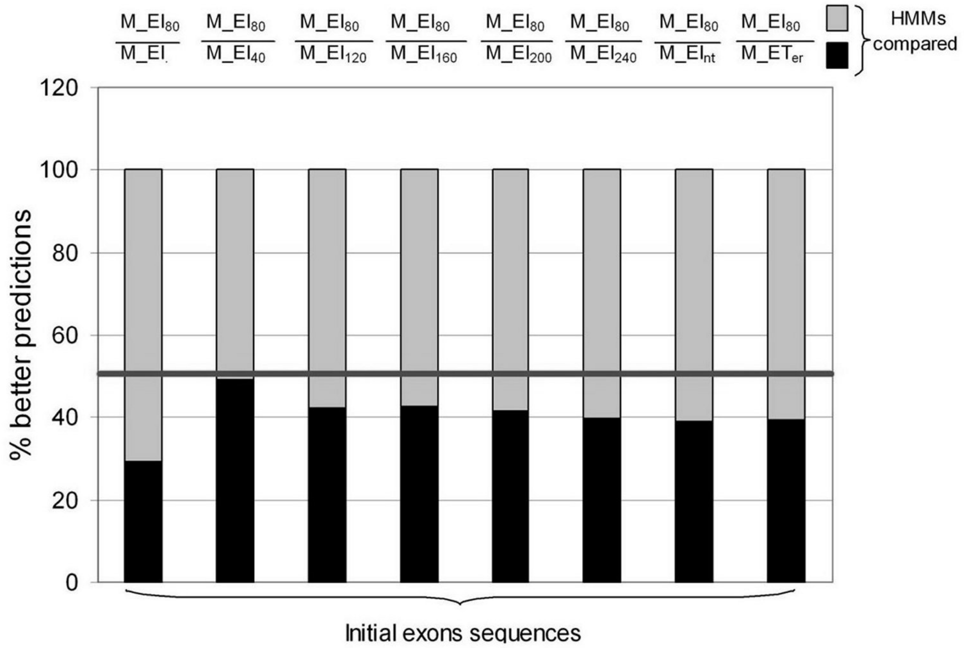


Figure 5. The models learned from different sequences (initial, internal and terminal exons) were compared pairwise to the initial exons to identify the best predictions. The $IE80$ model provides a better prediction of initial exons than any other model tested.

less equivalent for the introns of class L (Figure 7, Histograms 5, 6). This analysis clearly reveals some major statistical differences between the three isochore classes, and the importance of taking into account this heterogeneity of the genome in a context of prediction of genes. The poor recognition of introns in L isochores by all these

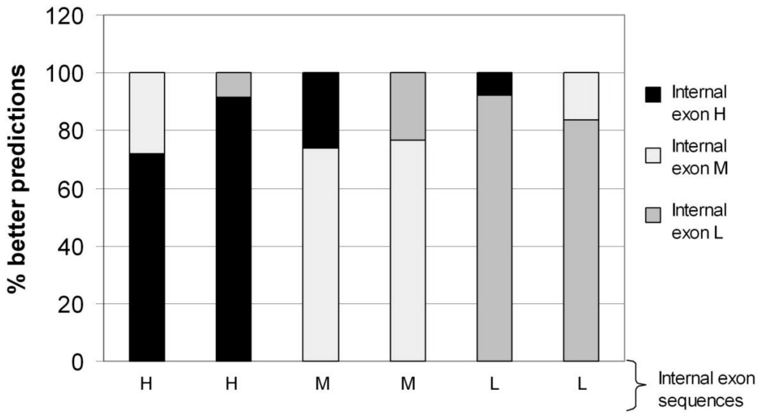


Figure 6. The models learned from different sequences (internal exons of classes H, M and L) were compared pairwise on the same sequences to determine the best predictions.

models might result from an over-simplistic modelling. We point out that repeated elements, particularly LINEs, were not taken into account. Their higher frequency in the isochores of class L could explain the response of the model.

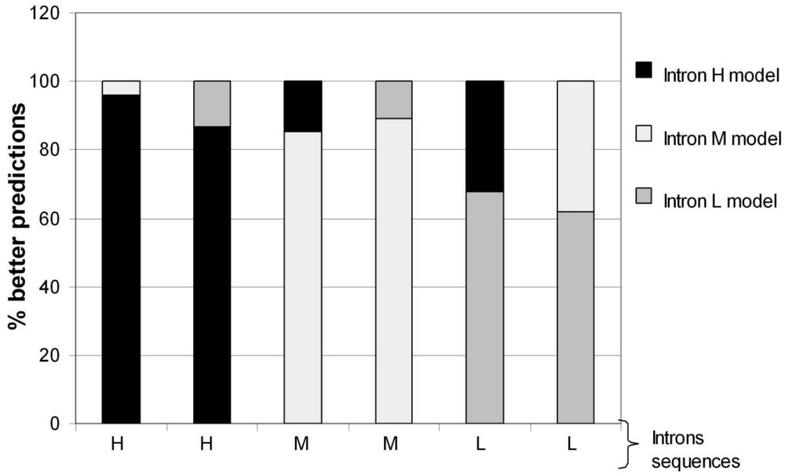


Figure 7. The models learned from different sequences (introns of classes H, M and L) were compared pairwise on the same sequences to determine the best predictions.

Many other data exploration tools exist. Multivariate analysis is on among the most popular methods that uses exactly the same data as HMMs. Indeed, if sequences are represented by frequencies of 6-bases words (see method), then a correspondence analysis will take into account exactly the same data as the one which is used to estimate the parameters of an HMM (see method). Figures 8 and 9 show the general patterns found by correspondence analysis. The frequencies of words of length 6 in the exons and the introns are neatly divided into four groups: H exons, M exons, L exons, and introns (Figure 8). When the reading frames are also taken into account (Figure 9), they are separated on the first factor, showing that the statistical differences between the codon positions represent the main statistical pattern in coding sequences.

5 Conclusion

The use of Markov models for the purpose of data exploration has been underestimated in genome analysis. This study is the first large scale exploration of the use of macro-states. Our approach allows to discriminate most genomic regions and is based on a selection among HMM models using macro-states. Macro-states allow to model distributions of lengths which are not geometric. Our strategy yields a comprehensive description of the human genome that highlights the following features:

1. The particular structure of intronless genes revealed the large number of errors of annotation in the databases for these genes: most small intronless genes are actual pseudogenes.
2. The great statistical differences between the three classes of isochores, and therefore the importance of taking into account this heterogeneity of the genome for the purpose of gene prediction. Initial exons exons are longer in the H class ($G+C$ rich). Introns are longer in the L class ($G+C$ poor).

3. Initial exons exhibit a very specific pattern, due to the fact that half of them contain a peptide signal. An average duration of stay in the first state of M_E180 of 80 bases long was observed, this is consistent with biological knowledge about such signals, which are 45 to 90 bases long. Initial exons without a peptide signal, and the second parts of the initial exons with a peptide signal, are statistically similar to internal exons and terminal exons, respectively.

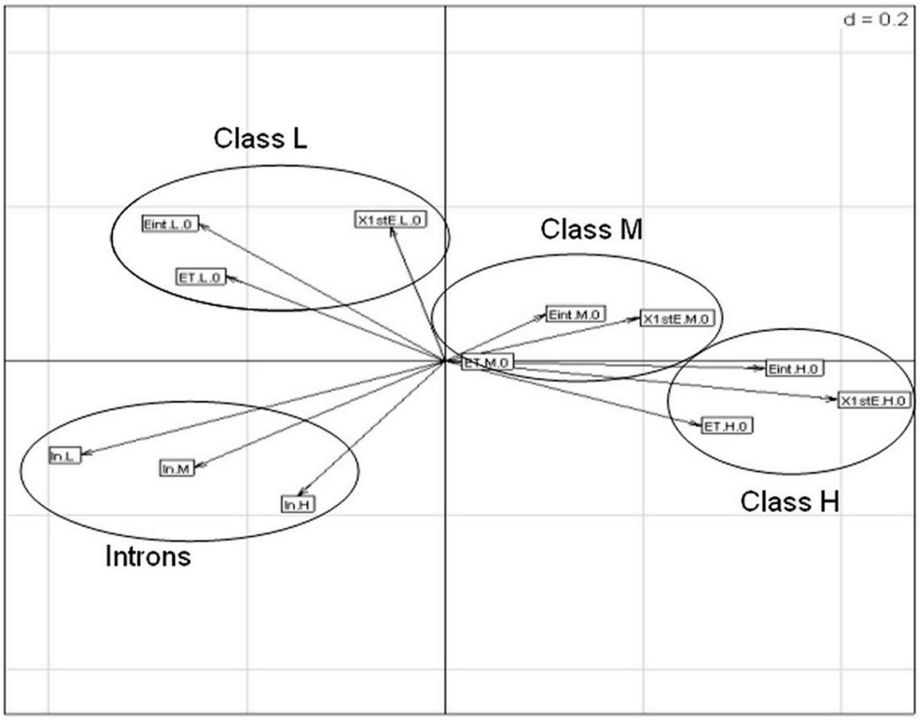


Figure 8. Correspondence analysis of the emission probabilities of the different states models in reading frame 0. The first axis (36.2% of total variability) represents the $G+C$ gradient. $Eint.H.0$=internal exon model of class H and reading frame 0; $Eint.M.0$=internal exon model of class M and reading frame 0; $Eint.L.0$=internal exon model of class L and reading frame 0; $ETt.H.0$=terminal exon model of class H and reading frame 0; $ETt.M.0$=terminal exon model of class M and reading frame 0; $ETt.L.0$=terminal exon model of class L and reading frame 0; $First.E.H.0$=initial exon model of class H and reading frame 0; $first.E.H.0$=initial exon model of class M and reading frame 0; $first.E.H.0$=initial exon model of class L and reading frame 0. $IN.H$=intron model of class H; $IN.M$=intron model of class M; $IN.L$=intron model of class L.

Macro-states HMMs models are based on exactly the same data as. Multivariate analysis but allows to identified the general patterns with a much lower cost in CPU resources. This is very close to the principle of some "old" gene prediction methods (see RECSTA [14]). However, the markovian approach has important advantages: it is not necessary to know the limits of the regions before the analysis, and more importantly, the model is more versatile; hence, new hypotheses can be explicitly introduced, as was done for the signal peptide.

6 Acknowledgements

The computations have been made at the IN2P3 computer centre using a large computer farm (more than 1500 CPUs). We thanks Marie-France Sagot for assistance in preparing and reviewing the manuscript.

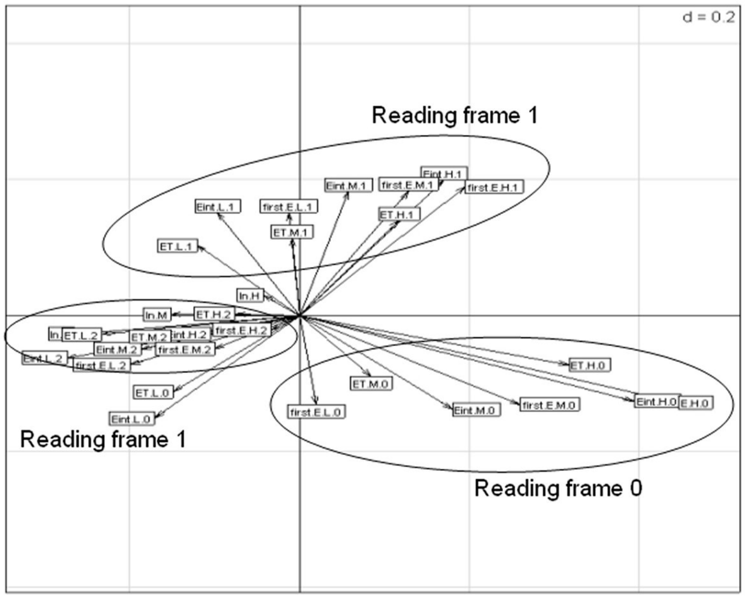


Figure 9. Correspondence analysis of the emission probabilities of the different state models. The first axis (59.5% of the total variability) represents the reading frame gradient.

References

1. S. Altschul, W. Gish, E. Myers, and D. Lipman: *Basic local alignment search tool.* J. Mol. Biol., 1990, pp. 215–410.
2. S. Berget: *Exon recognition in vertebrate splicing.* The Journal of Biological Chemistry, 270(6) 1995, pp. 2411–2414.
3. G. Bernardi: *Isochores and the evolutionary genomics of vertebrates.* Gene, 241(1) 2000, pp. 3–17.
4. G. Bernardi, B. Olofsson, J. Filipski, M. Zerial, J. Salinas, G. Cuny, M. Meunier-Rotival, and F. Rodier: *The mosaic genome of warm-blooded vertabrates.* Science, 228(4702) 1985, pp. 953–958.
5. M. Borodovsky and J. McIninch: *Recognition of genes in dna sequences with ambiguities.* Biosystems, 30(1-3) 1993, pp. 161–171.
6. C. Burge and S. Karlin: *Prediction of complete gene structure in human genomic dna.* Journal of Molecular Biology, 268 1997, pp. 78–94.
7. C. Burge and S. Karlin: *Finding the genes in genomic dna.* Curr.Opin.Struc.Biol., 8 1998, pp. 346–354.
8. C. Chen, A. Gentles, J. Jurka, and S. Karlin: *Genes, pseudogenes, and alu sequence organization across human chromosomes 21 and 22.* PNAS, 99 2002, pp. 2930–3935.
9. O. Clay, S. Caccio, S. Zoubak, D. Mouchiroud, and G. Bernardi: *Human coding and non coding dna: compositional correlations.* Mol. Phyl. Evol., 1 1996, pp. 2–12.
10. G. D'Onofrio, D. Mouchiroud, B. Aïssani, C. Gautier, and G. Bernardi: *Correlations between the compositional properties of human genes, codon usage, and amino acid composition of proteins.* J. Mol. Evol., 32 1991, pp. 504–510.

11. L. Duret, D. Mouchiroud, and M. Gouy: *Hovergen: a database of homologous vertebrate genes*. Nucleic Acids Research, 22(12) 1994, pp. 2360–2365.
12. L. Duret, D. Mouchiroud, and C. Guatier: *Statistical analysis of vertebrate sequences reveals that long genes are scarce in gc-rich isochores*. Journal of Molecular Evolution, 40 1995, pp. 308–317.
13. A. Eyre-Walker and L. Hurst: *The evolution of isochores*. Nat. Rev. Genet., 2(7) 2001, pp. 549–55.
14. G. Fichant and C. Gautier: *Statistical method for predicting protein coding regions in nucleic acid sequences*. CABIOS, 3 1987, pp. 287–295.
15. Y. Guédon: *Estimating hidden semi-markov chains from discrete sequences*. Journal of Computational and Graphical Statistics, 12(3) 2003, pp. 604–639.
16. L. Guéguen: *Sarment: Python modules for hmm analysis and partitioning of sequences*. Bioinformatics, 21(16) 2005, pp. 3427–8.
17. J. Hawkins: *A survey on intron and exon lengths*. Nucleic Acids Research, 16 1988, pp. 9893–9908.
18. J. Henderson, S. Salzberg, and H. Fasman: *Finding genes in dna with a hidden markov model*. Journal of Computational Biology, 4 1997, pp. 127–141.
19. International Human Genome Sequencing Consortium: *Initial sequencing and analysis of the human genome*. Nature, 409 2001, pp. 860–919.
20. K. Jabbari and G. Bernardi: *Cpg doublets, cpg islands and alu repeats in long human dna sequences from different isochore families*. Gene, 224(1-2) 1998, pp. 123–127.
21. A. Khelifi, L. Duret, and D. Mouchiroud: *Hoppsigen: a database of human and mouse processed pseudogenes*. Nucleic Acids Research, 3D 2005, pp. 59–66.
22. A. Krogh: *Two methods for improving performance of an hmm and their application for gene-finding*. In Proceedings of the Fifth International Conference on Intelligent Systems for Molecular Biology, 1997, pp. 179–186.
23. V. Lukashin and M. Borodovsky: *Gene-mark.hmm : new solutions for gene finding*. Nucleic Acids Research, 26 1998, pp. 1107–1115.
24. G. Mouchiroud, G. D'Onofrio, B. Aissani, G. Macaya, C. Gautier, and G. B. G: *The distribution of genes in the human genome*. Gene, 100 1991, pp. 181–7.
25. A. Nekrutenko and W. Li: *Assessment of compositional heterogeneity within and between eukaryotic genomes*. Genome Res., 10(12) 2000, pp. 1986–95.
26. H. Nielsen and A. Krogh: *Prediction of signal peptides and anchors by a hidden markov model*. In Proceedings of the Sixth International Conference on Intelligent Systems for Molecular Biology, 1998, pp. 122–130.
27. L. Rabiner: *A tutorial on hidden markov models and selected applications in speech recognition*. Poceeding of the IEEE, 77(2) 1989.
28. S. Rogic, A. Mackworth, and F. Ouellette: *Evaluation of gene-finding programs on mammalian sequences*. Genome Research, 11 2001, pp. 817–832.
29. J. Thiery, G. Macaya, and G. Bernardi: *An analysis of eukaryotic genomes by density gradient centrifugation*. J Mol Biol, 108(1) 1976, pp. 219–35.
30. S. Zoubak, O. Clay, and G. Bernardi: *The gene distribution of the human genome*. Gene, 174(1) 1996, pp. 95–102.

FireμSat: An Algorithm to Detect Microsatellites in DNA

Corné de Ridder[1], Derrick G. Kourie[2], and Bruce W. Watson[2]

[1] School of Computing, University of South Africa, South Africa, Pretoria 0003
[2] Fastar Research Group, Department of Computer Science, University of Pretoria, South Africa, Pretoria 0002
driddc@unisa.ac.za, dkourie@cs.up.ac.za, watson@cs.up.ac.za

Abstract. In the context of this paper microsatellites (short *approximate* tandem repeats) refer to consecutive patterns contained in genomic sequences. A new algorithm to detect such microsatellites in DNA is proposed. The algorithm relies on the construction of finite automata originating from the Moore machine paradigm. The proposed finite automata contain "counting states". The overall algorithm is designed to support user requirements as expressed by the typical geneticist.

Keywords: finite automata, microsatellites, tandem repeats, computational biology

1 Introduction

A perfect tandem repeat (PTR) is a string of nucleotides in a genomic sequence whose initial substring (of some arbitrary length) is followed by two or more copies of that substring. The introductory substring is called the motif of the PTR.

For example, `ACGACGACGACGACG` is a PTR, in which the motif is a substring of length 3 (*i.e.* $|motif| = 3$), namely `ACG`.

In contrast, an approximate tandem repeat (ATR) is a genomic sequence whose introductory substring (or motif) is followed by two or more substrings, of which at least one need not necessarily be an *exact* copy of the motif. The extent to which these non-exact copies may vary from the motif is limited, as will be discussed later in this article.

An example of an ATR is: `ACGACTACG`. In this case, the substring `ACT` that directly follows on the motif `ACG` has a mismatch in the third position.

In the absence of further qualification, reference to a TR should be construed as a reference to either a PTR or an ATR. It should be noted that there is not complete consensus on the precise meaning of a TR. In some cases, it is not required that the TR starts off with its motif. In fact, there are some who would be content to regard a string of approximate repeats as a TR, even if it did not contain the motif at all. However, this research has been driven by the requirements of geneticists and molecular biologists who were interested in detecting TR's as defined above.

A TR element (TRE) that matches the identified motif of the TR will be referred to as a PTR element (PTRE). A TRE that does not match the motif is referred to as an ATR element (ATRE).

In the literature ([21]; [15]; [16];) a distinction is made between TR's that constitute microsatellites, minisatellites and satellites. However, terminology is not used consistently in the literature.

Castelo et al. [4] coins the term *Simple Sequence Repeats (SSR's)* for microsatellites; Tran et al. [22] terms microsatellites *short tandem repeats*. Delgrange and Rivals

[6], Benson [3] and Abajian [1] consider TR's characterized by motif lengths greater than or equal to two and smaller than or equal to five ($2 \leq |motif| \leq 5$) to be microsatellites. Thurston and Field [21] consider TR's microsatellites if $2 \leq |motif| \leq 6$. For the purposes of this paper microsatellites will be consider to be TR's with a PTRE or motif such that $2 \leq |motif| \leq 5$. Microsatellites may include both PTRE *and* ATRE.

Although the algorithm, FireμSat, that is proposed here, can in theory be applied to search for TR's of any length, the focus at this stage, is to introduce an algorithm that searches specifically for microsatellites.

It will be seen that FireμSat has several parameters that can be used to tune its search. It should be emphasised that these parameters have been devised in consultation with the intended user community, who have been unable to usefully deploy existing software for TR detection. The objective is to fine tune a TR search so that redundant data is avoided, and relevant data is not missed.

The remainder of this paper is laid out as follows. Section 2 provides a brief overview of existing software packages or proposed algorithms that attempt to address the computational problem of detecting microsatellites on DNA. Section 3 defines the problem in a formal manner. In section 4 an outline is provided of how finite automaton (FA) technology can be used to detect tandem repeats in a DNA string, culminating in pseudo-code for the FireμSat algorithm. Section 5 concludes the paper, and points to work currently underway to empirically test the FireμSat algorithm.

2 Related work

There are various software packages that search directly or indirectly for microsatellites. In this regard Van den Bergh [23] mentions that although most authors reference a selection of software that has been developed before the software that they propose, there does not seem to be a comprehensive catalogue of relevant software. It is possible to classify existing software in various ways. Benson [3] divided the algorithms that he investigated into three categories and mentions their shortcomings as follows:

- *Alignment algorithms*
  Alignment algorithms proposed by (Benson [2], Kannan and Myers [7] and Schmidt [19]) have an excessive running time—their running time is exponential.
- *Algorithms from the field of data compression*
  An algorithm proposed by Milosavljevic and Jurka [14] detects simple sequences thus mixtures of fragments that occur elsewhere. Simple sequences may or may not contain TR's. This algorithm makes no attempt to deduce a repeated pattern. Rivals et al. [17] also developed an algorithm belonging to this category that is based on the presence of preselected patterns with ($1 \leq |motif| \leq 3$). This algorithm suffers from severe limitations in terms of the motif length that is allowed, and in terms of the fact that the algorithm only searches for preselected motifs.
- *Algorithms that aim to find TR's more directly.*
  Of these algorithms, the one developed by Landau et al. [11] is limited by its definition of approximate repeats. The algorithm requires that two copies differ by k or fewer substitutions (Hamming distance) or by k or fewer substitutions and indels (unit cost edit distance). The requirement for a fixed number of differences rather than a percentage is regarded as unsatisfactory. Similarly, the treatment of substitutions and indels as equals is regarded as unsatisfactory. The heuristic

algorithm proposed by Karlin et al. [8] is hampered in the same manner by the use of matching blocks separated by error blocks of fixed size. Myers and Sagot [15] has proposed an exact algorithm that requires that the approximate pattern size and a range for the number of copies should be pre-specified. An earlier algorithm of Benson [2] only finds TR's if they have a pattern size that is specified in advance.

Delgrange and Rivals [6] argue that an exact algorithm that entails the systematic detection of significant TR's in a way that is independent of the motif or of the sequence length is beyond the scope of present methods. In regard to existing software Delgrange and Rivals [6] also distinguish between three different classes of algorithms and their shortcomings as follows:

- *Fast algorithms from the field of computer science.*
  In the field of computer science there are several fast algorithms that search for only two exact tandem repeats. Authors presenting these approaches include Main and Lorentz [13]; Kolpakov and Kucherov [9] and Stoye and Gusfield [20]. Although these algorithms may be useful as filters to detect possible duplicate motifs they do not comply with the needs of molecular biologists [6].
- *Algorithms that do not make provision for the detection of TR's containing substitutions, deletions and insertions at once.*
  Algorithms in this category include those developed by Kolpakov and Kucherov [10], as well as the algorithms developed by Landau et al. [12] and Coward and Drablos [5]. These particular algorithms only make provision for substitutions.
- *Algorithms that detect TR's and allow for substitutions, insertions and deletions.*
  These algorithms include the work of Myers and Sagot [15] who introduced a combinatorially exhaustive approach that identifies several possible motifs and alignments for each TR. The complexity of this approach depends exponentially on some parameters. The work of Rivals [18] is limited to small motifs and allows only indels between two of the motifs within a TR.

3 Formal problem statement

ATR's on genetic sequences are defined in terms of the following, more formal conventions. A PTR whose motif ρ is repeated p times where $p \geq 1$, is denoted by ρ^p. An ATR u that is derived from this PTR ρ^p, must also have the motif (ρ) as its prefix. It therefore has the form $\rho u_2 \cdots u_p$ where each ATRE, $u_k (k = 2 \cdots p)$, is the result of at most ε mutations on ρ. Here ε is called the *motif error*. In theory, ε could be anywhere in the range $0 \leq \varepsilon \leq |\rho|$.

However, when running FireμSat, the user is required to choose a maximum value for ε that complies with certain practical considerations. In determining whether the string $\rho u_2 \cdots u_p$ is to be construed as an ATR, this value of ε represents the maximum number of mutations (or errors) that are tolerated, in deciding whether or not, for each $k = 2 \cdots p$, u_k represents an acceptable ATRE. In 3.1, the author discusses the types of mutations that are tolerated. Here it is emphasized that the following toleration limits on ε apply for a given ρ.

1. If $|\rho| = 2$ or $|\rho| = 3$ then only zero or one error is tolerated; i.e. ε may be chosen as either 0 or 1. (The default is 1.)
2. If $|\rho| = 4$ or $|\rho| = 5$ then zero, one or two errors are allowed; i.e. ε may be chosen as either 0, 1 or 2. (The default is 2.)

Recall that the objective is to detect microsatellites. This means that $2 \leq |\rho| \leq 5$.

Consider an example where $\rho = \texttt{ACGTT}$. Then $|\rho| = 5$ and the user may consequently select the maximum number of errors to be either 0 or 1 or 2. If the user selects "2", then ACT would be regarded as an ATRE, since it may be construed as the motif in which two deletions (see 3.1) have occurred. Likewise, ACGT could be regarded as an ATRE, since it may be seen as the motif in which one deletion has occurred. (See 3.1.) However, AC will not be regarded as an ATRE.

3.1 Type of mutations tolerated

A substring u is considered similar to the substring ρ^p if it can be written as $u = u_1 u_2 \cdots u_p$ where each word u_k ($k = 1 \cdots p$) is obtained by at most ε mutations on ρ and where ε is some pre-specified limit in the range $0 \leq r \leq |\rho|$. This was explained in the previous paragraph (3). (Note that in running FireμSat, the user has further options for constraining the search for ATR's. These options are discussed in 3.2. They are concerned with constraining the ratio of ATRE's to PTRE's in a string and/or constraining the number of consecutive ATRE's in the string.)

To further illustrate the above, consider an example based on the three letter PTRE $\rho = \texttt{ACG}$, where $\varepsilon = 1$ has been selected. This means that at most 1 mutation is allowed. The authorized forms of each ATRE u_k are, therefore, as follows:

1. The word ρ itself: $u_k = \texttt{ACG}$ and $|u_k| = 3$.
2. The word ρ with the deletion of one nitrogenous base: $u_k \in \{\texttt{CG}, \texttt{AG}, \texttt{AC}\}$. Thus, in all these cases $|u_k| = 2$.
3. The word ρ with the mismatch of one base: $u_k \in \{\texttt{XCG}|\texttt{X} : \{\texttt{C,G,T}\}\} \cup \{\texttt{AXG}|\texttt{X} : \{\texttt{A,G,T}\}\} \cup \{\texttt{ACX}|\texttt{X} : \{\texttt{A,C,T}\}\}$. In all these cases $|u_k| = 3$.
4. The word ρ with an insertion in front of or behind any position ρ_i of ρ. $u_k \in \{\texttt{ACGX}|\texttt{X} : \{\texttt{A,C,G,T}\}\} \cup \{\texttt{ACXG}|\texttt{X} : \{\texttt{A,C,G,T}\}\} \cup \{\texttt{AXCG}|\texttt{X} : \{\texttt{A,C,G,T}\}\} \cup \{\texttt{XACG}|\texttt{X} : \{\texttt{A,C,G,T}\}\}$. In all these cases $|u_k| = 4$.

It should be noted that all these words keep at least 2 bases from the original word ρ. As it stands, the foregoing could lead to ambiguity in determining the mutational origin of a string. For example, ACG could be construed as some intended PTRE, ρ, or as a deletion of the last nucleotide, G, of the PTRE ρ, followed by the insertion of G.

To resolve such ambiguities, the following rules will be applied wherever possible:

1. A string will be interpreted as a PTRE rather than as an ATRE with mutations.
2. A string will always be regarded as an ATRE that results from mismatches, rather than from insertions or deletions.
3. An ATRE will be regarded as originating from a deletion rather than from an insertion.

This manner of defining authorized forms of mismatches and deletions of u_k derives from experimental observations cited by Rivals et al. [17]. It has been endorsed by Benson

[3] as providing statistically relevant information. The algorithm proposed also allows for other types of errors that can be adjusted by the user. More details pertaining to this matter are to be found in 3.2.

Indeed, the approach was discussed with a molecular biologist, L. P. Wright, from the University of Pretoria, who was positive about the statistical relevance of the information that would be generated by the proposed algorithm.

In principle, then, an algorithm seeking TR's could rely on the motif error (ε) alone to determine when the end of a candidate string has been found. However, in practice, it is useful to rely on additional metrics. In 3.2 three such metrics are introduced. They determine whether a string that has been found to be a possible TR at some point in the algorithm, should be output as such, or whether further processing should occur to see if the string can be further extended to produce a longer TR.

3.2 Additional metrics and threshold values

In addition to considering ε (the maximum motif error that may occur within a TR), FireμSat also computes three additional metrics. These are σ, the so-called substring error; tn_atreC, the number of ATRE's that occur consecutively; and tn_tre, the total number of TRE's. In each case, the user can specify maximum values for these metrics, which FireμSat will use as a threshold value in determining when a given substring can be regarded as a TR. Each of these metrics will now be considered in turn.

1. *The substring error*:σ
   This is a measure of the extent to which the number (weighted as described below) of ATRE's in the candidate TR exceeds the number of PTRE's. The measure is computed at appropriate points by FireμSat and then compared against a user-specified threshold value of the *maximum substring error allowed*, τ. During processing $\sigma \leq \tau$ should always hold.
   In line with the guidelines suggested by Benson [3], the value of σ depends, *inter alia* on penalties (or weights) allocated by the user to mismatches (p_m), deletions (p_d) and insertions (p_i). For a given motif, ρ, and a given substring that has been partitioned into the form $u = \rho u_2 \cdots u_p$, σ on u is computed as:

   $$\sigma = (n\_d * p\_d) + (n\_i * p\_i) + (n\_m * p\_m) - n\_ptre$$

   where n_d is the number of deletions in u; n_i is the number of insertions in u; n_m is the number of mismatches in u; and n_ptre is the number of PTRE's in u. The user may rely on system default values for the penalties. These are $p\_i = 1.0$, $p\_d = 1.0$ and $p\_m = 0.5$ respectively. A penalty weight of 0 may be chosen for one or more of the mutation types, in which case no penalty is assigned to ATRE's that derive from that mutation type.
   The value of σ therefore reflects the extent to which the number of ATRE's exceeds the number of PTRE's, weighted in terms of penalty values associated with mismatches, deletions and insertions.
   The foregoing implies that FireμSat has to keep a count of the number of the various types of mutations. As will be seen in section 4.1, FireμSat makes use of an FA denoted by FA_{TR}, which is the sum (in the sense of Kleene's theorem Rule 2 of part 3) of four other FA's: one for recognizing PTRE's, and one each for recognizing insertions, deletions and mismatches. In general the substring error σ is calculated every time a final state is reached in FA_{TR}. Each such final state is associated with a unit increment in either the number of PTRE's, or the number of insertions, or the number of deletions or the number of mismatches. It is these final states, therefore, that enable the counting of the various types of mutations.

For as long as $\sigma \leq \tau$ holds, the scan of the input string continues in an effort to increase the length of the TR found to date. If the condition is not met, then the TR found to date is output, and the next TR in the input string is sought.
Of course, whenever a dead end state (a state that has only incoming edges, including a loop into the state itself labelled with all the alphabet letters of the input alphabet) of FA_{TR} is reached, then the TR is also output, and the search for the next TR resumed.

2. *The maximum number of consecutive ATRE's* (α)
The user has the option of entering a value denoted by α. This value indicates the maximum number of ATRE's that are allowed to occur next to each other. Thus α serves as a second threshold value.
If the user specifies a value for α, then the counter *tn_atreC* is maintained to record the total number of consecutive ATRE's since the last PTRE. The counter is incremented whenever an ATRE has been read (indicated by a transition to a final state of FA_{TR}) irrespective of the type of elements—whether it be an insertion, deletion or mismatch. However, when a PTRE is read, then the value of *tn_atreC* is again set to zero. The processing of a string will only proceed if $\alpha \leq$ tn_atreC.
Note that a value for α is not activated by default. Thus, if the user does not enter a value for α, then there is no limit to the number of ATRE's that may occur consecutively. (Alternatively, one might say that the default value of α is ∞.)
3. *The minimum number of tandem repeat elements (β)*
To avoid the output of unwanted data, the user may indicate the minimum number of TRE's that has to occur before a TR is output, denoted by β. To this end, a count, *tn_tre*, is kept of the total number of tandem repeat elements encountered to date in the current candidate TR. In fact, a count is also kept of the total number of PTRE's encountered to date, *tn_ptre*, and of the total number of ATRE's encountered to date, *tn_atre*. Clearly, *tn_tre* = *tn_ptre* + *tn_atre*.
The current candidate TR will only be reported as a TR if one of the previously mentioned thresholds or terminating conditions is encountered *and* if *tn_tre* $\geq \beta$. The default value for β is two.

To illustrate these concepts, consider the genetic substring `ACGACACACACGCGCGACGACT`. Let the motif be `ACG`. The values for *n_d,n_i, n_m, tn_ptre, tn_atre* and *tn_atreC* are as follows at different processing intervals of the substring.

0.	`ACGACACACACGCGCACGACT`	n_d	n_i	n_m	tn_ptre	tn_atre	tn_atreC
1.	`ACG`	0	0	0	1	0	0
2.	`ACGAC`	1	0	0	1	1	1
3.	`ACGACAC`	2	0	0	1	2	2
4.	`ACGACACAC`	3	0	0	1	3	3
5.	`ACGACACACACG`	3	0	0	2	3	0
6.	`ACGACACACACGCGC`	3	0	1	2	4	1
7.	`ACGACACACACGCGCGACG`	3	0	1	3	4	0
8.	`ACGACACACACGCGCACGACT`	3	0	2	3	5	3

Suppose that τ was specified by the user as 5, and that the default values for the penalties are used, namely $p\_i = 1.0, p\_d = 1.0, p\_m = 0.5$. Then:

$$\begin{aligned}\sigma &= (n\_d * p\_d) + (n\_i * p\_i) + (n\_m * p\_m) - n\_ptre\\ &= (3 * 1) + (0 * 1) + (2 * 0.5) - 3\\ &= 1\end{aligned}$$

and since this is less than the specified value for τ, FireμSat would attempt to explore elements beyond the given genetic substring before deciding at which stage the substring should be reported as a TR.

The algorithm is invoked by:

$$\text{Fire}\mu\text{Sat}(min,\ max, \varepsilon, \tau, \alpha, \beta, p\_d, p\_i, p\_m, gSeq)$$

where *gSeq* represents the entered genetic sequence. It returns all TR's with motif lengths in the range [*min,max*] in *gSeq*, subject to motif error ε and threshold values τ, α and β as discussed in subsections 3.1 and 3.2 respectively.

4 Algorithm Construction

4.1 FireμSat

The theory underlying FireμSat is a combination of straightforward FA technology combined with a flavour of Moore machine technology. How this theory is applied will be elaborated in the process of introducing the theoretical underpinnings of FireμSat.

For illustrative purposes, `ACG` will be used throughout as the motif string. In addition, to facilitate the explanation of the algorithm, the following FA's are introduced. In each case, the way in which the given FA scans a string of the form $u = \rho u_2 u_3 \cdots u_p$ will be described.

- $FA_P(\rho)$ is an FA that reaches a final state after scanning the first occurrence of ρ in u. In fact, it reaches the final state again if $u_2 = \rho$ is encountered in u, and again if $u_3 = \rho$ is encountered in u, etc. However, $FA_P(\rho)$ goes to a dead end state as soon as a character in u is encountered that indicates that u is not a PTR. Thus, $FA_P(\rho)$ accepts a PTR of arbitrary length, with motif ρ, entering the final states as many times as there are PTRE's in the PTR.
- $FA_D(\rho, \varepsilon)$ is an FA that, upon scanning u, reaches its first final state once the substring ρ has been read. $FA_D(\rho, \varepsilon)$ continues to reach final states after scanning each word, u_i (where $i = 2 \cdots p$) provided that one of the following conditions hold: a) either $u_i = \rho$ or b) u_i is a word deduced from ρ that contains a maximum of ε deletions.
- $FA_M(\rho, \varepsilon)$ is an FA that functions analogously to $FA_D(\rho, \varepsilon)$, except that it functions in terms of *mismatches* rather than deletions.
- $FA_I(\rho, \varepsilon)$ is an FA that functions analogously to $FA_D(\rho, \varepsilon)$, except that it functions in terms of *insertions* rather than deletions.
- $FA_{TR}(\rho, \varepsilon)$ is an FA obtained from the sum of all the previously defined FA's. Thus:
$$FA_{TR}(\rho, \varepsilon) = FA_P(\rho) + FA_D(\rho, \varepsilon) + FA_M(\rho, \varepsilon) + FA_I(\rho, \varepsilon)$$
- Finally, the predicate $isTR(gSeq[i,j], \varepsilon, \tau, \alpha, \beta, \rho)$ is defined as true if there is a TR with motif ρ in the genetic sequence $gSeq[i,j]$, such that the motif error is no greater than ε, the substring error (σ) is no greater than τ, the number of consecutive occurrences of ATRE's (tn_atreC) is no greater than α and the total number of TRE's (tn_tre) is at least β.

The *Fire Engine* software [24] constructs an FA from a regular expression (r.e.) that is provided as input. For a given motif, it is relatively easy to specify the regular expressions that correspond to the various FA's just mentioned above.

For example, the language accepted by $FA_P(\mathtt{ACG})$ can be defined by the r.e. $(\mathtt{ACG})(\mathtt{ACG})^*$. Similarly, the languages accepted by $FA_D(\mathtt{ACG}, \mathtt{1})$, $FA_M(\mathtt{ACG}, \mathtt{1})$ and $FA_I(\mathtt{ACG}, \mathtt{1})$ may be defined by means of r.e.'s, respectively, as follows:

- $FA_D(\mathtt{ACG}, \mathtt{1})$ accepts the language defined by the r.e. $(\mathtt{ACG})(\mathtt{ACG} + \mathtt{AC} + \mathtt{AG} + \mathtt{CG})^*$.
- $FA_M(\mathtt{ACG}, \mathtt{1})$ accepts the language defined by the r.e. $(\mathtt{ACG})(\mathtt{ACG} + \mathtt{CCG} + \mathtt{GCG} + \mathtt{TCG} + \mathtt{AAG} + \mathtt{AGG} + \mathtt{ATG} + \mathtt{ACA} + \mathtt{ACC} + \mathtt{ACT})^*$.
- $FA_I(\mathtt{ACG}, \mathtt{1})$ accepts the language defined by the r.e. $(\mathtt{ACG})(\mathtt{ACG} + \mathtt{AACG} + \mathtt{CACG} + \mathtt{GACG} + \mathtt{TACG} + \mathtt{ACCG} + \mathtt{AGCG} + \mathtt{ATCG} + \mathtt{ACAG} + \mathtt{ACGG} + \mathtt{ACTG} + \mathtt{ACGA} + \mathtt{ACGC} + \mathtt{ACGT})^*$.

If these deterministic FA's ($FA_D(\mathtt{ACG}, \mathtt{1})$, $FA_M(\mathtt{ACG}, \mathtt{1})$) are constructed and distinction is made between the type of final states a trace through each respective FA will confirm that strings of the form $\rho u_1 \cdots u_q$ are recognized, where:

- $\rho u_1 \cdots u_q$ may be preceded by p, some arbitrary non-motif prefix.
- ρ is the motif (in the present example, ACG) of the TR,
- each $u_i, i = 1 \cdots q$ is an ATRE based on ρ, allowing for an error of maximally ε. Note that in the present example, $\rho = \mathtt{ACG}$ and thus $|\rho| = 3$. Therefore, as previously discussed, ε is only allowed to assume the value of 1 or 0.
- $q \geq 1$ is the number of ATRE's that follow on from the motif in the TR.

If $FA_D(\mathtt{ACG}, \mathtt{1})$, $FA_M(\mathtt{ACG}, \mathtt{1})$ and $FA_I(\mathtt{ACG}, \mathtt{1})$ are constructed as deterministic FA's then it will be seen that if any additional element that does not belong to the TR identified up to that point is encountered in the input string, the FA will transit to a dead-end state in each respective case.

It is possible to distinguish between two kinds of final states in each of these machines: those which signal that a motif (PTRE) has been scanned, and those which indicate that a deletion or mismatch or insertion has been scanned. These states will be referred to as PTR- and ATR-final states, respectively. As explained below, the number of transitions into these states have to be counted, and the respective values of σ, *tn_atreC* and *tn_tre* have to be correspondingly updated so as to ensure that the strings designated as TR's are consistent with thresholds τ, α and β respectively, as explained in section 3.2 above.

In order to contribute to the foregoing explanation figure 1 has been included. A trace through figure 1 will confirm that strings of the form $p\rho u_1 \cdots u_q$ are recognized, where:

- p is some arbitrary non-motif prefix preceding a TR,
- ρ is the motif (in the present example, ACG) of the TR,
- each $u_i, i = 1 \cdots q$ is a TRE based on ρ, allowing for an error of maximally ε. Note that in the present example, $\rho = \mathtt{ACG}$ and thus $|\rho| = 3$. Therefore, as previously discussed, ε is only allowed to assume the value of 1. Thus only 1 deletion may occur.
- $q \geq 1$ is the number of ATRE's that follow on from the motif in the TR.

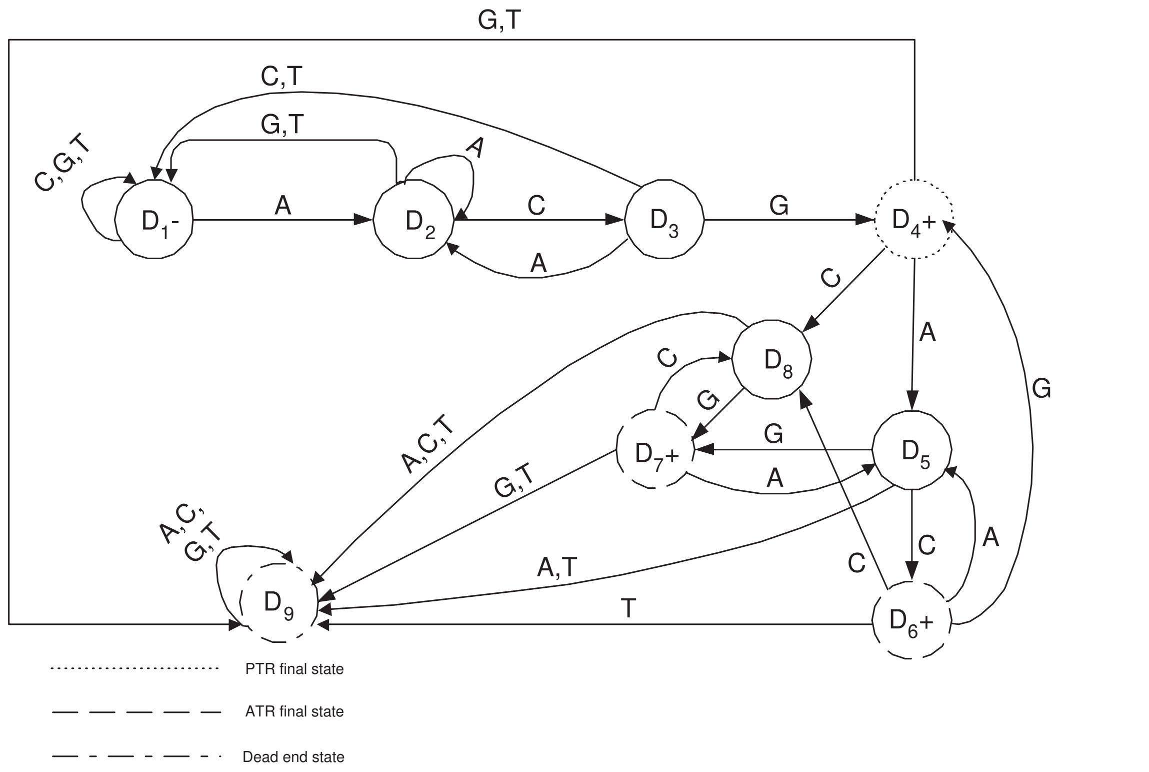


Figure 1. $FA_D(\texttt{ACG},1)$

It will also be seen that if any additional element that does not belong to the TR identified up to that point is encountered in the input string, then the FA transits to a dead-end state. State D_9 in figure 1 is a dead-end state without any outgoing edges.

In figure 1, there are also two kinds of final states: those which signal that a motif (PTRE) has been scanned, and those which indicate that a deletion has been scanned. These states will be referred to as PTR- and ATR-final states, respectively. As explained below, the number of transitions into these states have to be counted, and the respective values of σ, tn_atreC and tn_tre have to be correspondingly updated so as to ensure that the strings designated as TR's are consistent with thresholds τ, α and β respectively, as explained in section 3.2 above.

In order to construct $FA_{TR}(\rho, 1)$ we first construct the respective constituent machines and then apply the constructive algorithm which forms part of the proof of *Rule 2, Part 3* of Kleene's theorem.
$FA_{TR}(\texttt{ACG}, \texttt{1}) = FA_P(\texttt{ACG}) + FA_D(\texttt{ACG}, \texttt{1}) + FA_M(\texttt{ACG}, \texttt{1}) + FA_I(\texttt{ACG}, \texttt{1})$.
This can be done by using the *Fire Engine* software toolkit [24] that provides a function for adding FA's.

The discussion to date can be generalized: $FA_{TR}(\texttt{XYZ}, \texttt{1})$ is an FA for recognizing the parameterized motif XYZ of length 3. The parameters are X, Y and Z and each of these parameters can be instantiated to any one of the nucleotides $\{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}\}$. This parameterized FA is, as before, the sum of four other FA's, each of which are also parameterized.

Thus, the r.e. associated with $FA_P(\texttt{XYZ})$ can be defined as $(\texttt{XYZ})(\texttt{XYZ})^*$. Furthermore, $FA_D(\texttt{XYZ}, \texttt{1})$, $FA_M(\texttt{XYZ}, \texttt{1})$ and $FA_I(\texttt{XYZ}, \texttt{1})$ can also be defined as follows:

- $FA_D(\mathtt{XYZ}, 1)$ accepts the language defined by the r.e.
  $(\mathtt{XYZ})(\mathtt{XYZ} + \mathtt{XY} + \mathtt{XZ} + \mathtt{YZ})^*$.
- $FA_M(\mathtt{XYZ}, 1)$ accepts the language defined by the r.e.
  $(\mathtt{XYZ})(\mathtt{XYZ} + \mathtt{YYZ} + \mathtt{ZYZ} + \mathtt{RYZ} + \mathtt{XXZ} + \mathtt{XZZ} + \mathtt{XRZ} + \mathtt{XYX} + \mathtt{XYY} + \mathtt{XYR})^*$.
- $FA_I(\mathtt{XYZ}, 1)$ accepts the language defined by the r.e.
  $(\mathtt{XYZ})(\mathtt{XYZ} + \mathtt{XXYZ} + \mathtt{YXYZ} + \mathtt{ZXYZ} + \mathtt{RXYZ} + \mathtt{XYYZ} + \mathtt{XZYZ} + \mathtt{XRYZ} + \mathtt{XYXZ} + \mathtt{XYZZ} + \mathtt{XYRZ} + \mathtt{XYZX} + \mathtt{XYZY} + \mathtt{XYZR})^*$.

Thus, in principle, any FA_{TR} of motif length 3 can be algorithmically constructed. Similarly, parameterized versions for $FA_{TR}(\rho, \varepsilon)$ can be constructed for $|\rho| = 2, 4, 5$ and for permissable values of ε. In each case, the r.e.'s relating to the constituent FA's have to be determined, the corresponding FA's are then derived using the Fire Engine toolkit, and these derived FA's are then summed, also using the toolkit, to provide $FA_{TR}(\rho, \varepsilon)$.

Once $FA_{TR}(\rho, \varepsilon)$ is constructed, certain adaptations to the conventional FA language recognition algorithm are required when scanning through a genomic sequence in search of the next TR. Some of the details relating to these adaptations will be discussed later.

For the present, consider the high-level description of FireμSat given henceforth. As mentioned previously, the algorithm requires as parameters:

- The lower and upper bound of motif lengths to be considered (*min* and *max* respectively);
- the maximum allowable motif error (ε - discussed in section 3.1);
- the maximum allowable substring error (τ - discussed in section 3.2);
- the penalty values used to calculate the substring error (*p_m*, *p_d* and *p_i* all explained in section 3.2);
- the maximum allowable number of ATRE's that may occur consecutively (α - discussed in section 3.2);
- the minimum number of TRE's that should occur before a string is output as a TR (β - explained in section 3.2) and;
- the genomic sequence itself (*gSeq*). (The development of the actual software is in progress the user is provided with an additional option to select default values for the respective threshold values.)

The following functions are assumed:

- $GenerateWords(\rho Length)$ generates a set of all words of length $\rho Length$ from the alphabet $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$.
- $CreateFA_{TR}(\rho, \varepsilon)$ returns $FA_{TR}(\rho, epsilon)$ as discussed.
- $FindIndices(gSeq, FA_{TR}, \tau, \alpha, \beta, p\_m, p\_d, p\_i)$ returns a set of index pairs in *gSeq*. A substring of *gSeq* is a TR recognized by FA_{TR} within the constraints specified by τ, α and β as explained in section 3.2 if and only if its start and endpoint indices constitute a pair in the returned set. Note that the call to this function is independent of all prior and subsequent calls to it.

proc $Fire\mu Sat(min, max, \varepsilon, \tau, \alpha, \beta, p\_m, p\_d, p\_i, gSeq)$
 pre $\{(0 < min \leq max) \wedge (0 \leq \varepsilon) \wedge (\sigma \leq \tau) \ \wedge \ (0 \leq \alpha \leq tn\_atreC) \wedge$
  $(0 \leq \beta \leq tn\_tre) \wedge (gSeq \in \Sigma^*)\}$
 $indices := \phi$
 ; **for** $\rho Length : [min, max] \rightarrow$
  $; words := GenerateWords(\rho Length)$
  $; FASet := \phi$
  ; **for** $w : words \rightarrow$
   $; FA_{TR} := CreateFA_{TR}(w, \varepsilon)$
   $; indices := indices \cup FindIndices(gSeq, FA_{TR}, \tau, \alpha, \beta, p\_m, p\_d, p\_i)$
  rof
 rof
 post $\{(i, j) \in indices \Leftrightarrow \exists \rho : \Sigma^* \cdot |\rho| \in [min, max] \wedge$
  $isTR(gSeq[i, j], \varepsilon, \tau, \alpha, \beta, p\_m, p\_d, p\_i, \rho)\}$

Note that in order to use FA_{TR} appropriately in FireμSat, it is required that the final states of the original component FA's be identifiable in it. Note that of the features of the constructive algorithm introduced in the proof of *Rule 2, Part 3* of Kleene's algorithm is that if it is used to compute say $FA_X = FA_Y + FA_Z$, then every final state in FA_Y can be mapped to a final state in FA_X. The same holds true for every final state in FA_Z. Moreover, every final state in FA_X will either map to a final state in FA_Y or to a final state in FA_Z or to a final state in both FA_Y and FA_Z.

To determine whether the conditions on the threshold values, τ (representing the maximum allowable substring error), α (representing the maximum number of ATRE's that may occur consecutively) and β (the minimum allowable number of TRE's that have to occur before a TR is reported) have been met when scanning through a tandem repeat, various counters, initially at 0, have to be updated once a motif is encountered as we scan through a string. To this end let the variables tn_ptre and tn_atre store the number of PTR-final states and ATR-final states encountered to date, respectively. Additionally, the variables n_d, n_m and n_i store the number of deletions, mismatches and insertions encountered to date, respectively.

For reasons explained later, the number of symbols scanned since the last tally of a final state is stored in ℓ, and a flag *motif* is set to true that final state marked the end of a PTRE and to false, otherwise. Finally ε, is the maximum number of symbols by which an ATRE may differ from the motif. (The value of ε is easily determined and depends on the motif length as explained in section 3.)

The logic of how these counters are to be updated whenever a state, Q, of an FA_{TR} is being examined. A number of predicates are assumed that test whether Q is a final state ($isFinal(Q)$) and/or whether Q is a state that terminates a motif ($isPTRE(Q)$), and/or a deletion ($isDel(Q)$), insertion ($isIns(Q)$) or mismatch ($isMis(Q)$).

Note, specifically, that more than one of these conditions may hold for a final state, as forthcoming discussed. Dijkstra's guarded command language (GCL) is used, in which the semantics of the if-statement specifies that non-deterministic selection of the guards takes place if more than one guard evaluates to true. Therefore, to avoid ambiguity, guards have to be designed to be mutually exclusive. In each case, the body then adjusts the counters according to the rules already given above.

Thus, it will be seen that if a final state is of multiple types, then the PTRE counter (tn_ptre) takes precedence, followed by the mismatch counter (n_m), followed by the deletions counter (n_d), followed by the insertion counter (n_i). By this it is meant that if a final state is encountered that is final for both PTRE's and mismatches, then the PTRE counter is incremented rather than the mismatch counter. Similarly, mismatches are incremented rather than deletions, etc.

However, there are a few exceptions to be dealt with in the case of an insertions final state being reached. Firstly, the insertions counter n_i is only incremented if the next state, R, is *not* also an insertion state. Secondly, suppose that the last TRE encountered was a PTRE (indicated by the flag *motif*) and that the number of transitions from this last PTRE state to this current insertion state (recorded in ℓ) is less than or equal to ε. It is then assumed that an insertion has been encountered instead of a motif ℓ transitions earlier. Consequently the *tn_ptre* counter that was previously incremented is now decremented.

Note that similar logic ought to be built in, to check that when arriving at a PTR state, a deletion was not incorrectly recorded less than ε transitions earlier. If so, the n_d, tn_atre and tn_atreC ought to be decremented. The outline below leaves out this logic in the interests of overall simplicity. However, it is built into the implemented algorithm.

Note in passing that the semantics of GCL dictates that, if a condition arises that does not fire a guard in an if-statement, then the if-statement should abort, indicating that such a condition constitutes an error. Thus, for example, in the code below, there is no guard to deal with a condition where a state is designated as final, but it is not associated with a PTRE, nor with a mismatch, nor with a deletion, nor with an insertion. Such a condition ought not to arise, and would indeed constitute an error if it did.

$$
\begin{array}{l}
R := nextState(Q, nextSymbol) \\
\textbf{if}\ \ isFinal(Q) \rightarrow \\
\quad \textbf{if}\ \ isPTRE(Q) \rightarrow \\
\quad\quad tn\_ptre, tn\_atreC := tn\_ptre + 1, 0 \\
\quad\quad ; \ell, motif := 0, true \\
\quad [\!] \ \ (\neg isPTRE(Q) \wedge isMis(Q)) \rightarrow \\
\quad\quad tn\_atre, tn\_atreC, n\_m := tn\_atre + 1, tn\_areC + 1, n\_m + 1 \\
\quad\quad ; \ell, motif := 0, false \\
\quad [\!] \ \ (\neg isPTRE(Q) \wedge \neg isMis(Q) \wedge isDel(Q)) \rightarrow \\
\quad\quad tn\_atre, tn\_atreC, n\_d, := tn\_atre + 1, tn\_atreC + 1, n\_d + 1 \\
\quad\quad ; \ell, motif := 0, false \\
\quad [\!] \ \ (\neg isPTRE(Q) \wedge \neg isMis(Q) \wedge \neg isDel(Q)) \rightarrow \\
\quad\quad \textbf{if}\ \ (\ isIns(Q) \wedge \neg isIns(R)) \rightarrow \\
\quad\quad\quad \textbf{if}\ \ (\ell \leq \varepsilon \wedge motif) \rightarrow tn\_ptre := tn\_ptre - 1\ \ \textbf{fi} \\
\quad\quad\quad ; tn\_atre, tn\_atreC, n\_i := tn\_atre + 1, tn\_atreC + 1, n\_i + 1 \\
\quad\quad\quad ; \ell, motif := 0, false \\
\quad\quad [\!] \ \ (isIns(Q) \wedge isIns(R)) \rightarrow \\
\quad\quad\quad \ell := \ell + 1 \\
\quad\quad \textbf{fi} \\
\quad \textbf{fi} \\
[\!] \ \ \neg isFinal(Q) \rightarrow \ell := \ell + 1 \\
\textbf{fi}
\end{array}
$$

5 Conclusion

The above code indicates that counter and threshold values are only adjusted whenever a final state is reached in FA_{TR}. This gives FireμSat a certain Moore machine character - Moore machines print output only in relation to the state that is reached—irrespective of which arc was followed to get to that state. However, in this case, when particular states are reached corresponding counters are adjusted, instead of printing specific characters as would be required in a Moore machine.

If a dead-end state is reached or $\tau < \sigma$ or $\alpha < tn\_atreC$, then the processing on the applicable FA_{TR} will terminate, and the TR scanned up to that point will be output, provided that $\beta \geq tn\_tre$. Scanning will then continue in search of the next TR.

The various parameters have already been discussed. They were derived in close collaboration with molecular biologists with a view to enhancing the useability of the algorithm. For example the software under construction allows the user to allocate penalties in a very sensitive manner, which enables the user to predetermine relatively easily at least what type of repeats will definitely be detected. It should be noted that these useability features are a direct consequence of using FA technology.

The implementation of FireμSat, is in progress. Preliminary runtime results show that FireμSat copes satisfactorily in searching for microsatellites with at most one mutation. Runtime results of motifs with length four or five where two mutations are allowed compares very well with STAR ([6]) but not that well with Tandem Repeats Finder ([3]). However, it should be emphasised that FireμSat provides the user with a degree of flexibility and usability does not appear to be available in the other algorithms. Future initiatives are directed at the completion of the FireμSat software and at reporting comparative results in more detail.

References

1. C. Abajian: *Sputnik*. Online: http://espressosoftware.com/pages/sputnik.jsp.
2. G. Benson: *A space efficient algorithm for finding the best non-overlapping alignment score*. Theoretical Computer Science, 145 1995.
3. G. Benson: *Tandem repeats finder*. Nucleic acids research, 27(2) November 1999, pp. 573 – 580.
4. A. T. Castelo, W. Martins, and G. R. Gao: *Troll: Tandem repeat occurrence locator*. Bioinformatics Applications Note, 18(4) 2002, pp. 634–636.
5. E. Coward and F. Drablos: *Detecting periodic patterns in biological sequences*. Bioinformatics, 14 1998.
6. O. Delgrange and E. Rivals: *Star: an algorithm to search for tandem approximate repeats*. Bioinformatics, 20(16) June 2004, pp. 2812–2820.
7. S. Kannan and E. Myers: *An algorithm for locating nonoverlapping regions of maximum alignment score*. SIAM Journal on Computing, 25(3) 1996, pp. 648–662.
8. S. Karlin, M. Morris, G. Ghandour, and M. Leung: *Efficient algorithms for molecular sequence analysis*. In: Proceedings of the National Academy of Sciences of the United States of America, 85 1988.
9. R. Kolpakov and G. Kucherov: *Finding maximal repetitions in a word in linear time*. In: 40th FOCS. IEEE Computer Society Press, 1999.
10. R. Kolpakov and G. Kucherov: *Finding approximate repetitions under hamming distance*. In: ESA: Annual European Symposium on Algorithms, Lecture Notes in Computer Science, 2161 2001.

11. G. Landau and J. Schmidt: *An algorithm for approximate tandem repeats*. In: Proceedings of the 4th Combinatorial Pattern Matching Conference, Lecture Notes in Computer Science 648, 1993.
12. G. Landau, J. Schmidt, and D. Sokol: *An algorithm for approximate tandem repeats*. Journal of Computational Biology, 8(1) 2001, pp. 1–18.
13. M. Main and R. Lorentz: *An $O(n \log n)$ algorithm for finding all repetitions in a string*. Journal of Algorithms, 5 1984.
14. A. Milosavljevic and J. Jurka: *Discovering simple dna sequences by the algorithmic significance method*. Computer Applications in Biosciences, 9(4) 1993, pp. 407–411.
15. G. Myers and M.-F. Sagot: *Identifying satellites and periodic repetitions in biological sequences*. Journal of Computational Biology, 5(3) 1998, pp. 539–554.
16. E. Rivals: *Eric rivals's homepage*. Online: http://www.lirmm.fr/˜rivals/tete-en.html.
17. E. Rivals, J. Delahaye, O. Delgrange, and M. Dauchet: *A first step toward chromosome analysis by compression algorithms*. In: Proceedings of the First International IEEE Symposium on Intelligence in Neural and Biological Systems (INBS '95), 1995.
18. E. Rivals, O. Delgrange, J.-P. Delahaye, M. Dauchet, M.-O. Delorme, A. Henaut, and E. Ollivier: *Detection of significant patterns by compression algorithms: the case of approximate tandem repeats in dna sequences*. CABIOS, 13 1997.
19. J. Schmidt: *All highest scoring paths in weighted grid graphs and its application to finding all approximate repeats in strings*. SIAM Journal on Computing, 27 1998.
20. J. Stoye and D. Gusfield: *Simple and flexible detection of contiguous repeats using a suffix tree*. Theoretical Computer Science, 27 2002.
21. M. Thurston and D. Field: *Msatfinder: detection and characterisation of microsatellites*. Online: http://www.genomics.ceh.ac.uk/˜milo/msatfinder/, 2005.
22. N. Tran, B. Bharaj, E. Diamandis, M. Smith, B. Li, and H. Yu: *Short tandem repeat polymorphism and cancer risk: influence of laboratory analysis of epidemiologic findings*. Cancer Epidemiology Biomarkers and Prevention, 13 2004.
23. I. Van den Bergh: *Finding microsatellites in whole genomes*, Master's thesis, Technische Universiteit Eindhoven, 2006.
24. B. W. Watson: *The design and implementation of the FIRE Engine: A C++ toolkit for finite automata and regular expressions*. Online: http://alexandra.tue.nl/extra1/wskrap/public html/9411065.pdf.

Using Alignment for Multilingual Text Compression

Ehud S. Conley and Shmuel T. Klein

Department of Computer Science
Bar Ilan University
52 900 Ramat-Gan
Israel
Tel: (972–3) 531 8865
Fax: (972–3) 736 0498
{konli,tomi}@cs.biu.ac.il

Abstract. Multilingual text compression exploits the existence of the same text in several languages to compress the second and subsequent copies by reference to the first. We explore the details of this framework and present experimental results for parallel English and French texts.

Keywords: Compression, multilingual texts, text alignment, coding

1 Introduction

In countries like Canada, Belgium and Switzerland, where speakers of two or more languages live side-by-side, all official texts have to be published in multilingual form. The current legislation of the ever expanding European Union obliges the translation of all official texts into the languages of all member states. As a result, there is a growing corpus of important texts, large parts of which are highly redundant, since they do not have any information content of their own, and are just transformed copies of some other parts of the text collection.

We wish to exploit this redundancy to improve compression efficiency in such situations, and introduce the notion of *Multilingual Text Compression*: one is given two or more texts, which are supposed to be translations of each other and are referred to as *parallel* texts. One of the texts will be stored on its own (or compressed by means of pointers referencing only the text itself), the other texts can be compressed by referring to the translation, using appropriate dictionaries.

Data compression in general, and text compression in particular, have for long been prominent topics in the Information Retrieval literature, as full text IR systems are voracious consumers of storage space, both for the underlying textual database itself, but also for the auxiliary overhead, such as indices, dictionaries, thesauri, etc., see, for example, [14,20,13]. This work concentrates on multilingual information retrieval systems and how their data could be compressed.

In a certain sense, multilingual text compression is an extension of *delta-coding*, in which source and target files S and T are given, with the assumption that T is very similar to S, for example in the case of several versions of the same software package. Highly efficient compression schemes have been designed for that case, and the compressibility is obviously a function of the similarity of the input files. Our problem extends the delta-coding paradigm to the case where similarity is not based on the appearance of identical strings, but allow the use of some transformation to pass from a given text fragment to its matching part.

The basis for enabling multilingual text compression is first the ability to match the corresponding parts of related texts by identifying semantic correspondences across the various sub-texts, a task generally referred to as *alignment*. As the methods for detailed alignment are quite sensitive to noise, they usually use a rough alignment of the text as an auxiliary input. They might also use an existing multilingual glossary, but they always generate their own probabilistic glossary, which corresponds to the processed text.

The current work extends the use of alignment to the question of whether and how the property of parallelism can be exploited to store those texts in a more space-efficient way. In other words, we wish to find a way to compress the constituent parallel sub-texts so that the result will demand less space than would be required if they were compressed without exploiting their parallelism.

In the next section, we review some related work. Section 3 then brings the suggested algorithm and in Section 4 we report on preliminary experimental results. The last section suggests future work.

2 Related work

Multilingual texts have been considered in the Information Retrieval literature, where the challenge is to access information in one language while the query might be given in another, see, e.g. [5]. Alignment of parallel texts has been used mainly for machine translation, machine-aided translation and bilingual term extraction [17]. Most algorithms for alignment are designed for bilingual texts only [9], but some work has been done already for three languages as well [16]. However, the state of the art for detailed alignment, even for two languages, is still far from perfect. It is thus not surprising that works on more than two languages do not exist, but a reasonable mapping for (A, B, C) can be synthesized given alignment outputs for (A, B) and (A, C).

Most current detailed alignment techniques are based on one of the following models: (a) IBM's Model 2 [3], from which the *word_align* algorithm [7] has been derived; and (b) Hiemstra's model [12], used both by Xerox' system [10] and the *linköping Word Aligner* [1].

All these methods use some monolingual tools such as part-of-speech taggers, lemmatizers and possibly parsers for phrase detection. Determining the lemma (= base form) of each word is critical for the success of the alignment process, especially when performed across languages from different groups [6]. When the lemmatized versions of the texts are processed instead of the original versions, the words within the induced bilingual glossary will naturally be all lemmata rather than morphological variants.

The compression of similar texts has been considered in the vast research area dealing with delta coding, see [4,2]. The popular ZLIB tool is optimized to take advantage of the similarity across the files, and some of its features are used also in our algorithm. The compression of parallel texts is treated in [15], but without using text alignment tools.

3 Compression of a text using its translation

3.1 Compression modeling

The following compression algorithm tries to take advantage of the fact that the text being compressed is divisible into two parallel parts which are translations of each other. Dictionary based compression algorithms use pointers to occurrences of the same substrings either along the text, as in LZ77 [21] or within an auxiliary dictionary, as in LZW [19]. The current algorithm, however, uses pointers to the translations of the substring appearing in the parallel section of the text. The original substrings may be easily retrieved through these pointers using a bilingual glossary along with some other linguistic resources.

Pointing to another occurrence of a given substring within the same text sometimes requires a relatively large number of bits. That is because the closest occurrence of that substring can happen far back in history, which is why most implementations limit the size of the window in which a previous occurrence is to be searched for. In contrast, translations of words or phrases within a parallel text, if such exist, must appear in the corresponding translation unit, namely a sentence or paragraph. Moreover, if no large omissions or insertions occur, the translation is expected to be found within a very narrow text window, whose middle position is computable using the given alignment. The encoding pointers can store the offset of the translation from that alignment; these offsets are always very small and thus may be encoded using only a few bits.

It is important to emphasize that the quality of the alignment does not have any effect on the correctness of the compression algorithm. That is because the missing words or word sequences are restored according to the same glossary by which the alignment has been determined. It is expected that the compression rate would not be affected either, as alignment algorithms make mistakes due to the consistent appearance of the wrong translations in the corresponding text windows, even in more probable positions. This means that the same sequence can be compressed at least the same number of times using the erroneous translation and perhaps even at a better cost.

The suggested algorithm assumes the following resources:

1. S, T: The source- and target-language texts, respectively, where T is a translation of S.
2. $A_{S,T}$: A word- and phrase-level alignment of the text pair (S, T). Let $s_{i,l}$ denote the word sequence of length l within S beginning at the ith word. Similarly, let $t_{j,m}$ denote the word sequence of length m within T beginning at the jth word. $A_{S,T}$ consists of a set of connections of the form $\langle i, l, j, m\rangle$, each of which indicating the fact that $s_{i,l}$ and $t_{j,m}$ have been determined as matching phrases. We assume that for any pair (j, m) there is at most one connection of the form $\langle i, l, j, m\rangle$ within $A_{S,T}$. From here and below, s_i and t_j stand for $s_{i,1}$ (the ith word of S) and $t_{j,1}$ (the jth word of T), correspondingly.
3. S^{lem}, T^{lem}: Lemmatized forms of S and T. Let $s^{lem}_{i,l}$ and $t^{lem}_{j,m}$ denote the lemma sequences corresponding to $s_{i,l}$ and $t_{j,m}$, respectively. That is the concatenations of the lemmata of s_i, s_{i+1}, $\ldots$, s_{i+l-1} and t_j, t_{j+1}, $\ldots$, t_{j+l-1}, correspondingly.
4. L_S: A lemmata dictionary. The entries of this dictionary are the words appearing in S. Each entry stores a list of all possible lemmata of the keyword, sorted in descending order of frequency. Let $L_S(s)$ denote the lemma list for the word s. For instance, if S is an English text, then $L_S(\texttt{working}) = (\texttt{work}, \texttt{working})$.

```
COMPRESS_TARGET

j ⟵ 1
while j ≤ |T| do
        found ⟵ false
        for m ⟵ m_max downto 1 do
                if ∃i,l such that ⟨i,l,j,m⟩ ∈ A_{S,T}     //⟨i,l,j,m⟩ is unique
                        diff ⟵ i − al(j)
                        if diff ≥ 0                then   sign ⟵ 0
                                                   else   sign ⟵ 1
                        offset ⟵ B(|diff|)
                        length ⟵ B(l − 1)
                        for n ⟵ 0 to l − 1 do
                                lemma_n ⟵ I(s^{lem}_{i+n}, L_S(s_{i+n}))
                        trans ⟵ I(t^{lem}_{j,m}, G_{S,T}(s^{lem}_{i,l}))
                        for n ⟵ 0 to m − 1 do
                                variant_n ⟵ I(t_{j+n}, V_T(t^{lem}_{j+n}))
                        pointer ⟵ concatenation ( 1, offset, sign, length,
                                          lemma_0, ..., lemma_{l−1}, trans,
                                          variant_0, ..., variant_{m−1})
                        output pointer
                        j ⟵ j + m
                        found ⟵ true
                        break
                endif
        end for
        if not found
                output concatenation ( 0, code(t_j))
                j ⟵ j + 1
        endif
end while
```

Figure 1. Compression using a translated file

5. V_T: A variant dictionary. The entries of this dictionary are the lemmata of all words appearing in T. Each entry stores a list of all possible morphological variants of the key lemma, sorted in descending order of frequency. Let $V_T(t)$ denote the variant list for the lemma t. For example, if T is a French text, then V_T(`normal`) = (`normal`, `normale`, `normaux`, `normales`).
6. $G_{S,T}$: A bilingual glossary corresponding to the text pair (S, T). The entries of this glossary are source language lemma sequences. Each entry includes a list of possible translations of the key sequence into target language sequences, sorted in descending order of probability. The translations also appear in lemmatized form. Let $G_{S,T}(s)$ denote the translation list of the source language sequence s into the target language. For instance, if S and T are English and French texts, correspondingly, then $G_{S,T}$(`mineral water`) = (`eau mineral`). Note that the word `eau` (water) in French is feminine, which requires a feminine-form adjective, namely `minerale`, whereas the adjective `mineral` is the masculine singular form, which is the corresponding lemma.

Let $al(j)$ denote the expected position within S of the term corresponding to t_j in T, that is,

$$al(j) = \left\lfloor \frac{|S|}{|T|} j + \frac{1}{2} \right\rfloor .$$

In other words, $s_{al(j)}$ is the source word parallel to t_j if taking into account only the proportion between the lengths of S and T. The accurate alignment may then be expressed by the signed offset from $s_{al(j)}$. If a paragraph- or sentence-level alignment is available, then S and T can be referred to as the current parallel units, and the indices i and j are then relative to the beginnings of these units.

Token number	S (English)	T (French)	Encoding
1	`Subject`	`Objet`	1(0,ϵ,0,ϵ,6,0)
2	`:`	`:`	0(c(:))
3	`Supplies`	`Livraisons`	0(c(`livraison`),2)
4	`of`	`de`	1(2,0,0,ϵ,1,0,0)
5	`military`	`matériel`	
6	`equipment`	`militaire`	1(0,ϵ,0,ϵ,0,1)
7	`to`	`à`	0(c(`à`),0)
8	`Iraq`	`l’`	0(c(`le`),2)
9		`Irak`	1(0,ϵ,0,ϵ,0,ϵ)

Figure 2. Example of compression of French text using its English parallel

The algorithm works as follows: beginning at the first position $j = 1$ within T, use $A_{S,T}$ to find the longest sequence $t_{j,m}$ having a corresponding sequence $s_{i,l}$ in S. If found, create a pointer to $s_{i,l}$ by concatenating some binary encodings of the following details:

1. $i - al(j)$: Offset of s_i from $al(j)$, including sign bit.
2. $l - 1$: Length of the source sequence minus 1. As l is always greater than 0, $l - 1$ can be encoded.
3. Indices of $s_i^{lem} \dots s_{i+l-1}^{lem}$ within $L_S(s_i) \dots L_S(s_{i+l-1})$, respectively. If a single lemma exists, then the empty string ϵ is used as index (no need for encoding).
4. Index of $t_{j,m}^{lem}$ within $G_{S,T}(s_{i,l}^{lem})$. As above, in the case of a single translation, ϵ will be used.
5. Indices of $t_j \dots t_{j+m-1}$ within $V_T(t_j^{lem}) \dots V_T(t_{j+m-1})$, correspondingly. Again, ϵ is used in the case of singletons.

The pointer is then output with a 1-bit prefix. The next iteration will work for $j = j + m$.

If no m is found such that $\langle i, l, j, m \rangle \in A_{S,T}$, an alternative encoding of t_j is written to the output stream preceded by a 0-bit, and j is incremented by 1. The process continues while $j \leq |T|$. We shall use some UD (Uniquely Decypherable) code, e.g., a Huffman code, for all unaligned words in T. This code may be initially generated for all words in T and then be improved when the counts of unaligned words are known. Alternatively, the final code can be generated in advance following a preliminary parsing stage.

As to the encoding of the pointer consisting of a sequence of generally very small numbers, many of which are zeros, a simple solution would be to use an Elias γ-code for each component. A more compact encoding can be achieved by devising a Huffman code for the possible numbers, see the section on coding below.

Figure 1 displays the formal pseudo-code. $B(x)$ denotes the variable length binary encoding of x and $I(x, y)$ denotes the variable length binary encoding of the index of x within the dictionary entry y; if y contains only one item, $I(x, y) = \epsilon$.

The decompression algorithm is straightforward. Note that it needs only the dictionary files, as all relevant information included in the other files is encoded within the compressed text itself.

Figure 2 gives an example of the algorithm's output. The second and third columns contain the English and French parallel texts, respectively. The fourth column is a decimal representation of the binary encoding. The 0 to the left of parentheses denotes the encoding of unaligned words, while a 1 indicates a pointer. Numeric values within the parentheses are actually written to the binary output as variable length binary numbers, for example, if a γ-code is used, the 6-tuple $(2, 0, 0, \epsilon, 1, 0, 0)$ would be encoded as 1100|0|0||10|0|0 (10 bits).

As an example, we explain in detail the decoding of the fourth encoded token, which is $(2, 0, 0, \epsilon, 1, 0, 0)$, assuming that the first three items have already been decoded to `Objet : Livraison`. The current position (in terms of tokens) in the file T is therefore 4, and in S, it is $\lfloor (8/9) \times 4 + \frac{1}{2} \rfloor = 4$, corresponding to the word `of`. The first two numbers of the 6-tuple are retrieved: 2, 0 are translated to +2, indicating the fact that the translation sequence is located two words to the right of the current position in S, which brings us to the term `equipment`. Adding 1 to the next value, 0, tells the decoder that it should relate to a 1-word English sequence beginning (and ending) at the word `equipment`. Taking a look at the entry `equipment` in the English lemmata dictionary (L_{en}(`equipment`)) reveals there is only one lemma for that word (the lemma `equipment`). Therefore, no bits are needed in order to lemmatize it.

Now the decoder looks up the entry `equipment` within the bilingual glossary ($G_{en,fr}$(`equipment`)) and finds the list `le équipement, de matériel, équipement`. Since several French translations exist, it reads the next value, 1, and retrieves the corresponding translation (the second option), namely `de matériel`, so the translation sequence is of length 2. Since both words in this sequence have more than one variant, another two values are fetched in order to determine the exact form of each. The variants list of the lemma `de` starts with `de`, `des`, `d'`, `du`... and that of `matériel` starts with `matériel`, `matériaux`, `matériels`, `matérielles`, `matérielle`.... The two last zeros in the sequence to be decoded indicate that the first variant of each list should be taken, yielding finally the terms (not the lemmata) `de matériel` as translation for `equipment`.

Note that this translation, if considered on its own and not within the larger context of a bilingual corpus, is in fact quite wrong, since `de matériel` is a genitive form rather corresponding to `of equipment`. This is an example for the fact that an erroneous translation can still be useful in our case, if the error appears consistently.

3.2 Choosing the encoding

To understand the rational of the encoding decisions, consider Figure 3, listing the first few output lines of the above algorithm applied to our test data.

The first column is a flag indicating whether the element is a pointer or one of the non-aligned words, If it is a word, it may be followed by a number, giving the index of the requested variant in the list of alternatives for this lemma. If it is a pointer, it starts with a number k, representing an offset, in number of words, between some term

1	0 0 0 0
0	:
0	`organigramme` 1
0	`de` 0
1	11 0 0 1 1 0
1	5 0 0 3 5 0
1	1 0 0 3 0
0	`elle` 0
0	:
1	9 0 0 0
0	)
1	3 0 0 2 0
1	5 0 0 5 4 0 2
0	`agent` 0
1	7 0 0 1
1	10 0 0 0 3 0
0	`dans` 0

Figure 3. Output of translation algorithm

positions as explained above. If k is not zero, it is followed by a sign bit, encoded here by 0 or 1. The rest of the numbers in the pointers are indices within sets of variants.

The encoding tries to take advantage of the fact that the distribution of the elements in the different fields is not the same. In fact, three Huffman codes are used:

1. H_1 — for the different words in the lines labeled 0;
2. H_2 — for the offsets (first numbers in lines labeled 1);
3. H_3 — for all the indices appearing in both types of lines, words and pointers.

The first tree H_1 is quite large, giving a codeword for each of the different non-aligned words. As to H_2, most of the offsets are small, and their distribution is skewed, with a clear bias to the smaller numbers. The numbers encoded by H_3 are usually even smaller, since for most sets, there are generally very few variants. Moreover, since these variants are ordered by decreasing frequency, the first few integers, especially 0, will appear with high probability. The reason for not using the same Huffman tree for the last two classes, in spite of the fact that similar elements are encoded, is that their distributions are different enough to justify two trees, in particular because no ambiguity arises: there is only one element of H_2 for each pointer line, so no special indicator is needed for the fact that the next codeword is from H_3.

There is no need to encode the sign field by some Huffman code. Once we know that a pointer is encoded, the first codeword belongs to H_2, and if it is decoded as representing a number different from 0, we know that it is followed by a sign bit, so the Huffman codeword is just followed by the sign bit itself. On the other hand, the flag bit indicating if the current line is a word or a pointer, needs to be encoded. Instead of wasting one bit for each line, it turned out, on our tests, to be advantageous to adopt the following scheme: every new line is by default assumed to represent a word, and the Huffman tree H_1 is extended to accommodate also an "Escape" word, which will be used at the beginning of every pointer line.

The encoding of H_3 can further be improved by noticing that the probability of the number 0 will be higher than $\frac{1}{2}$, suggesting, as in [8], to build a Huffman code for a set of items consisting of (a) individual numbers appearing in the sequences and (b) of runs of zeros of different lengths. The elements to be encoded by H_3 are therefore $0, 1, 2, \ldots, Z_2, Z_3, \ldots$, where Z_i stands for a run of i zeros.

As example, the first 5 lines of Figure 3 would be encoded by the sequence: H_1(ESC), $H_2(0)$, $H_3(Z_3)$, H_1(:), H_1(`organigramme`), $H_3(1)$, H_1(`de`), $H_3(0)$, H_1(ESC), $H_2(11)$, 0, $H_3(0)$, $H_3(1)$, $H_3(1)$, $H_3(0)$.

3.3 Results

The bilingual text used for evaluating the new algorithm comprises the English and French versions of the European Union's JOC corpus, a collection of pairs of questions and answers on various topics. These texts, used on the ARCADE evaluation project [18] were supplied aligned at the question/answer (paragraph) level. As the translation is rather precise, correct word- and phrase-level alignments reside quite close to the linear alignment of each paragraph pair. The automatic word- and phrase-level alignment as well as the bilingual glossary were obtained using an extended version of the *word_align* algorithm [7].

The English raw text has about 1,050,000 words, whereas the respective French text consists of about 1,162,000 words. Table 1 brings the sizes of the compressed French file (as a fraction of the original) for various compression schemes: Gzip, based on LZ77, Bzip, based on the Burrows-Wheeler transform, Hword, a Huffman code encoding each of the different words in the text as single items, and finally Trans, the algorithm suggested in this work, based on the translation from the English parallel.

The numbers do not include the sizes of the auxiliary files for Trans and Hword, since in the scenario of a large multilingual information retrieval system, dictionaries and glossaries are needed anyway and are not stored exclusively as an aid for compression. However, even if those sizes are to be considered, it should be kept in mind that, according to Heaps' Law [11], the size of a dictionary for a text of size n is expected to be αn^{β}, where $0.4 \leq \beta \leq 0.6$. The total size of the auxiliary dictionaries for the current evaluation corpus, compressed using Bzip (rather than a dictionary-oriented compression scheme), is about 9% of the French raw text. Should a 1GB corpus be compressed, then corresponding dictionaries would comprise less than 0.9% of the original text. Obviously, specific dictionary compression can further decrease that rate.

Full size	Gzip	Bzip	Hword	Trans
7551550	0.307	0.214	0.225	0.212

Table 1. Comparison of compression efficiency

As can be seen, Trans is better than Gzip, Hword and Bzip, even without attempting to optimize the code further. Additional savings can be achieved by using an improved alignment module, transforming a larger part of the file into pointers rather than words, or by improving the encoding schemes. Consider, for example, again the table in Figure 3. At first sight, having a variant number associated with words like `agent` seems reasonable, as the word could also appear in plural form `agents`, but getting such a number for a preposition like `dans` might be surprising. A closer look however reveals that almost every word appears in at least two forms: all lower case and capitalized (except, obviously, special words like punctuation signs). This suggests the following strategy (not yet implemented).

Only one form of every word will be kept, using capitalization for proper nouns and lower case for the other words. If a word appears at the beginning of a sentence (follows a period or similar mark), it will be assumed to be capitalized. Exceptions, which should be rare, are handled by adding a codeword for an OVERRIDE, which will be encoded as part of the Huffman tree H_1 and will have the interpretation of (a) being followed by another codeword w from H_1; and (b) changing the case of the first letter of the word represented by w. The OVERRIDE will be used in case of lower case proper nouns (like in email addresses) or capitalized other words in the middle of a sentence. The effect of such a change will be to reduce the number of variants, so that smaller numbers will be encoded, and in some cases, if the number of variants is reduced to 1, no encoding at all is needed.

Another optimization could be to compare, for each item, the number of bits required to encode it with reference to its translation with the number of bits needed for the corresponding word using a word based Huffman code, that is, it might sometimes pay to consider a term that could be aligned as if it were unaligned. The resulting hybrid algorithm improves on both the original form of TRANS and on HWORD.

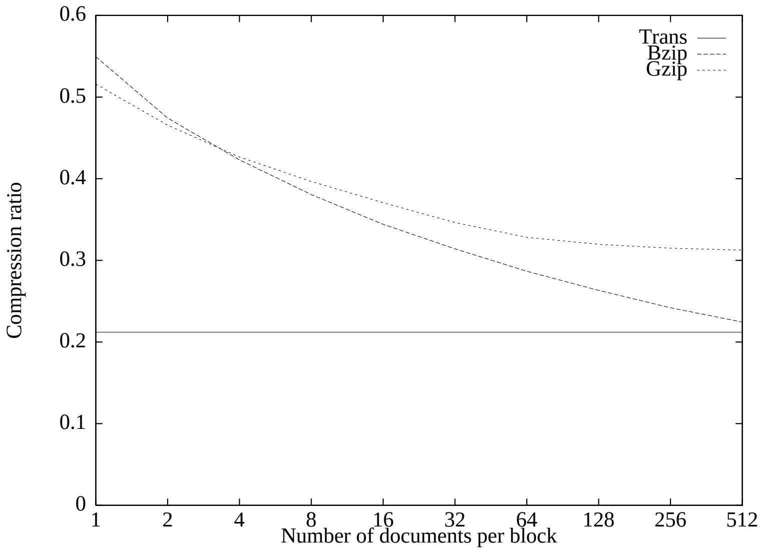


Figure 4. Compression performance as function of basic block size

GZIP and BZIP are adaptive methods and not really competitors for the applications treated here. The full decoding of the entire corpus is rarely needed, and small sub-parts, such as a single question/answer document, should be accessible individually. This is, however, not the case for adaptive methods, which require a sequential scan from the beginning of the file, while methods like TRANS and static Huffman coding support selective access and decoding. One can, of course, encode smaller parts of the file individually also by GZIP and BZIP, but compression will deteriorate. Figure 4 shows the relative size of the compressed French file for the various methods, as a function of the size of a basic block, which is supposed to be encoded independently from the others. This size is expressed by the number of consecutive question/answer documents in each block. For example, if each document is compressed on its own, compression by GZIP and BZIP reduces the full file only to 0.516 and 0.549, respectively, while TRANS stays at 0.212. With increasing block size, compression by the adaptive methods improves, but approaches the performance of Trans only for very large blocks of more than 500 documents.

4 Conclusion and Future Work

The existence of the same text in several languages can be used to improve the compression of a multilingual system. We have presented preliminary tests for two languages, achieving a good performance. By fine tuning the encoding, the compression results may be improved.

We intend to test our method on much larger parallel corpora of various languages, in order to obtain more reliable and generic results. We plan to explore also the possibility of bidirectional bilingual compression, where pointers can refer both from S to T and vice versa, which could lead to improvements, since phrases may have different lengths in different languages. A further topic to be treated is pattern matching directly within the compressed bilingual text, allowing the treatment of simple queries in a multilingual Information Retrieval environment.

Acknowledgement: This work has been supported in part by Grant 25915 of the Israeli Ministry of Industry and Commerce (Magnet Consortium Kite). The first author also wishes to express his gratitude for the support by a grant from Globes, the Israeli business daily.

References

1. Ahrenberg, L., Andersson, M., and Merkel, M.: *A knowledge-lite approach to word alignment*, in Parallel Text Processing, J. Véronis, ed., Kluwer Academic Publishers, Dordrecht, 2000, pp. 97–116.
2. Ajtai, M., Burns, R. C., Fagin, R., and Long, D. D. E.: *Compactly encoding unstructured inputs with differential compression*. Journal of the ACM, 49(3) 2002, pp. 318–367.
3. Brown, P. F., Della Pietra, S., Della Pietra, V. J., and Mercer, R. L.: *The mathematics of statistical machine translation: parameter estimation*. Computational Linguistics, 19(2) 1993, pp. 263–311.
4. Burns, R. C. and Long, D. D. E.: *Efficient distributed backup and restore with delta compression*, in Workshop on I/O in Parallel and Distributed Systems (IOPADS), ACM, 1997.
5. Carbonell, J. G., Yang, Y., Frederking, R. E., Brown, R. D., Geng, Y., and Lee, D.: *Translingual information retrieval: A comparative evaluation*, in Proc. IJCAI, 1997, pp. 708–715.
6. Choueka, Y., Conley, E. S., and Dagan, I.: *A comprehensive bilingual word alignment system: Application to disparate languages: Hebrew and english*, in Parallel Text Processing, J. Véronis, ed., Kluwer Academic Publishers, Dordrecht, 2000, pp. 69–96.
7. Dagan, I., Church, K. W., and Gale, W. A.: *Robust bilingual word alignment for machine-aided translation*, in Proc. of the Workshop on Very Large Corpora: Academic and Industrial Perspectives, Columbus, Ohio, 1993, pp. 1–8.
8. Fraenkel, A. S. and Klein, S. T.: *Novel compression of sparse bit-strings*, in Combinatorial Algorithms on Words, Apostolico, A. and Galil, Z., eds., vol. F12 of NATO ASI Series, Springer Verlag, Berlin, 1985, pp. 169–183.
9. Gale, W. A. and Church, K. W.: *A program for aligning sentences in bilingual corpora*. Computational Linguistics, 19(3) 1993, pp. 75–102.
10. Gaussier, É., Hull, D., and Aït-Mokhtar, S.: *Term alignment in use: Machine-aided human translation*, in Parallel Text Processing, J. Véronis, ed., Kluwer Academic Publishers, Dordrecht, 2000, pp. 253–274.
11. J. Heaps: *Information Retrieval: Computational and Theoretical Aspects*, Academic Press, Inc., New York, NY, 1978.
12. D. Hiemstra: *Using statistical methods to create a bilingual dictionary*, Master's thesis, Universiteit Twente, 1996.

13. S. T. Klein: *Techniques and applications of data compression in information retrieval systems*, in Database and Data Communication Network Systems, C. Leondes, ed., vol. 2, Elsevier Science, San Diego, CA, 2002, ch. 16, pp. 573–633.
14. Moffat, A. and Zobel, J.: *Adding compression to a full-text retrieval system.* Software — Practice & Experience, 25(8) 1995, pp. 891–903.
15. Nevill, C. and Bell, T.: *Compression of parallel texts.* Information Processing & Management, 28 1992, pp. 781–793.
16. M. Simard: *Translation-text alignment: Three languages are better than two*, in Proc. of the Joint SIGDAT Conference on Empirical Methods in Natural Language Processing and Very Large Corpora, June 1999, pp. 2–11.
17. J. Véronis, ed., *Parallel Text Processing*, Kluwer Academic Publishers, Dordrecht, 2000.
18. Véronis, J. and Langlais, P.: *Evaluation of parallel text alignment systems: The arcade project*, in Parallel Text Processing, J. Véronis, ed., Kluwer Academic Publishers, Dordrecht, 2000, pp. 369–388.
19. T. A. Welch: *A technique for high-performance data compression.* IEEE Computer, 17 June 1984, pp. 8–19.
20. Witten, I. H., Moffat, A., and Bell, T. C.: *Managing Gigabytes: Compressing and Indexing Documents and Images*, Van Nostrand Reinhold, New York, 1994.
21. Ziv, J. and Lempel, A.: *A universal algorithm for sequential data compression.* IEEE Trans. on Inf. Th., IT–23 1977, pp. 337–343.

Modeling Delta Encoding of Compressed Files

Shmuel T. Klein[1], Tamar C. Serebro[1], and Dana Shapira[2]

[1] Department of Computer Science
Bar Ilan University
Ramat Gan, Israel
tomi@cs.biu.ac.il, t_lender@hotmail.com

[2] Department of Computer Science
Ashkelon Academic College
Ashkelon, Israel
shapird@ash-college.ac.il

Abstract. The Compressed Delta Encoding paradigm is introduced, i.e., delta encoding directly in two given compressed files without decompressing. Here we explore the case where the two given files are compressed using LZW, and devise the theoretical framework for modeling delta encoding of compressed files. In practice, although working on the compressed versions in processing time proportional to the compressed files, our target file is much smaller than the corresponding LZW form.

Keywords: differencing encoding, delta file, LZW

1 Introduction

Delta compression is a main field in data compression research. In this paper we introduce a new model of differencing encoding, that of *Compressed Differencing*. In this model we are given two files, at least one in its compressed form. The goal is to create a third file which is the differencing file (i.e. the delta file) of the two **original** files, in time proportional to the size of the input, that is, without decompressing the compressed files.

More formally, let S be the source file and T be the target file (probably two versions of the same file). The goal is to create a new file $\Delta(S,T)$ which is the differencing file of S and T. If both S and T are given in their compressed form, we call it the *Full Compressed Differencing Problem*. If one of the files is given in its compressed form we call it the *Semi Compressed Differencing Problem*. If none of the files are compressed, it refers to the original problem of differencing.

One motivation for this problem is when the encoder is interested in transmitting the compressed target file when both encoder and decoder have the source file in its compressed or uncompressed form. Creating the Delta file can reduce the transmitted file's size and therefore the number of I/O operations. Working on the compressed given form, the encoder can save memory space as well as processing time. Another motivation is detecting resemblance among a set of files when they are all given in their compressed form without decompressing them, perhaps saving time and space. When the size of the difference file is much less than the target file, it indicates resemblance.

Traditional differencing algorithms compress data by finding common strings between two versions of a file and replacing substrings by a copy reference. The resulting file is often called a *delta* file. Two known approaches to differencing are the Longest

Common Sub-sequence (LCS) method and the edit-distance method. LCS algorithms find the longest common subsequence between two strings, and do not necessarily detect the minimum set of changes. Edit distance algorithms find the shortest sequence of edits (e.g., insert, delete, or change character) to convert one string to another. One application which uses the LCS approach is the UNIX diff utility, which lists changes between two files in a line by line summary, where each insertion and deletion involves only complete lines. Line oriented algorithms, however, perform poorly on files which are not necessarily partitioned into lines, such as images and object files.

Tichy [16] uses edit-distance techniques for differencing and considers the string to string correction problem with block moves, where the problem is to find the minimal covering set of T with respect to S such that every symbol of T that also appears in S is included in exactly one block move. Weiner [17] uses a suffix tree for a linear time and space left-to-right copy/insert algorithm, that repeatedly outputs a copy command of the longest copy from S, or an insert command when no copy can be found. This left-to-right greedy approach is optimal (e.g., Burns and Long [5], Storer and Szymanski [15]). Hunt, Vo, and Tichy [11] compute a delta file by using the reference file as part of the dictionary to LZ-compress the target file. Their results indicate that delta compression algorithms based on LZ techniques significantly outperform LCS based algorithms in terms of compression performance. Factor, Sheinwald, and Yassour [7] employ Lempel-Ziv based compression to compress S with respect to a collection of shared files that resemble S; resemblance is indicated by files being of same type and/or produced by the same vendor, etc. At first the extended dictionary includes all shared data. They achieve better compression by reducing the set of all shared files to only the relevant subset. Based on these researches we construct the delta file using edit distance techniques including insert and copy commands. We also reference the already compressed part of the target file for better compression.

Ajtai, Burns, Fagin, and Long [2] and Burns and Long [5] present several differential compression algorithms for when memory available is much smaller than S and T, and present an algorithm named checkpointing that employs hashing to fit what they call *footprints* of substrings of S into memory; matching substrings are found by looking only at their footprints and extending the original substrings forwards and backwards (to reduce memory, they may use only a subset of the footprints). Heckel [10] presents a linear time algorithm for detecting block moves using Longest Common Subsequences techniques. One of his motivations was the comparison of two versions of a source program or other file in order to display the differences. Agarwal et al. [1] speed up differential compression with hashing techniques and additional data structures such as suffix arrays. In our work we use tries in order to detect matches.

Burns and Long [6] achieve in-place reconstruction of standard delta files by eliminating write before read conflicts, where the encoder has specified a copy from a file region where new file data has already been written. Shapira and Storer [14] also study in-place differential file compression. The non in-place version of this problem is known to be NP-Hard, and they present a constant factor approximation algorithm for this problem, which is based on a simple sliding window data compressor. Motivated by the constant bound approximation factor, they modify the algorithm so that it is suitable for in-place decoding and present the In-Place Sliding Window Algorithm (IPSW). The advantage of the IPSW approach is simplicity and speed, achieved in-place without additional memory, with compression that compares well with existing methods (both in-place and not in-place). Our delta file construction algorithm is not necessarily in place, but minor changes (such as limiting the offset's size) can result in an in place version.

If both files, S and T, are compressed using Huffman coding (or any other static method), generating the differencing file can be done in the traditional way (perhaps a sliding window) directly on the compressed files. The delta encoding is at least as efficient as the delta encoding generated on the original files S and T. Common substrings of S and T are still common substrings of the compressed versions of S and T. However the reverse is not necessarily true, since the common substrings can exceed the codeword boundaries. For example, consider the alphabet $\Sigma = \{a, b, c\}$ and the corresponding Huffman code $\{$`00, 01, 1`$\}$. Let $S =$`abab` and $T =$`cbaa`, then $\mathcal{E}(S) =$`00010001` and $\mathcal{E}(T) =$`1010000`. A common substring of S and T is `ba` which refers to the substring `0100` in the compressed file. However, this substring can be extended in the compressed form to include also the following bit, as the LCS is 01000 in this case.

The problem is less trivial when using adaptive compression methods such as Lempel-Ziv compressions. The encoding of a substring is determined by the data, and depends on its location. For this reason the same substring is not necessarily encoded in the same way throughout the text. Our goal is to identify reoccurring substrings in the compressed form so that we can replace them by pointers to previous occurrences. In this paper we explore the compressed differencing problem on LZW compressed files and devise a model for constructing delta encodings on compressed files. In Section 2 we perform Semi Compressed Differencing using compressed pattern matching, where the source file is encoded using the corresponding compression method. In Section 3 we present an optimal algorithm in terms of processing time for the Semi and Full versions of the compressed differencing problem using tries.

2 Delta encoding in compressed files using compressed pattern matching

Many papers have been written in the area of compressed pattern matching, i.e. performing pattern matching on the compressed form of the file. Amir, Benson and Farach [4] propose a pattern matching algorithm in LZW compressed files which runs in $O(n + m^2)$ processing time or $O(n \log m + m)$, where n is the size of the compressed text and m is the length of the pattern. Kida et al. [12], present an algorithm for finding all occurrences of multiple patterns in LZW compressed texts. Their algorithm simulates the Aho-Chorasick pattern matching machine, and runs in $O(n + m^2 + r)$ processing time, where r is the number of pattern occurrences.

Compressed pattern matching was also studied in [9,8,13] and in many others. In this section we use any compressed pattern matching algorithm to perform compressed differencing using the same compression method. Given a file T and a compressed file $\mathcal{E}(S)$, our goal is to present T as a sequence of pointers and individual characters. The pointers point to substrings that either occur in S, or previously occurred in T. This can be done by processing T from left to right and repeatedly finding the longest match between the incoming text and either the text of S, or the text to the left of the current position in T itself; the matching string is then replaced by a pointer, or, if a match of two or more characters cannot be found, a single character of the incoming text is output.

The input of any given compressed pattern matching algorithm is a specific pattern P we are interested in searching for. Since we are interested in locating the longest possible match at the current position, we only know the position the pattern starts with but not the one it ends with. A naive algorithm can, therefore, try to locate all substrings of T starting at the current position by concatenating the following character to the pattern each time a match in S or in the already scanned part of T can still be found. A formal algorithm is given in Figure 1.

We use u to denote the length of the uncompressed file. Let $cpm(P, \mathcal{E}(S))$ denote any compressed pattern matching algorithm for matching a given pattern P in the compressed file $\mathcal{E}(S)$. It returns the position of the (last) occurrence of P in the original text S, and 0 if such location is not found. Similarly, $pm(P, T)$ denotes any (standard, non-compressed) pattern matching algorithm for matching a given pattern P in T, which returns the position of the (last) occurrence of P in T, and 0 if no such location is found. If the match between the pattern starting at the current position and the compressed form of S or the previous scanned part of T can not be extended, we output the match we have already found. If this longest match is only of a single character we output the character at the current position, and advance the position in T by one.

```
1        i ⟵ 1
2        while i ≤ u do
         {
2.1          P ⟵ T_i
2.2          j ⟵ 0
2.3          while i + j ≤ u and
             (pos1 ⟵ cpm(P, E(S)) ≠ 0 or pos2 ⟵ pm(P, T_1 ··· T_{i−1}) ≠ 0)
2.3.1            j ⟵ j + 1
2.3.2            P ⟵ P · T_{i+j} // concatenate the next character
2.4          if j = 0 // no match was found
2.4.1            output T_i
2.5          elseif pos1 ≠ 0 //output a pointer to S
2.5.1            output a pointer (pos1, j)
2.6          else // output a pointer to T
2.6.1            output a pointer (pos2, j)
2.7          i ← i + j + 1
         }
```

Figure 1. Solving the Semi Compressed Differencing problem using Compressed Pattern Matching

For analyzing the processing time of the naive algorithm presented in Figure 1, let us assume that the compressed pattern matching algorithm is optimal in terms of processing time. By the definition of Amir and Benson [3] of optimal compressed matching algorithms the running time is $O(n + m)$, where n is the size of the compressed text and m is the length of the pattern. Thus the total running time of the algorithm presented in Figure 1 is $O(u(n+m))$, even for optimal compressed pattern matching algorithms. Our goal is, therefore, to reduce the processing time. In the following chapter we concentrate on differencing in LZW files.

3 Delta Encoding in LZW Files

The LZW algorithm by Welch [18] is a common compression technique which is used for instance by the `compress` command of UNIX. The LZW algorithm parses the text into phrases and replaces them by pointers to a dictionary trie. The nodes of the trie are labeled by characters of the alphabet, and the string associated with each node is the concatenation of the characters on the path going from the root down to that node. The trie initially consists only of all the individual characters of the alphabet. At each stage, the algorithm looks for the longest match between the string starting at the current position and the previously scanned text. Since the trie includes the individual characters, there is always a possible match. Once the longest match has been found, the dictionary trie is updated to include a node corresponding to the string obtained by concatenating the matched characters with the following character in the text.

3.1 Semi Compressed Delta Encoding

```
1       construct the trie of ℰ(S)
2       i ⟵ 1
3       while i ≤ u
        {
3.1         P ⟵ T_i T_{i+1} ... T_u
3.2         Starting at the root, traverse the trie using P
3.3         When a leaf v is reached
3.3.1           ℓ ⟵ depth of v in trie (= length of matching prefix)
3.3.2           output the position in S corresponding to v
3.4         i ⟵ i + ℓ
        }
```

Figure 2. Semi Compressed Differencing algorithm for LZW compressed files

During LZW decompression a trie identical to that of the LZW compression is constructed. Although the LZW decompression algorithm takes linear time in the size of the original file, if the reconstructed file is not needed, the construction of the dictionary trie can be done in time proportional to the size of the *compressed* file [4,18].

Figure 2 presents an improved algorithm for constructing the delta file of the compressed file $\mathcal{E}(S)$ and a given file T. The algorithm constructs the dictionary trie corresponding to $\mathcal{E}(S)$. Starting from the root, the trie is traversed according to the characters read from T until a leaf is reached. The position of the string corresponding to the leaf is then output; this information has been kept in the nodes of the trie during construction. The prefix of T corresponding to the matched string is truncated, and traversing the trie continues with the remaining part of T starting again from the root. Since the trie is initialized with nodes corresponding to all the characters of the alphabet, outputting the positions of these nodes to the delta file corresponds to inserting individual characters.

The processing time of this algorithm is $O(|\mathcal{E}(S)|+|T|)$, which is linear in the size of the input. In order to improve the compression performance, we can add pointers to the portion of T that has already been processed. This can be done by constructing the trie for T in addition to that of S, as shown in the algorithm for full compressed delta files of the next sub-section.

3.2 Full Compressed Delta Encoding

In this section we present a linear time algorithm for the Full Compressed Delta Encoding problem. Figure 3 presents the algorithm for constructing the delta file of S and T given $\mathcal{E}(S)$ and $\mathcal{E}(T)$. It constructs the dictionary trie of S recording in the node corresponding to the string x, a triplet (pos, len, k), where pos is the starting position of the last occurrence of x, len is the length of x (the depth of the node in the trie), and k is the first character of x. In addition, each node of the trie records the corresponding index in the LZW list, indicating the order in which the nodes were created; this index is also the codeword corresponding to that node in the LZW encoding.

Let $Dictionary[cw]$ be a function returning a pointer to the node in the trie corresponding to codeword cw. Since the original file T is not needed, we construct the trie of T in parallel to traversing the trie of S. New nodes are introduced during this phase of the algorithm when substrings of T correspond to nodes that are not present in the trie of S. When adding or updating a node (recording the position in T and changing the corresponding codeword), the algorithm outputs the following to the delta file, depending on whether the node did exist before or not. If it is an existing node, the variables pos and len stored in it are output to the delta file; if the node is a newly created one, then the variables pos and len stored in the parent of this node are output. The ordered pair (pos, len) can refer to a substring of either S or T, depending on the last update of the output node.

The variable $flag$ indicates whether the character k, the first character of the string corresponding to the current codeword, was already written to the delta file in the preceding stage. If so, we should eliminate k from the string corresponding to the current pointer by changing the (pos, len) pair to be emitted to $(pos+1, len-1)$, that is, advancing the start position of the string to be copied by 1, while shortening the length of this string by 1. The processing time of this algorithm is $O(|\mathcal{E}(S)|+|\mathcal{E}(T)|)$, which is again linear in the size of the input.

To improve the compression performance of the delta file, we can check whether each ordered pair of the form $(position, length)$ can be combined with its previous ordered pair, i.e., if two consecutive ordered pairs are of the form (i, ℓ_1) and $(i+\ell_1, \ell_2)$ where i denotes a position in S or T and ℓ_1 and ℓ_2 denote lengths, we combine them into a single ordered pair $(i, \ell_1+\ell_2)$. The combined ordered pair can then be combined with successive ordered pairs.

Consider the following example: $S =$ `abccbaaabccba`, $T =$ `ccbbabccbabccbba`. Applying LZW we get that $\mathcal{E}(S) =$ `1233219571` and $\mathcal{E}(T) =$ `33221247957`. The dictionary trie of $\mathcal{E}(S)$ and the combined dictionary of $\mathcal{E}(S)$ and $\mathcal{E}(T)$ are given in the following figures. In the combined trie, dotted nodes indicate new nodes that were introduced during the parsing of $\mathcal{E}(T)$. Bold numbers represent data that was updated during the parsing of $\mathcal{E}(T)$, and therefore corresponds to positions in T.

We get that $\Delta(S,T) = \langle 3,2\rangle\langle 5,1\rangle\langle 5,2\rangle\langle 2,1\rangle\langle 3,1\rangle(2,1)(4,2)\langle 9,3\rangle(3,1)(4,2)$, where pointers to S are delimited with brackets, and pointers to T with parentheses. Two

```
1       construct the trie of E(S)
2       flag ⟵ 0 // output character k
3       counter ⟵ 1 // position in T
4       input oldcw from E(T)
5       while oldcw ≠ NULL // still processing E(T)
        {
5.1         input cw from E(T)
5.2         node ⟵ Dictionary[oldcw]
5.3         if (Dictionary[cw] ≠ NULL)
5.3.1           k ⟵ first character of string corresponding to Dictionary[cw]
5.4         else
5.4.1           k ⟵ first character of string corresponding to node
5.5         if ((node has a child k) and (cw ≠ NULL) )
5.5.1           output (pos + flag, len − flag) corresponding to child k of node
5.5.2           flag ⟵ 1
5.6         else
5.6.1           output (pos + flag, len − flag) corresponding to node
5.6.2           create a new child of node corresponding to k
5.6.3           flag ⟵ 0
5.7         pos of child k of node ⟵ counter
5.8         oldcw ⟵ cw
5.9         counter ⟵ counter + len − flag
        }
```

Figure 3. Full Compressed Differencing algorithm for LZW compressed files

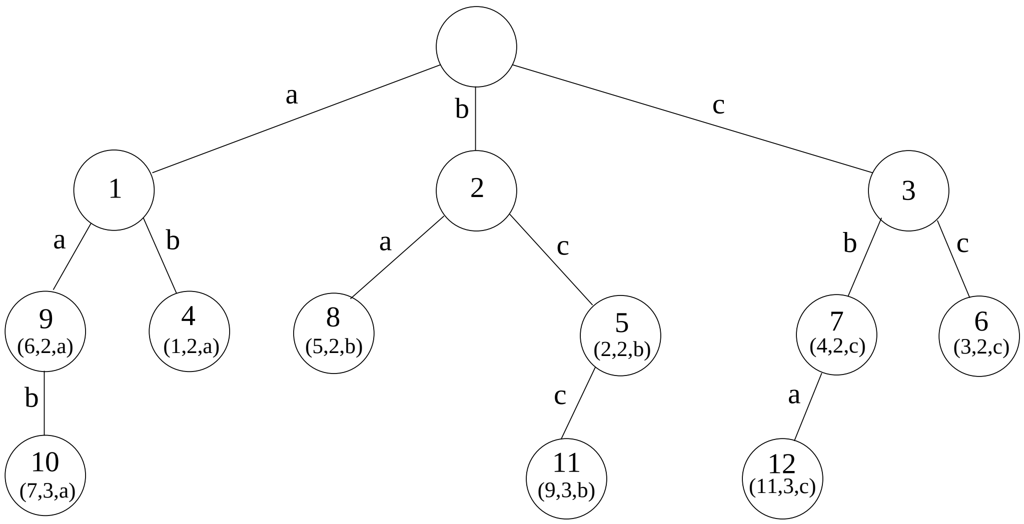


Figure 4. The Dictionary Trie for $\mathcal{E}(S)$

ordered pairs can be combined thus $\Delta(S,T) = \langle 3,3\rangle\langle 5,2\rangle\langle 2,2\rangle\mathsf{c}(4,2)\langle 9,3\rangle b(4,2)$. In this stage ordered pairs of length 1 are translated to the corresponding character.

The drawback of the above algorithm is that each codeword of the compressed file T corresponds to an ordered pair (or a single character) in the delta file. Thus relative to LZW, the only savings in compression is achieved by combining ordered pairs, leaving the performance still similar to that of LZW. In fact, the limitation of

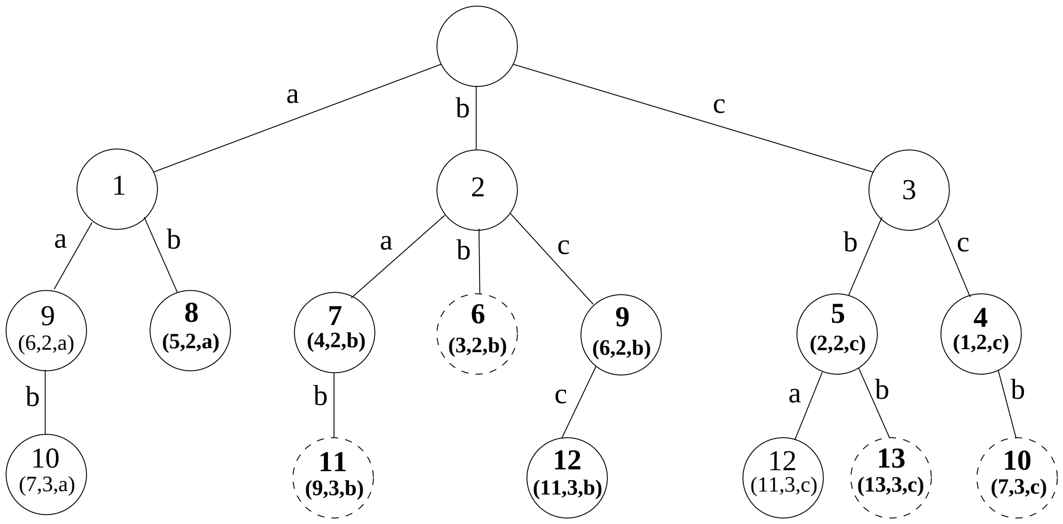


Figure 5. The Combined Dictionary Trie for $\mathcal{E}(S)$ and $\mathcal{E}(T)$

adhering to a strict use of the LZW compressed form of T in full compressed delta encoding, is that it cannot take advantage of the similarity between S and T, which might void the very basis of the applicability of differential compression.

The algorithm presented in Figure 3 constructs the trie of T after the construction of the trie of S has been completed, recording at each node of the trie information about the **last** occurrence of the corresponding string. However, it might be the case that at the beginning of the compressed file of T we are interested in information about earlier occurrences of that string in S, which have already been overwritten with later information. To improve that, we allowed various degrees of partial decoding. That is, the construction of the tries of S and T is done in parallel.

Partial decoding of degree d means alternatingly treating d codewords of S and then d codewords of T. Using this approach, we increase the probability, for synchronized files with long matches, that a codeword of the compressed file of T will refer to a node in the combined trie of S and T that corresponds to a substring of a long match. We then try to extend this match forwards while performing partial decoding. In addition to the triplet (pos, len, k) and the pointer into the original file that are stored in each node, we also add a pointer to the location in the *compressed* file of the corresponding codeword. When a match is detected, that is, when a node is reached in the combined trie that refers to a substring of S and T, we retrieve its corresponding location in the compressed file S and decompress both S and T from that point forwards, as long as the match can be extended. For highly similar files, we can thus get matches that are much longer than the limit imposed by the depth of the original LZW trie of S.

Preliminary tests with these variants gave encouraging results. For example, using the executable file `xfig.3.2.1.exe` as source file S to compress the next version `xfig.3.2.2.exe`, playing the role of T, the resulting delta file was smaller than 3K, whereas the original size of T was 812K, `Gzip` would reduce that only to 325K, and `LZW`, the method on which the delta encoding has been applied here, would yield a file of size 497K. The coding used was a combination of different Huffman codes for the characters, the offsets and the lengths. Non-compressed delta encoding could achieve even better results, but loose the advantage of working directly with the compressed files.

4 Future Work

Our exposition here has been mainly theoretical, presenting optimal algorithms (in the sense defined in [3]) for constructing delta files from LZW compressed data. We intend to extend these techniques also to LZ77 based compression, which resembles more to the basic delta encoding scheme.

References

1. Agarwal, R. C., Amalapurapu, S., and Jain, S.: *An approximation to the greedy algorithm for differential compression of very large files*, in Tech. Report, IBM Alamaden Res. Center, 2003.
2. Ajtai, M., Burns, R. C., Fagin, R., and Long, D. D. E.: *Compactly encoding unstructured inputs with differential compression*. Journal of the ACM, 49(3) 2002, pp. 318–367.
3. Amir, A. and Benson, G.: *Efficient two-dimensional compressed matching*, in Proceedings of the Data Compression Conference DCC–92, IEEE Computer Soc. Press, 1992, pp. 279–288.
4. Amir, A., Benson, G., and Farach, M.: *Let sleeping files lie: Pattern matching in z-compressed files*. Journal of Computer and System Sciences, 52 1996, pp. 299–307.
5. Burns, R. C. and Long, D. D. E.: *Efficient distributed backup and restore with delta compression*, in Workshop on I/O in Parallel and Distributed Systems (IOPADS), ACM, 1997.
6. Burns, R. C.and Long, D. D. E.: *In-place reconstruction of delta compressed files*, in Proceedings of the ACM Conference on the Principles of Distributed Computing, ACM, 1998.
7. Factor, M., Sheinwald, D., and Yassour, B.: *Software compression in the client/server environment*, in Proceedings of the Data Compression Conference, IEEE Computer Soc. Press, 2001, pp. 233–242.
8. Farach, M. and Thorup, M.: *String matching in lempel-ziv compressed strings*, in Proceedings of the 27th Annual ACM Symposium on the Theory of Computing, 1995, pp. 703–712.
9. Gąsieniec, L. and Rytter, W.: *Almost optimal fully lzw-compressed pattern matching*, in Proceedings of the Data Compression Conference, 1999, pp. 316–325.
10. P. Heckel: *A technique for isolating differences between files*. CACM, 21(4) 1978, pp. 264–268.
11. Hunt, J. J., Vo, K. P., and Tichy, W.: *Delta algorithms: An empirical analysis*. ACM Trans. on Software Engineering and Methodology 7, 1998, pp. 192–214.
12. Kida, T., Takeda, M., Shinohara, A., Miyazaki, M., and Arikawa, S.: *Multiple pattern matching in lzw compressed text*. Journal of Discrete Algorithms, 1(1) 2000, pp. 130–158.
13. Navarro, G. and Raffinot, M.: *A general practical approach to pattern matching over ziv-lempel compressed text*, in Proceedings of the 10th Annual Symposium on Combinatorial Pattern Matching CPM–99, vol. 1645, LNCS, Springer Berlin / Heidelberg, 1999, pp. 14–36.
14. Shapira, D. and Storer, J. A.: *In place differential file compression*. The Computer Journal, 48 2005, pp. 677–691.
15. J. A. Storer: *An Introduction to Data Structures and Algorithms*, Birkhauser/Springer, 2001.
16. W. F. Tichy: *The string to string correction problem with block moves*. ACM Transactions on Computer Systems, 2(4) 1984, pp. 309–321.
17. P. Weiner: *Linear pattern matching algorithms*, in Proceedings of the 14th Annual IEEE Symposium on Switching and Automata Theory (FOCS), 1973, pp. 1–11.
18. T. A. Welch: *A technique for high-performance data compression*. IEEE Computer, 17 June 1984, pp. 8–19.

Working with Compressed Concordances

Miri Ben-Nissan and Shmuel T. Klein

Department of Computer Science
Bar Ilan University
52 900 Ramat-Gan
Israel
Tel: (972–3) 531 8865
Fax: (972–3) 736 0498
miribn@gmail.com, tomi@cs.biu.ac.il

Abstract. A combination of new compression methods is suggested in order to compress the concordance of a large Information Retrieval system. The methods are aimed at allowing most of the processing directly on the compressed file, requesting decompression, if at all, only for small parts of the accessed data, saving I/O operations and CPU time.

Keywords: compression, concordance, full-text retrieval

1 Introduction

Research in Data Compression has recently dealt with the *compressed matching* paradigm, in which a compressed text T is searched directly for the occurrence of a given pattern P, rather than the usual approach of first decompressing the text and then searching for P in the original text. This has several advantages, such as enabling the search for some pattern on remote computers, saving time and space for the decoding, etc. Not every compression method is suitable for compressed matching, but many are, e.g. static Huffman coding [13], LZW [1] and many others [16,15,5].

But there are also other file types for which compressed matching could be advantageous. In a large full-text Information Retrieval System (IRS), a text is not searched directly, but by means of auxiliary files, such as a dictionary and a concordance. Searching in compressed dictionaries [12] means that the dictionary, which is the list of all the different terms in the database, is searched in its compressed form, by compressing the term to be looked up. The present work now extends the paradigm to the other large file of the IRS, namely the concordance.

The *concordance* gives for each word W in the database a list $\mathcal{L}(W)$ of indices to all its locations. We shall refer to such indices as the *coordinates* of W. In order to find all the places in which the words A and B occur, one has to intersect $\mathcal{L}(A)$ with $\mathcal{L}(B)$. Often, each keyword represents a family of linguistically different variants, which are all semantically equivalent to the given keyword. In this case, we first need to merge the coordinates of each variant in this list, before processing the query itself [4].

In fact, our approach is slightly different from the classical compressed matching framework. A simple search as in a compressed text does not make any sense, since one usually does not try to locate a single coordinate in the file. What is rather needed is a way to process entire lists of coordinates, which leads to our topic of working with compressed concordances.

There are several ways to build the concordance. Each depends on the needs of the system and the accuracy level that the system wants to support. Typically, a coordinate of a word W is a quintuple (b, r, p, s, w), where b is the index of the book in which the word W appears, considering the book as the highest level in the hierarchy describing the locations of the words in the database. The other elements of the coordinate are: r, the index of the part within the book; p, the index of the paragraph (within the part); s, the sentence number (in the paragraph), and w, which is the word number (in the sentence). However, there are Information Retrieval systems that do not support such an accuracy level, and the concordance keeps then only a document number and maybe also the paragraph number. In this work, we assume that the concordance consists of quadruples of the form (d, p, s, w), where we have merged the pair (b, r) to a single *document* field.

In order to achieve convenient computer manipulations, one may choose to keep a fixed length for each field of a coordinate. In this case, each field must be long enough to hold the representation of the maximal possible value. This, however, is extremely wasteful, since most of the values are small. On the other hand, if we decide to keep a variable length encoding for each field, some extra information is needed, to be able to read the concordance and identify the boundaries of each coordinate.

The problem with concordances is that their initial size may be 50–300 % of the size of the plain text itself. In addition, using the original concordance when processing a query often leads to a very large set of coordinates which need to be checked. It is therefore necessary to compress the concordance. However, if the database is queried frequently, there is a large time and space overhead for decompressing the necessary parts of the concordance. For a query of the form A AND B, we need to extract all the coordinates of term A, and intersect them with all the coordinates of term B, all at runtime. In large textual databases this may require a huge amount of data to decompress, and technically, large parts of the concordance will be in uncompressed form most of the time [5].

The present work is an extension of the compressed pattern matching problem, as it presents a new compression method for compressing the concordance, which is aimed to allow working directly with the compressed concordances, and accessing the extracted data only for final decision, while most of the work is done on the compressed part. All logical operations will be applied on the encoded concordance itself. The method to be presented is suitable for static databases, since it relies on statistical information, collected off line before the compression phase is done.

The paper is organized as follows: in the next section, we review previous and related work in the area of concordance compression methods, and discuss its applicability to compressed matching. Section 3 then presents the details of the new algorithm.

2 Previous and Related Work

2.1 The Prefix Omission Method

The most basic compression method that can be applied on concordances is the Prefix Omission Method (POM)[4]. This method is based on the observation that since the coordinates are ordered, consecutive coordinates may share the same b, r, p or even s fields (obviously, different coordinates cannot share all 5 fields). In that case, we can omit the repeated fields from the second coordinate. In order to maintain the ability of reconstructing the file, we need to adjoin a *header* to that coordinate, holding information on which fields are to be copied from the preceding coordinate. For a coordinate of the form (d, p, s, w), it is sufficient to keep a 2-bit header for the four possibilities: don't copy any field from the previous coordinate, copy the d-field, copy d and p fields, and copy fields d, p and s. For example, if the d field is the same as in the preceding coordinate, we shall save as coordinate only the triple (p, s, w), with header 01. Although POM yields quite good compression, it is not possible to work directly with the compressed file.

2.2 Variable Length Fields

As mentioned before, one may choose, for the ease of implementation, to represent each field in the coordinates by a fixed length code, at the cost of keeping a much larger file; turning to variable length codes may reduce its size considerably. But in this case, we need to save extra information to be able to read the file and identify the field boundaries. The idea is to add a fixed-length header to each field, which contains a codeword that represents either the length (in bits) of the stored value, or the value itself, which is useful for specially frequent values in a given field. A small table is kept translating those codewords to the corresponding values.

2.3 Numerical Compression

Most of the data stored in the concordance is numerical and traditional compression algorithms cannot provide sufficient compression for such data. In the concordances, all the values are sorted in non-descending order. For some of the fields, we would like to find a method that allows to compare the values of two compressed elements directly, without decoding first. For other fields we just need a good compression method, that will allow us to identify the element's boundaries fast. Studying the distribution of each of the concordance components can help us to choose the proper compression technique to encode it [3].

A well known technique for compressing lists of non-decreasing values is *Delta Encoding*, where each element, except the first, is replaced by difference from its predecessor [2,7], resulting in smaller values to be stored. This can be done either directly, or, e.g., devising a Huffman code for the differences. Witten *et al.* [17] review some compression techniques that can be applied on those delta values and compare their performances. Linoff et al. [14] also presented some techniques for compressing numerical data, such as n–s Coding.

There are more techniques that use variable blocks with escape codes, such as the Elias codes, Fibonacci codes, etc. The idea is to set aside one bit per block as a *flag-bit*, indicating the lengths of the blocks. The γ *Elias Codes* [6] maps an integer x onto the binary value of x prefaced by $\lfloor \log(x) \rfloor$ zeros. The binary value of x is expressed in as few bits as possible and therefore begins with a 1, which serves to delimit the prefix. Using this code, each integer with N significant bits is represented in $2N + 1$ bits [8]. The *Fibonacci Code* is a universal variable length encoding of integers, based on the Fibonacci sequence, rather than on powers of 2. The advantage of this scheme is its simplicity, robustness and speed. In this code, there are no adjacent 1's, so that the string 11 represent a delimiter between two codewords [9].

3 An algorithm enabling work on the compressed concordance

3.1 General layout of the compressed file

In the suggested algorithm, the concordance will be compressed along the lines described in [4], with a few adaptations intended to facilitate the work directly within the compressed file. For the ease of description, we shall use the statistics of a real life concordance, that of a subset of the Responsa Retrieval Project [10], the relevant parameters of which appear below in Table 1. It should, however, be emphasized that the methods to be described are general in nature, and could straightforwardly be adapted to concordances of other natural language full text Information Retrieval Systems, of different sizes, for different languages and even with different hierarchical structure of the coordinates.

number of unique words	725,966
total number of words (coordinates)	80,928,240
number of documents	67,937
average number of coordinates per word	111.48
average number of documents per word	52.33
average number of coord. per word in each doc.	2.13
average size of coordinate	7.33 bits
average size of delta field	3.30 bits
maximum size of document field	17 bits
average size of document delta values	3.55 bits

Table 1. Statistics of the Responsa Retrieval Project concordance

Figure 1 shows the layout of a full coordinate, as it is handled by the retrieval procedures of the Responsa Project. The numbers under each field design their sizes, in bytes, for a total of 8 bytes per coordinate. The main idea of the compression is to represent the values in the different fields using a variable number of bits. This reduces the average number of bits needed, since most values stored in the coordinate are quite small, while the size of the fields in the full coordinate are chosen to accommodate the highest possible values, which occur only extremely rarely. The meta-information necessary for the decompression is kept, for each coordinate, in a fixed length *header*, which is itself partitioned into fields, each encoding the number of bits needed in the corresponding field of the compressed coordinate, as represented in Figure 2.

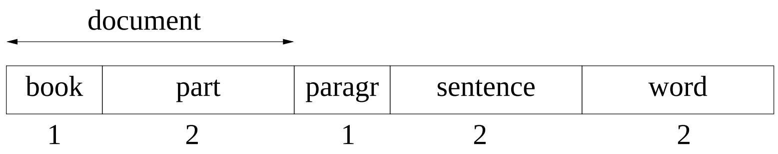


Figure 1. Layout of a non-compressed coordinate

In the original algorithm of [4], headers and coordinates were stored in an alternating sequence. This did not cause any problem, since for the processing of a query A AND B, the entire lists $\mathcal{L}(A)$ and $\mathcal{L}(B)$ were first fetched from the disk, then decompressed and finally intersected. To avoid the decompression of many coordinates,

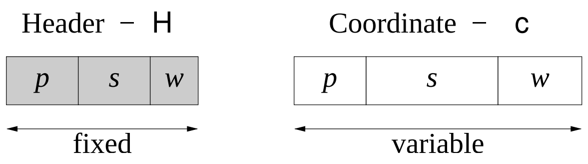


Figure 2. Layout of a compressed coordinate

it is convenient to partition the compressed file into two parts: the variable length compressed coordinates in one file, which we call the *Coordinates* file, and the fixed length headers in a *Headers* file. A high level schematic representation of the layout of the compressed files is given in Figure 3.

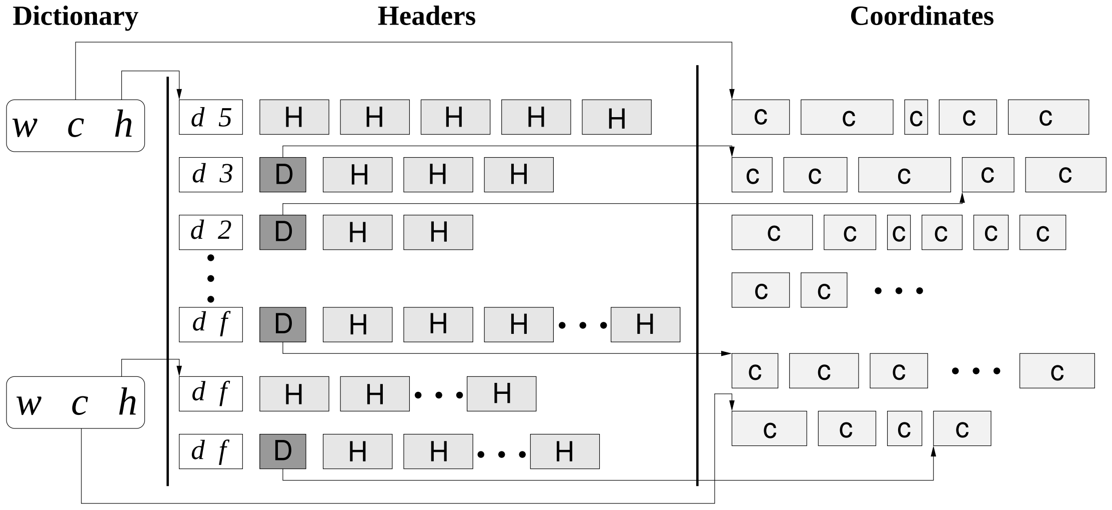


Figure 3. Layout of the compressed concordance

The concordance is accessed via the *Dictionary*, including all the different words of the text. The dictionary may be stored in a variety of ways, e.g., in lexicographic order of the terms, or sorted first by length and only internally in alphabetic order, or in form of a trie or even as a hash table. The main purpose is fast and direct access to the information stored for each word. The size of the dictionary is usually of a lower order of magnitude than that of the concordance, and by Heaps's Law [11, p. 206–208], its size is $\alpha\ N^{\beta}$, where α and β are suitable constants, $0.4 \leq \beta \leq 0.6$, and N is the number of words in the text.

Each word w_i in the dictionary is stored along with two pointers c_i and h_i, to the corresponding entries in the Coordinates and Headers files, respectively. In the Headers file, the items are grouped by document number: let n_i be the number of documents the i-th word, w_i, appears in, and let $d_{i,1}, d_{i,2}, \ldots, d_{i,n_i}$ be the indices of these documents; let $f_{i,j}$, $1 \leq j \leq n_i$ be the number of times w_i appears in document $d_{i,j}$. The entry in the Headers file corresponding to w_i then starts with the pair $(d_{i,1}, f_{i,1})$, followed by $f_{i,1}$ fixed length headers, followed then by $(d_{i,2}, f_{i,2})$, etc.

Note that by grouping the coordinates by documents, the document fields can be omitted, but this comes at the cost of having to add the corresponding number $f_{i,j}$ at the beginning of each group. This is very often a reasonable overhead, especially in Information Retrieval systems supporting, as many do, also a ranking of the passages to be retrieved. The ranking is performed by means of some scores that are usually based, among others, on the local frequencies $f_{i,j}$, so that these are needed anyway.

The Coordinates file consists of the sequence of the variable length (p, s, w) fields. There is no need to separate the elements belonging to different words, since the pointers c_i provide direct access to the elements belonging to w_i. In fact, if for a given word w_i one would always process the full list $\mathcal{L}(w_i)$, the data mentioned so far would be enough to restore the full list. However, since we wish to move as much of the processing as possible to the compressed domain, we would like to perform much of the work on the Headers file and access the Coordinates file only occasionally. We thus need, in addition to the pointer c_i, a sequence of pointers $D_{i,j}$, for $2 \leq j \leq n_i$ (the darker elements in Figure 3), pointing to the first element in the Coordinates file belonging to document $d_{i,j}$.

3.2 Compression of Coordinates

***p*, *s* and *w* fields** To compress the coordinates, one has first to collect statistics about the exact distribution on the values that appear in the different fields. These lengths are then partitioned into classes, e.g., according to the number of bits needed to represent each of the values. That is, the 1-bit class corresponds to the single value 1, the 2-bit class corresponds to 2 and 3, etc. On the basis of the frequencies of each of the classes, a Huffman code could be devised, for optimal compression. We prefer, however, to use a fixed length encoding of the classes, to facilitate the work directly on the compressed file. Table 2 gives the distribution of the values in the various fields for our test database. The columns correspond to the number of bits needed (up to and including the leading 1-bit) to represent the given numbers, and the values in the table are percents.

Bits	1	2	3	4	5	6	7	8	9	10	11-15
Range	1	2–3	4–7	8–15	16–31	32–63	64–127	128–511	512–1023	1024-2043	2048–65535
p-field	6	21	25	23	15	7	3	1			
s-field	25	30	20	12	7	4.5	1	0.5	0.01		
w-field	1.4	4	9.2	14	20	21	18	12	0.4		
delta	0.5	1.2	13	55.8	17.8	7.6	2.9	0.9	0.3	0.1	< 0.01

Table 2. Distribution of values grouped by lengths (in bits) of the numbers

As can be seen in Figure 2, the fixed length header allocated to each coordinate consists of 8 bits: 3 bits for each of the p and s fields, and 2 bits for the w field. This allows eight options for p and s and four options for w, which are depicted in Table 2. Option 0 for the p and s fields represent the fact that one copies the corresponding values from the previous coordinate, extending thereby the POM technique. The last line of Table 3 shows the number of times (in percent) a value is equal to that of the preceding coordinate. Since for the w field, this happens only rarely, the copy option is not used for w. To understand the values in Table 3, consider for example the w-field: if the code in the corresponding 2-bit header is 10, the w-field in the coordinate itself will be encoded by 7 bits.

The following amendment should be mentioned, which saves actually quite a few bits: when one considers the number of bits *needed* to encode an integer, one usually

	p-field (3 bits)	s-field (3 bits)	w-field (2 bits)
lengths of coord. fields	0 1 2 3 4 5 6 **8**	0 1 2 3 4 5 6 **9**	**5** 6 7 **9**
possible omissions	36%	42%	3%

Table 3. Interpretation of the codes in the header

refers to the number of significant bits, up to and including the leftmost 1-bit. So for the number 12, in binary 1100, one needs 4 bits. However, if one knows the *exact* number of bits needed, then the leftmost bit is in fact superfluous, since it must be a 1. That is, if the number to be encoded is 12, and we store the information that 4 bits are needed, then one needs only 3 additional bits, giving the relative index in the range $[2^3, 2^4 - 1]$. Returning to Table 3, one can save a bit in the coordinate fields in all cases where the header field gives the exact number of bits. Only for the others, indicated in bold-face in Table 3, one really needs all the bits as indicated by the value in the header. For example, the code 111 in the p-field header stands for a length of 8 bits, but since there is no code for a 7-bit integer, one cannot be sure that the highest bit is a 1, thus all 8 bits are needed; but if the code were 110, the corresponding length would be 6 and here we see that there is also a code for a length of 5, thus the encoded integer must be in the range $[32, 63]$, for which 5 bits are enough.

document, frequency and delta The 3-byte document index is first translated into a running index. Since the number of documents is only about 68000, one can encode an element of the index in 17 bits. The first element $d_{i,1}$ for each word w_i will indeed be encoded that way, but the subsequent elements $d_{i,j}$ for $j > 1$ will be delta encoded, that is, we store actually $d_{i,j} - d_{i,j-1}$. These differences have a skew distribution, and need, on the average, only 3.55 bits for their encoding. The classes corresponding to the different lengths are Huffman encoded, yielding an average codeword length of 3.45 bits.

The best encoding for the the frequencies $f_{i,j}$ is a unary one. Table 4 lists for the first few values, the percentage of (d, f) pairs having that value in their $f_{i,j}$ field, indicating that there is a rapid exponential decrease. The value 1 can thus be encoded by a single bit 1, the value 2 by 01, 3 by 001, etc. This yields an average of 2.13 bits for each $f_{i,j}$ value.

$f_{W,j}$	1	2	3	4	5	6	7	8	9	$\cdots$
%	67	15	6	3	2	1	1	0.7	0.5	$\cdots$

Table 4. Distribution of $f_{W,j}$

The location pointers $D_{i,j}$, for $j > 1$, pointing into the Coordinates file, can also be encoded by their differences, similarly to the $d_{i,j}$ values. Partitioned into classes by their lengths in bits (refer to the last line of Table 3), one needs 3.3 bits on the average for each value, plus 1.96 bits for an average Huffman codeword indicating to which class it belongs.

Table 5 summarizes the average lengths for one word. The first column is the parameter we compute, the second column holds the average number of times the corresponding parameter appears for a single word in the concordance, and the third column holds the average number of bits needed to represent that parameter.

parameter	mult factor	number of bits
$d_{i,1}$	1	17 bits
$d_{i,j}$ for $j > 1$	51.33	3.55+3.45 = 7.0 bits
$f_{W,j}$	52.33	2.13 bits
$D_{i,j}$ for $j > 1$	51.33	3.3+1.96 = 5.26 bits
header block	111.48	8 bits
coordinate	111.48	7.33 bits

Table 5. Components of the compressed coordinate

To calculate the total number of bits needed for each coordinate in the suggested compressed concordance, we multiply the second and third columns, and then divide the total sum by the average number of coordinates per a word. Following the above statistics, we get an average of 22.12 bits = 2.77 bytes for each coordinate. Recall that the original, uncompressed coordinate was of length 8 bytes, so we get a compression ratio of 2.9, in spite of the fact that we have added also information on the local frequencies $f_{i,j}$.

3.3 The Algorithm

The algorithm below shows how to work with the compressed concordance. Given a query

$$\mathcal{Q} = q_1\ (l_1, u_1)\ q_2\ (l_2, u_2) \cdots q_{m-1}\ (l_{m-1}, u_{m-1})\ q_m,$$

where q_i $(1 \leq i \leq m)$ is a term. The couple (l_i,u_i) imposes a lower an upper limit on the distance from q_i to q_{i+1}, that is, an m-tuple of coordinates $(c_1, \ldots, c_m)$ is considered relevant if c_i belongs to the list of coordinates of q_i and

$$l_i \leq dist(c_i, c_{i+1}) \leq u_i \qquad \text{for } 1 \leq i < m.$$

Negative distance means that q_{i+1} may appear before q_i in the text. The distance is measured in words. Note that this is a conjunctive query, and in real life applications, each terms q_i would in fact stand for a disjunction of several terms, all considered equivalent for the given query.

In the algorithm below, we use the following notations: *HVal(w)* and *CVal(w)* return the pointers h and c, respectively, of the word w from the dictionary. *fp* holds the position (in bits) in the Headers file. *index(w)* returns the index of the word w in the dictionary. The constant HS denotes the size of each header block (8 bits in our implementation).

```
AND Processing(q1, q2)
i ← 1,   j ← 1
a ← index(q1),   b ← index(q2)
fp1 ← HVal(q1),   fp2 ← HVal(q2)
Queue1 ← empty,   Queue2 ← empty
limit1 ← HVal(a + 1),   limit2 ← HVal(b + 1)
pos1 ← 0,   pos2 ← 0
mask ← 11111100
read ⟨d_{a,i}, f_{a,i}⟩ from H(q1)
read ⟨d_{b,j}, f_{b,j}⟩ from H(q2)
WHILE  fp1 < limit1  AND  fp2 < limit2  DO:
    IF  d_{a,i} < d_{b,j}  THEN:
        IF  i > 1  THEN:
            read deltaCode from H(q1)
            Push deltaCode into Queue1
        fp1 ← fp1 + HS · f_{a,i}
        i ← i + 1
        read ⟨d_{a,i}, f_{a,i}⟩ from H(q1)
    ELSE IF  d_{a,i} > d_{b,j}  THEN:
        IF  j > 1  THEN:
            read deltaCode from H(q2)
            Push deltaCode into Queue2
        fp2 ← fp2 + HS · f_{b,j}
        j ← j + 1
        read ⟨d_{b,j}, f_{b,j}⟩ from H(q2)
    ELSE: //the same document number
        ii ← 1
        jj ← 1
        WHILE  ii < f_{a,i}  AND  jj < f_{b,j}  DO:
            IF  h1_{ii} ∧ mask  <  h2_{jj} ∧ mask  THEN: //compare first 6 bits of the header
                ii ← ii + 1
            ELSE IF  h1_{ii} ∧ mask  >  h2_{jj} ∧ mask  THEN:
                jj ← jj + 1
            ELSE: // p and s fields in the header are equal
                //compute the position of two corresponding coordinates
                pos1 ← CVal(q1) + sum of elements in Queue1
                pos2 ← CVal(q2) + sum of elements in Queue2
                //compare the extracted coordinates
                WHILE  h1_{ii} ∧ mask = h2_{jj} ∧ mask  DO:
                    read Coord1.p, s from pos1
                    read Coord2.p, s from pos2
                    IF  Coord1.p, s < Coord2.p, s  THEN:
                        ii ← ii + 1
                    ELSE IF  Coord1.p, s > Coord2.p, s  THEN:
                        jj ← jj + 1
```

```
45                         ELSE: // Coord1.p,s = Coord2.p,s
46                             //make sure that the w fields are in the right range
47                             read  Coord1.w  and  Coord2.w
48                             IF  l ≤ Coord1.w − Coord2.w ≤ u   THEN:
49                                 return  Coord1  and  Coord2
50                             ELSE IF   Coord1.w < Coord2.w   THEN:
51                                 ii ← ii + 1
52                             ELSE:
53                                 jj ← jj + 1
54                   //we compared all the headers that has the same p and s codes
55                   //and didn't find a match
56                   return false
57   //we finished to compare all H(q1) and H(q2) and found no match
58   return false
```

Note that up to line 38, the processing deals only with the Headers file, which is used in its compressed form, the main idea of the suggested compression being that compressed headers can be compared directly as if they were numbers, that is, the compression methods preserve order.

4 Experimental design

It obviously makes no sense to try to compare empirically the retrieval time by the compressed algorithm to that of decompression and afterwards retrieval, using a set of "random" queries. A query consisting of random terms will most probably retrieve an empty set of locations. An empirical study should thus involve what could be called a "typical query", though this is hard to define. We therefore leave the present proposal on the theoretical level.

5 Conclusions

Although the compression methods presented do not necessarily give the most effective compression, they give a quite good compression ratio after all on the one hand, and on the other hand allow working directly with the compressed concordance, thus saving expensive I/O and CPU operations.

Acknowledgement: This work has been supported in part by Grant 25915 of the Israeli Ministry of Industry and Commerce (Magnet Consortium Kite).

References

1. Amir, A., Benson, G., and Farach, M.: *Let sleeping files lie: Pattern matching in z-compressed files.* Journal of Computer and System Sciences, 52 1996, pp. 299–307.
2. Anh, V. and Moffat, A.: *Compressed inverted files with reduced decoding overheads*, in Proceedings of the 21st Annual SIGIR Conference on Research and Development in Information Retrieval, ACM Press, 1998, pp. 290–297.
3. Bookstein, A., Klein, S.T., and Raita, T.: *Model based concordance compression*, in Proceedings of the Data Compression Conference DCC–92, IEEE Computer Soc. Press, 1992, pp. 82–91.

4. Choueka, Y., Fraenkel, A.S., and Klein, S.T.: *Compression of concordances in full-text retrieval systems*, in Proceedings of the 11st Annual SIGIR Conference on Research and Development in Information Retrieval, ACM Press, 1988, pp. 597–612.
5. E. S. de Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates: *Fast and flexible word searching on compressed text.* ACM Trans. Inf. Syst., 18(2) 2000, pp. 113–139.
6. Elias, P.: *Universal codeword set and representations of the integers.* IEEE Trans. Information Theory, IT–21(2) 1975, pp. 194–203.
7. Engelson, V., Fritzson, D., and Fritzson, P.: *Lossless compression of high-volume numerical sata from simulations*, in Proceedings of the Data Compression Conference DCC–00, IEEE Computer Soc. Press, 2000, p. 547.
8. P. Fenwick: *Punctured elias codes for variable-length coding of the integers.*
9. Fraenkel and Klein: *Robust universal complete codes for transmission and compression.* Discrete Applied Mathematics, 64 1996, pp. 31–55.
10. Fraenkel, A.S.: *All about the responsa retrieval project you always wanted to know but were afraid to ask, expanded summary.* Jurimetrics Journal, 16 1976, pp. 149–156.
11. J. Heaps: *Information Retrieval : Computational and Theoretical Aspects*, Academic Press, Inc., New York, NY, 1978.
12. S. T. Klein and D. Shapira: *Searching in compressed dictionaries*, in Proceedings of the Data Compression Conference DCC–02, Washington, DC, USA, 2002, IEEE Computer Society, pp. 142–151.
13. S. T. Klein and D. Shapira: *Pattern matching in huffman encoded texts.* Inf. Process. Manage., 41(4) 2005, pp. 829–841.
14. G. Linoff and C. Stanfill: *Compression of indexes with full positional information in very large text databases*, in SIGIR '93: Proceedings of the 16th annual international ACM SIGIR conference on Research and development in information retrieval, New York, NY, USA, 1993, ACM Press, pp. 88–95.
15. U. Manber: *A text compression scheme that allows fast searching directly in the compressed file.* ACM Trans. Inf. Syst., 15(2) 1997, pp. 124–136.
16. M. Takeda, S. Miyamoto, T. Kida, A. Shinohara, S. Fukumachi, T. Shinohara, and S. Arikawa: *Processing text files as is: Pattern matching over compressed tests*, in Proceeding of the 9th International Symposium on String Processing and Information Retrieval (SPIRE'2002), LNCS 2476, 2002, pp. 170–186.
17. I. H. Witten, A. Moffat, and T. C. Bell: *Managing Gigabytes: Compressing and Indexing Documents and Images*, Morgan Kaufmann Publishers, San Francisco, CA, 1999.

The Gapped-Factor Tree

Pierre Peterlongo[1], Julien Allali[1], and Marie-France Sagot[2,3] ⋆

[1] Institut Gaspard-Monge, Université de Marne-la-Vallée, France
pierre.peterlongo@univ-mlv.fr
[2] INRIA Rhône-Alpes and Laboratoire de Biométrie et Biologie Évolutive, UMR 5558, Université Claude Bernard, Lyon, France
[3] King's College, London, UK

Abstract. We present a data structure to index a specific kind of factors, that is of substrings, called *gapped-factors*. A gapped-factor is a factor containing a gap that is ignored during the indexation. The data structure presented is based on the suffix tree and indexes all the gapped-factors of a text with a fixed size of gap, and only those. The construction of this data structure is done online in $O(n \times |\Sigma|)$ time and space, with n the length of the text and $|\Sigma|$ the size of the alphabet. Such a data structure may play an important role in some pattern matching and motif inference problems, for instance in text filtration.

Keywords: suffix tree, k-factor factor tree, string index, gapped-factor, gapped-factor tree

1 Introduction

The indexation and extraction of repeated short words (called k-*factors*[4] for words of length k) has become a widely used technique in many text algorithmic problems. One can mention their use in, for instance, FASTA [17] and BLAST [2,3]. Indeed, many algorithms for efficiently computing string matches [10,24,29] or alignments [5,4,9,12,16,18,21] use k-factors. In particular, filtration algorithms that have been created for quickly discarding large portions of the input before applying a more expensive algorithm on the remaining data are often based on the identification of such short repeated words [6,7,8,15,25,26,28].

Among the exact filtration algorithms (exact in the sense that they discard only portions of the text that can not be part of the final solution sought), some consider k-factors composed of non consecutive letters [7,8,15,26], or sets of k-factors [6,25,28]. Both present advantages for filtering purposes in comparison with single k-factors with no letters skipped as shown in [7,14,15].

In order to efficiently use such k-factors, one needs data structures to index them. Depending on the kind of k-factor adopted, different types of data structures may be considered. For instance, sets of k-factors may be indexed in a hash table or using a labelling technique as proposed in [13]. In this paper, we introduce a data structure designed for the indexation of sub-words composed of a k-factor, a gap of length d not taken into account during the indexation and a k'-factor. Such a sub-word is called a *gapped-factor* as it contains a unique gap.

The new data structure is an adaptation of the suffix tree [20]. More precisely, the construction we describe in this paper is an adaptation of the construction of a

⋆ Supported by the ACI Nouvelles Interfaces des Mathématiques π-*vert* project of the French Ministry of Research, the ARC *BIN* project from the INRIA, and the ANR project *REGLIS*.

[4] Another currently used term for designing k-factors is q-grams

k-factor tree [1], which itself is an extension of the Ukkonen construction of a suffix tree [31]. A k-factor tree is a tree indexing all k-factors of a text.

As indicated in Section 5, the new data structure, called a *gapped-factor tree*, allows to extract in linear time all the repeated gapped-factors of a text or of a set of texts. Furthermore, it offers the possibility to obtain in time $O(k + k')$ the list of all the positions of a gapped-factor.

The paper is organised as follows. In Section 2, we provide the context and some definitions about text and trees. In Section 3, we formally introduce gapped-factors and the gapped-factor tree. In Section 4, we present the algorithm to construct a gapped-factor tree for indexing the gapped-factors of a text after recalling the Ukkonen construction of a suffix tree and the Allali construction of a k-factor tree. We end by indicating two basic uses of gapped-factor trees.

2 Preliminaries

A *text*, also called a *string*, is a sequence of zero or more symbols from an alphabet Σ. A text t of length n is denoted by $t[0, n-1] = t_0t_1 \dots t_{n-1}$, where $t_i \in \Sigma$ for $0 \leq i < n$. The length of t is denoted by $|t|$. A string w is a *factor* of t if $t = uwv$ for $u, v \in \Sigma^*$; in this case, the string w occurs at position $|u|$ in the string t. A k-factor denotes a factor of length k. If $t = uv$ for $u,\ v\ \in \Sigma^*$ then v is called a *suffix* of t. A suffix starting at position i in t is denoted by $t_{i\dots}$.

A tree is a data structure composed of **nodes** connected together by **edges**. Except for a special node called the **root**, each node has exactly one **father**. Nodes with no children are called the **leaves** while all other nodes are called the **internal nodes** of the tree. An internal node having at least two children is called a **branching node**.

We call the **depth** of a node $\mathcal{N}$ the sum of the lengths of the edges that need to be traversed from the root of the tree to reach $\mathcal{N}$. By definition, the depth of the root is thus 0.

Nodes and edges may be labelled. For instance, in Figure 1, edges are labelled with letters from a given alphabet.

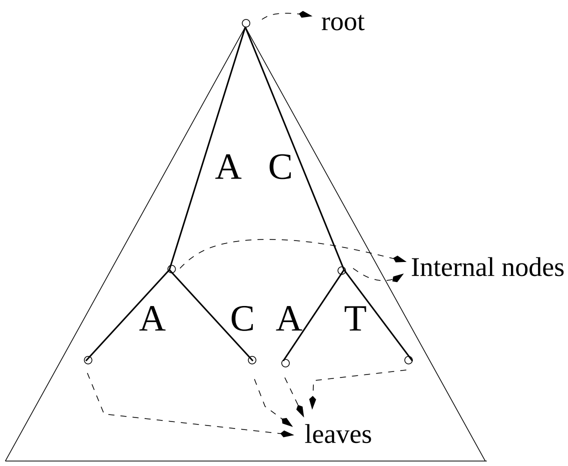


Figure 1. Example of a tree labelled with letters from a given alphabet. Reading all paths from the root to the leaves, leads to the strings AA, AC, CA and CT.

Let $\mathcal{N}$ be a node of a tree, we denote by $path(\mathcal{N})$ the text corresponding to the concatenation of the letters from a given alphabet labelling the edges from the root to $\mathcal{N}$.

For instance, if $\mathcal{N}_0$ denotes the leftmost leaf of the tree presented in Figure 1, $path(\mathcal{N}_0) = AA$.

The suffix trie of a text t is a tree with edges labelled with elements of Σ. For each factor of t, there exists a node N such that $path(\mathcal{N})$ is equal to that factor. If t has an ending symbol, all nodes $\mathcal{N}$ for which the path from the root spells a suffix of t are leaves.

The *implicit* suffix tree of t is a tree with edges labelled by non-empty elements of Σ^*. The suffix tree is a compressed version of the suffix trie. Each internal node $\mathcal{N}$ of the suffix trie that has only one child is deleted and its two adjacent edges are replaced by an edge that goes from the father of $\mathcal{N}$ to its child. The label of the new edge is equal to the concatenation of the label of the edge going from the father of $\mathcal{N}$ to $\mathcal{N}$ and of the label of the edge from $\mathcal{N}$ to its child. This tree is called implicit because not all suffixes of t lead to a leaf. The true suffix tree is obtained when a special ending symbol \$ not in Σ is added at the end of t. A suffix tree indexes all the $|t|$ suffixes of a text t.

3 Gapped-factor tree

A gapped-factor tree indexes gapped-factors that are defined as follows:

Definition 1 (Gapped-factor). *A* **gapped-factor** *is a concatenation of a factor of length k, a gap of length d and another factor of length k'. A gapped-factor occurring at position i in a text t is $t[i, i+k-1].t[i+k+d, i+k+d+k'-1]$. Such a gapped-factor is called a $(k$-d-$k')$-gapped-factor.*

An example of a (2-1-3)-gapped-factor is given in Figure 2.

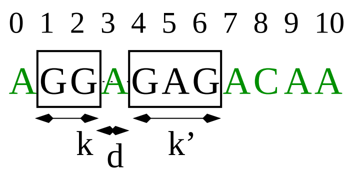


Figure 2. Example of a (2-1-3)-gapped-factor. The first factor length is $k = 2$, the gap is of length $d = 1$ and the second factor has a length $k' = 3$. It occurs at position 1 in the text. With these parameters, the content of the gapped-factor occurring at position 1 is *GGGAG* composed by *GG* and *GAG*.

We propose a new data structure, called a *gapped-factor tree*, to index all the $(k$-d-$k')$-gapped-factors of a text or of a set of texts. This is a modification of the suffix tree [20] data structure. The gapped-factor tree takes into account the gap of length d of the gapped-factors it indexes. This means that the tree contains a region up to which the k-factors are indexed as in a classical suffix tree, while below this region the second factors (of length k') of the $(k$-d-$k')$-gapped-factors starting with the same k-factor start from the same node. This region is called the *invisible region*.

An intuitive idea of such a data structure is given in Figure 3.

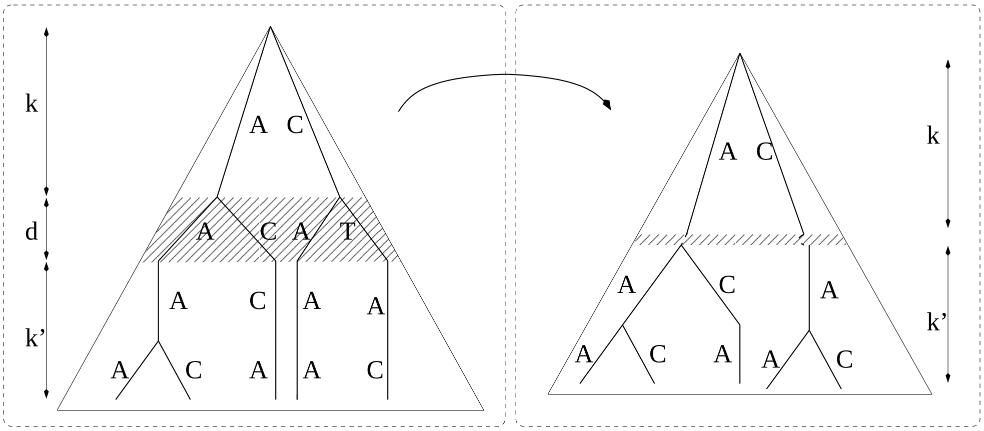


Figure 3. An intuitive view of a gapped-factor tree. Even if this is not the way the gapped-factor tree is constructed, a gapped-factor tree can be seen as a truncated suffix tree where a part has been removed, provoking merges in the lower part of the tree.

Definition 2 (Path in a Gapped-Factor Tree). *Let w be a $(k$-d-$k')$-gapped-factor starting at position $i < |t| - k - d - k'$ that is indexed in such a tree. Let $\mathcal{N}$ be the node at depth $z \leq k + k'$ corresponding to this $(k$-d-$k')$-gapped-factor. Then:*

$$path(\mathcal{N}) = \begin{cases} t[i, i+z-1] & \textit{if } z \leq k \\ t[i, i+k-1].t[i+k+d, i+d+z-1] & \textit{otherwise} \end{cases}$$

An example of gapped-factor tree and of a path in such a tree is presented in Figure 4.

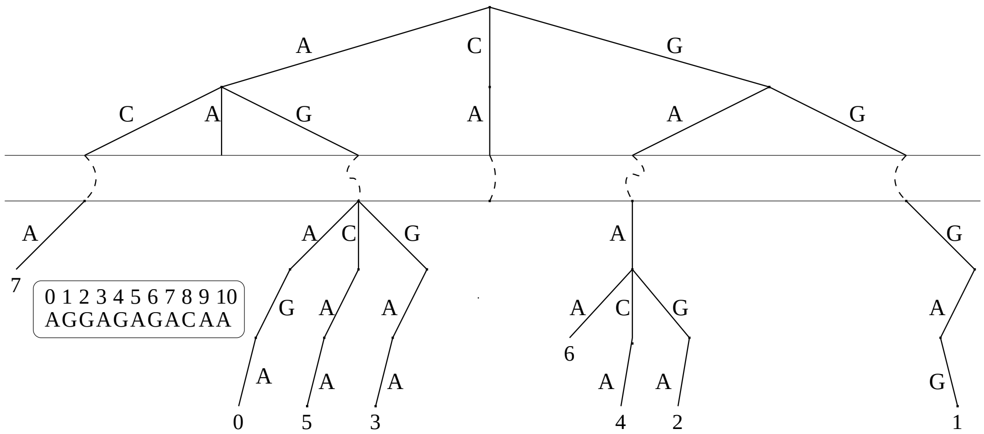


Figure 4. Example of gapped-factor tree. The input sequence is *AGGAGAGACAA*. The dashed lines correspond to the invisible region of the tree. In this case, the gapped factors indexed are (2-1-3)-gapped-factors. The information attached to one of the leaves corresponds to the starting positions of a gapped-factor in the text.

In the next section, we present the algorithm which performs the online construction of a gapped-suffix tree.

4 Construction

The algorithm for constructing a gapped-factor tree is an extension of the algorithm for constructing a k-factor tree [1], which is itself an extension of the suffix tree construction algorithm due to Ukkonen[31]. Therefore, in the following, we start by presenting the construction of a suffix tree, then the one of a k-factor tree, and finally we describe the construction of a gapped-factor tree.

4.1 Ukkonen construction of the suffix tree

To present the Ukkonen algorithm, we follow the description given in [11]. This algorithm constructs a full suffix tree of a text t in $O(|t|)$ time and space. An example of a suffix tree is given in the Figure 5.

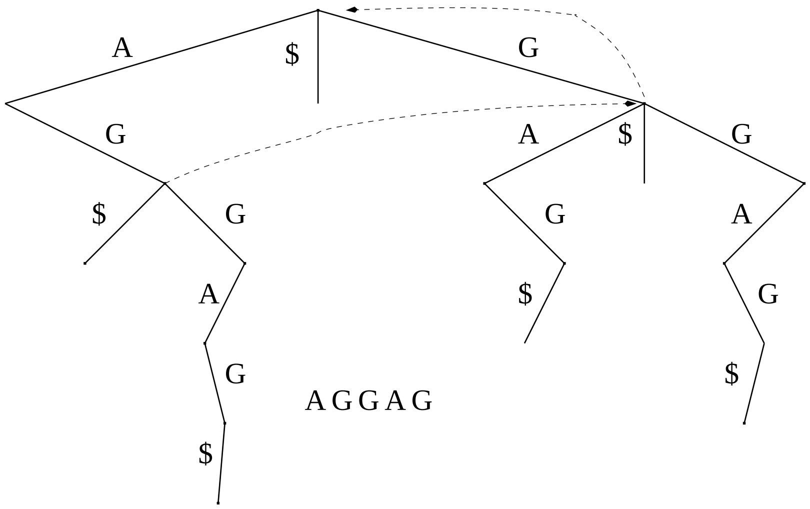


Figure 5. Example of a suffix tree for the text *AGGAG*\$. The dashed lines represent the suffix links.

The algorithm is divided into $|t|$ phases. The i^{th} phase (for $0 \leq i < |t|$) consists in the insertion of all the $i+1$ suffixes of $t[0,i]$ into the tree. The naive approach divides each phase i into $i+1$ steps, one step j ($0 \leq j < i$) consisting in the insertion of the suffix $t[j,i]$ into the tree. This naive version of the construction algorithm is presented in Algorithm 9. Clearly this algorithm is in $O(|t|^3)$.

Algorithm 9 Naive suffix tree construction algorithm

Require: A text t
Ensure: The suffix tree $ST(t)$ of t

```
for i from 0 to |t| − 1 do
   for j from 0 to i do
      Add(ST(t),t[j,i])
   end for
end for
```

The Ukkonen algorithm uses three *tricks* in order to reduce the time complexity to $O(|t|)$.

Before we present those three tricks, we describe the encoding of a suffix tree. The suffix tree created by this algorithm does not store the text: each node $\mathcal{N}$ contains a couple of integers (s, e) corresponding to the starting and ending positions of the factor in the text that led to the creation of the node itself. In the following, we denote by $\mathcal{N}_{s,e}$ such a node. Thus, by definition, in the suffix tree of a text t, $path(\mathcal{N}_{s,e})$ is equal to $t[s, e]$.

The Ukkonen algorithm uses suffix links. A *suffix link* is an oriented link between two branching nodes of a suffix tree. Given a node $\mathcal{N}_{s,e}$, its suffix link is denoted by $S_l(\mathcal{N}_{s,e})$ and the node pointed by $S_l(\mathcal{N}_{s,e})$ is denoted by $S_n(\mathcal{N}_{s,e})$. In this case, $path\left(S_n(\mathcal{N}_{s,e})\right) = path(\mathcal{N}_{s,e})\big[1, |path(\mathcal{N}_{s,e})|\big]$. For instance, if $path(\mathcal{N}_{s,e}) = AGGT$, then, $path(S_n(\mathcal{N}_{s,e})) = GGT$.

In Figure 5, the suffix links are represented by dashed lines.

We present the three ideas leading to a linear time complexity for constructing a suffix tree for the text t.

1. Let us assume that the suffix tree is constructed for $t[0, i-1]$. During the i^{th} phase, all the leaves have to be lengthened by one in order to take the character t_i into account. In other terms, the ending integer e of each leaf has to be incremented by one. Since by definition, all leaves have the same ending integer, the latter can be coded by a global variable that is incremented by one at each phase of the Ukkonen algorithm. This global variable is equal to i during phase i. Thus, the extension of the leaves is implicit and done in constant time.
2. (a) *Fast Insertion*: during the i^{th} phase, let $\mathcal{N}_{s,e}$ be the last branching node reached during the insertion of $t[j, i]$. By construction this node contains a suffix link. In this case, $t[j, i] = path(\mathcal{N}_{s,e}).w.\sigma$ where $w \in \Sigma^*$ and $\sigma \in \Sigma$. In order to insert $t[j+1, i]$, w (which is necessarily already in the tree) is read from $S_n(\mathcal{N}_{s,e})$ and σ is added if needed.
   To avoid having to read all the letters of w from $S_n(\mathcal{N}_{s,e})$, the following trick is used. At each branching node met during the reading of w, an edge is chosen depending on the current letter in w. Once the edge is identified, the node pointed by this edge is reached and we advance in the reading of w by the number of letters in the edge. The process is repeated while w is not totally read. Thus the complexity of the reading of w is related to the number of nodes traversed and not to $|w|$.
   If σ is added, a branching node is created. The suffix link of such a node points to the last branching node met during the next insertion (it can be a created one).
   The pseudo-code of this algorithm is given in appendix in Figure 10.

   (b) During phase i, all the suffixes of $t[0, i]$ have to be inserted. Yet if during the insertion of $t[j, i]$, this word is already in the tree, then, by definition, all the words $\{t[k, i], k \in [j, i]\}$ are already in the tree as well. In this case, the i^{th} phase stops here. Similarly, the $(i+1)^{th}$ phase can start inserting $t[j, i+1]$: with the implicit extension of the leaves, the factors $\{t[k, i+1], k \in [1, j-1]\}$ are already in the tree.

A pseudo-code of this construction algorithm is given in appendix in Algorithm A.

Each phase of the algorithm is not done in constant time. However the amortised construction time is linear with respect to the input text length. The demonstration of this complexity is given in [11]. It consists in bounding the overall number of nodes traversed during all insertions.

4.2 Construction algorithm of a k-factor tree

The k-factor tree, also called truncated suffix tree, has been presented in [22] and [1]. A k-factor tree is a suffix tree cut such that each word spelt from the root to a leaf has a length bounded by k. An example of k-factor tree is given in Figure 6. This structure finds applications in various areas such as data compression [22,23] where the indexation is made over a sliding window, or string matching and computational biology [19,27,30] where the length of the motifs searched for in the text is bounded.

The linear time construction algorithm we describe here is based on the Ukkonen suffix tree construction algorithm. For further details on implementation and proof of validity, the reader is referred to [1].

This algorithm is divided in two parts:

1. Build the suffix tree for $t[0, k-2]$.
2. Add in $|t| - k + 1$ phases the suffixes of $t[i-k, i]$ for i from $k-1$ to $|t|-1$.

The first part is achieved using the Ukkonen algorithm. During this part, the leaves created are added to a queue called $queue_{\ell eaf}$.

In the second part, we have to modify the Ukkonen algorithm so that:

- for each phase i, we start by inserting $t[j, i]$ with j not smaller than $i-k+1$;
- implicit leaf extensions are stopped when the length k is reached for the path of a leaf.

To do this last point, we use the queue $queue_{\ell eaf}$.

During the whole construction, each leaf created is added at the end of $queue_{\ell eaf}$. In the second part, for each phase i, there are two possibilities: either $queue_{\ell eaf}$ is empty or not.

Suppose $queue_{\ell eaf}$ contains at least one leaf. Let $\mathcal{L}_{s,e}$ denote a leaf starting position s and ending position e. We then have $queue_{\ell eaf} = \mathcal{L}^1_{s^1,e} \ldots \mathcal{L}^p_{s^p,e}$. We start by fixing the end position of $\mathcal{L}^1_{s^1,e}$ to i, that is $\mathcal{L}^1_{s^1,e}$ becomes $\mathcal{L}^1_{s^1,i}$. Indeed, we know that $path(\mathcal{L}^1_{s^1,i})$ has a length of k.

Suppose we are in phase $i = k-1$. Then $queue_{\ell eaf}$ contains at least one leaf which corresponds to the one created during the insertion of $t[0]$ in the tree (first insertion of the first phase). This leaf is $\mathcal{L}^1_{0,e}$. In phase $i = k$, the leaf is now $\mathcal{L}^1_{0,k-1}$, so its length is equal to k. If there is another leaf in the queue, it corresponds to $\mathcal{L}^1_{1,e}$ and it is clear that its length will be equal to k at the next phase. And so on, as the leaves $\mathcal{L}_{s,e}$ are created with s incremented by one between two leaves.

Once the leaf at the beginning of $queue_{\ell eaf}$ is fixed, we apply again the Ukkonen algorithm from the last leaf in $queue_{\ell eaf}$ (the last created which can be the one we have just fixed). At the end of phase (*i.e.* no leaf created during the last insertion), we remove the leaf at the head of $queue_{\ell eaf}$.

We describe now the case when there is no leaf in the queue. Suppose there were a leaf in the queue at the previous phase $i-1$. By fixing the end value of this leaf, we have fixed the leaf corresponding to $t[i-k, i-1]$. Then we started by inserting $t[i-k+1, i-1]$ in the tree. This insertion did not create a leaf ($queue_{\ell eaf}$ is empty in phase i) and lead to a position p in the tree that corresponds to the spelling of $t[i-k+1, i-1]$. In the current phase i, since $queue_{\ell eaf}$ is empty, we have to start by inserting $t[i-k+1, i]$ in the tree. This can be done in constant time by trying to insert $t[i]$ from the position p. If this insertion creates a leaf, its end value is directly set to i (not added in $queue_{\ell eaf}$) and it is used to try to insert $t[i-k+1, i]$. If no leaf is created, then we continue by trying to insert $t[i-k+1,i]$ from the leaf reached (we know that the insertion of $t[i-k,i]$ leads to a leaf since the path length of the leaves is bounded by k). If the insertion of $t[i-k+1,i]$ does not create a leaf, we use the position reached in the tree to start the next phase.

A pseudo-code of this algorithm is given in appendix in Algorithm A.

The time and space complexities of the algorithm are linear in the size of the input text (see[1] for details).

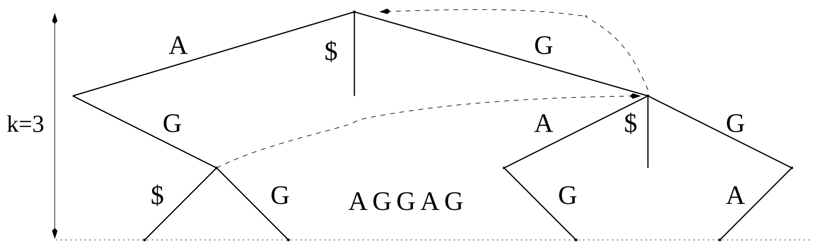


Figure 6. Example of a k-factor tree for the text $AGGAG\$$ with $k=3$

4.3 Gapped-factor tree construction

We now present the construction algorithm of a gapped-factor tree (**gft** for short). Once again, the construction algorithm is done online. As shown in Figure 4, a gft is composed of three different regions: the upper part of depth k, the invisible region corresponding to the gap of length d, and the lower part of depth k':

1. During the construction of the gft, the first region is treated exactly as for a k-factor tree. The queue containing the leaves in extension is denoted by $queue_{leaf\_up}$.
2. When a leaf reaches the depth k, it enters in the invisible region for d phases. To simulate this behaviour, a queue is created that contains the leaves in extension in the invisible region. This queue is denoted by $queue_{invisible}$. Leaves entering $queue_{invisible}$ stay inside for d phases. During those phases, leaves inside the queue are ignored. After d phases, a leaf in the queue is virtually reaching the depth $k+d$. It is then removed from the queue.
3. The construction algorithm of the lower part of the tree is again very similar to the one of a k-factor tree. All the tricks applied for the suffix tree construction are still available. Once more a queue is used to store the leaves in extension in the lower part of the tree. This queue is denoted by $queue_{leaf\_low}$. The ending integer of the leaves in extension in the queue is the global variable i. The leaves stay in the queue during k' phases before they become leaves that stay fixed, and contain the positions of the gapped-factors corresponding to the path leading to them from the root.
   However, for the construction of the lower part, the use made of suffix links is slightly different than in the upper part of the tree. This is due to the following particularity of the gapped-factor tree: a node in the lower part of the tree may have up to $|\Sigma|$ suffix links. Indeed, one node in this tree may correspond to several paths. According to the first letter in the invisible region leading to a node, the suffix link to follow will not be the same. Figure 7 illustrates this observation.

The algorithm 15 given in appendix gives an overview of the whole gapped-factor tree construction algorithm.

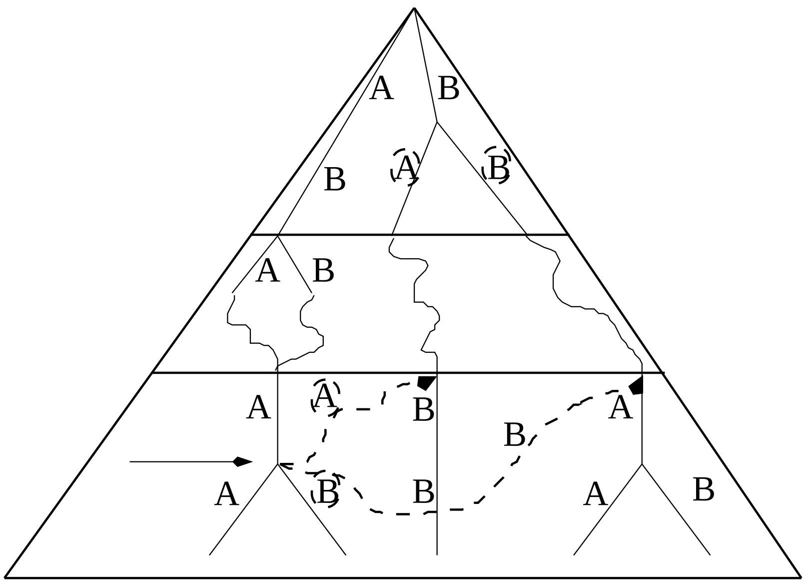


Figure 7. Example of multiple suffix links. The node pointed by an arrow has two suffix links (in dotted line). One is labelled with an A and the other is labelled with a B. The correct suffix link to follow depends on the path that leads to the node. If the node is reached reading $ABA.w$ ($w \in \Sigma^*$), the correct suffix link to follow is the one labelled with an A; it goes to a node reachable reading the text $BA.w$. Any other suffix link leaving the node would be labelled differently and would reach a node corresponding to the text $B.\sigma.w$, with $\sigma \in \Sigma$ and $\sigma \neq A$.

Complexity of the Gft construction The algorithm for constructing a gft uses all the tricks employed by Ukkonen and Allali to lead to a linear time and memory complexity. However, the multiple suffix links add a multiplicative term in $|\Sigma|$ to both complexities. Thus the total time and memory complexity for the construction of a gapped-factor tree for a text t is in $O(|t| \times |\Sigma|)$. One can notice that once the gapped-factor is constructed, the (multiple) suffix links are not useful anymore and can be removed. In this case, the memory complexity falls back to $O(|t|)$.

Generalisation to more than one text As for the suffix tree or the k-factor tree, the gft can be extended to a *generalised gapped-factor* tree and accept a set of $m > 1$ texts $t_0,\ t_1,\ \ldots,\ t_{m-1}$.

In this case, each text $i \in [0, m-1]$ ends with a special character $\$_i$ and the leaves are labelled not only with the positions of a gapped-factor but also with the sequence number in $[0, m-1]$ where the factors occur. The complexity for constructing a generalised gapped-factor tree is in $O\left(\left(\sum_{i=0}^{m-1} |t_i|\right) \times |\Sigma|\right)$.

5 Basic uses of a gapped-factor tree

To find all the positions where a (k-d-k')-gapped-factor occurs in a text given a (k-d-k')-gapped-factor tree for the text one needs to find the leaf corresponding to the given gapped-factor. This is done straightforwardly by traversing the gapped-factor tree from the root to the node as in a suffix tree. The list attached to the leaf corresponds to the positions of the occurrences of the gapped-factors.

This algorithm takes a time proportional to the number of nodes traversed, which is in the worst case $k + k'$. Thus retrieving the positions of a given (k-d-k')-gapped-factor is done in $O(k + k')$.

The gft data structure allows also to easily find all the repeated gapped-factors of a text or of a set of texts. If we are interested in finding all gapped-factors occurring at least r times in a text, for r a positive integer, we just have to visit the leaves. For each leaf, if the number of elements of the list attached to it is greater or equal to r, the corresponding gapped-factor is considered as repeated.

As the number of elements of each list may be stored in the leaves, this extraction is done in time proportional to the number of leaves. If n denotes the length of the indexed text, the number of leaves is no greater than n. The extraction is therefore done in time $O(n)$.

In the generalised case, one may want to extract all gapped-factors occurring in at least r different texts. In this case, to each leaf is attached the number of different texts in which the corresponding gapped-factor occurs. Thus extracting all gapped-factors occurring in at least r different texts is done by checking each leaf in constant time leading to a complexity in $O(\sum_{i=1}^{m} |t_i|)$.

6 Conclusion

We presented a new data structure used for indexing factors containing a gap (called the gapped-factors). This data structure is based on the suffix tree structure. Furthermore, we indicated an online construction algorithm of this data structure for a text t on an alphabet Σ in $O(|t| \times |\Sigma|)$ time and space. This algorithm is based on the Ukkonen algorithm for constructing a suffix tree.

References

1. J. Allali and M. Sagot: *The at most k-deep factor tree*, Tech. Rep. #2004-03, Institut Gaspard Monge, Université de Marne-la-Vallée, 2004.
2. S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman: *Basic local alignment search tool*. Journal of Molecular Biology, 215(3) 1990, pp. 403–410.
3. S. Altschul, T. Madden, A. Schaffer, J. Zhang, Z. Zhang, W. Miller, and D. Lipman: *Gapped BLAST and PSI–BLAST: a new generation of protein database search programs*. Nucleic Acids Research, 25 1997, pp. 3389–3402.
4. M. Brudno, M. Chapman, B. Göttgens, S. Batzoglou, and B. Morgenstern: *Fast and sensitive multiple alignment of large genomic sequences*. BMC Bioinformatics, 4 2003, p. 66.
5. M. Brudno, C. B. Do, G. M. Cooper, M. Kim, E. Davydov, E. D. Green, A. Sidow, and S. Batzoglou: *LAGAN and Multi-LAGAN: Efficient tools for large-scale multiple alignment of genomic DNA*. Genome Research, 13 2003, pp. 721–731.
6. S. Burkhardt, A. Crauser, P. Ferragina, H. P. Lenhof, and M. Vingron: *q-gram based database searching using a suffix array (QUASAR)*. Proceedings of the third annual international conference on Computational molecular biology (Recomb 99), 1999, pp. 77–83.
7. S. Burkhardt and J. Kärkkäinen: *Better filtering with gapped q-grams*, in Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching (CPM 2001), vol. 2089 of LNCS, 2001, pp. 73–85.

8. S. Burkhardt and J. Kärkkäinen: *One-gapped q-gram filters for Levenshtein distance*. 13th Annual Symposium on Combinatorial Pattern Matching (CPM 2002), 2373 of LNCS 2002, pp. 225–234.
9. R. C. Edgar: *MUSCLE: Multiple sequence alignment with high accuracy and high throughput*. Nucleic Acids Research, 32(5) 2004, pp. 1792–1797.
10. L. Gravano, P. Ipeirotis, H. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava: *Approximate string joins in a database (almost) for free*, in In Proc. of 27th Int'l Conf. on Very Large DataBases (VLDB 2001), 2001, pp. 491–500.
11. D. Gusfield: *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*, Cambridge University Press, 1997.
12. M. Höhl, S. Kurtz, and E. Ohlebusch: *Efficient multiple genome alignment*. ISMB (Supplement of Bioinformatics), Vol. 18 2002, pp. S312–S320.
13. C. S. Iliopoulos, J. Mchugh, P. Peterlongo, N. Pisanti, W. Rytter, and M.-F. Sagot: *A first approach to finding common motifs with gaps*. International Journal of Foundations of Computer Science, 16(6) 2005, pp. 1145–1154.
14. J. Kärkkäinen: *Computing the threshold for q-gram filters*. Proceedings of the 8th Scandinavian Workshop on Algorithm Theory (SWAT 2002), 2368 of LNCS 2002, pp. 348–357.
15. G. Kucherov, L. Noe, and M. Roytberg: *Multiseed lossless filtration*. IEEE/ACM Transactions on Computational Biology and Bioinformatics, 02 2005, pp. 51–61.
16. M. Li and B. Ma: *PatternHunter II: Highly sensitive and fast homology search*. Genome Informatics, 14 2003, pp. 164–175.
17. D. J. Lipman and W. R. Pearson: *Rapid and sensitive protein similarity searches*. Science, 227 1985, pp. 1435–1441.
18. B. Ma, J. Tromp, and M. Li: *PatternHunter: Faster and more sensitive homology search*. Bioinformatics, 18(3) 2002, pp. 440–445.
19. L. Marsan and M.-F. Sagot: *Extracting structured motifs using a suffix tree - algorithms and application to promoter consensus identification*, in RECOMB, 2000, pp. 210–219.
20. E. M. McCreight: *A space-economical suffix tree construction algorithm*. Journal of the Association of Computing Machinery, 23(2) 1976, pp. 262–272.
21. M. Michael, C. Dieterich, and M. Vingron: *Siteblast rapid and sensitive local alignment of genomic sequences employing motif anchors*. Bioinformatics, 21(9) 2005, pp. 2093–2094.
22. J. Na, A. Apostolico, C. Iliopoulos, and K. Park: *Truncated suffix trees and their application to data compression*. Theor. Comput. Sci., 304(1-3) 2003, pp. 87–101.
23. J. Na and K. Park: *Data compression with truncated suffix trees*. Data Compression Conference (DCC 2000), IEEE Computer Society, online edition: http://computer.org/proceedings/dcc/0592/0592toc.htm 2000, p. 565.
24. G. Navarro, E. Sutinen, J. Tanninen, and J. Tarhio: *Indexing text with approximate q-grams*. 11th Annual Symposium on Combinatorial Pattern Matching (CPM 2000), 1848 of LNCS 2000, pp. 350–363.
25. P. Peterlongo, N. Pisanti, F. Boyer, and M.-F. Sagot: *Lossless filter for finding long multiple approximate repetitions using a new data structure, the bi-factor array*. String Processing and Information Retrieval (SPIRE 2005), 3772 of LNCS 2005, pp. 179–190.
26. P. Pevzner and M. Waterman: *Multiple filtration and approximate pattern matching*. Algorithmica, 13 1995, pp. 135–154.
27. N. Pisanti, A. Carvalho, L. Marsan, and M.-F. Sagot: *RISOTTO: Fast extraction of motifs with mismatches*, in Proceedings of the 7th Latin American Theoretical Informatics Symposium, vol. 3887 of LNCS, 2006, pp. 757–768.
28. K. R. Rasmussen, J. Stoye, and E. W. Myers: *Efficient q-gram filters for finding all ε-matches over a given length*, in 9th Annual International Conference, Research in Computational Molecular Biology (Recomb 2005), vol. 3678 of LNCS, 2005, pp. 189–203.
29. E. Sutinen and J. Tarhio: *On using q-gram locations in approximate string matching*, in Third Annual European Symposium, (ESA 95), vol. 979 of LNCS, 1995, pp. 327–340.
30. P. Thébault, S. DeGivry, T. Schiex, and C. Gaspin: *Combining constraint processing and pattern matching to describe and locate structured motifs in genomic sequences*, in Fifth IJCAI-05 Workshop on Modelling and Solving Problems with Constraints, Edindurgh, Scotland, 2005, pp. 330–337.
31. E. Ukkonen: *On-line construction of suffix-trees*. Algorithmica, 14 1995, pp. 249–260.

A Pseudo-codes

Algorithm 10 Fast Insertion

Require: $\mathcal{N}$,t,$start$,end

Ensure: Insert a string $t_{start...end}$ from a node $\mathcal{N}$ assuming that the tree is already constructed for $t_{start...end-1}$ from $\mathcal{N}$

```
endJump ← false
while (not endJump) and ((end − start) ≠ 0) do
    set child to the child of N that starts with the letter t_start
    if (end − start) ≥ length(N, child) then
        start ← start + length(N, child)
        N ← child
    else
        endJump ← true
    end if
end while
if (end − start) = 0 and N has not a child for letter t_end then
    add a child to N with edge label start equal to end
end if
e ←the label of the edge between N and child
if e_(end−start+1) ≠ s_end then
    split e at position end − start
    add a leaf with start position equal to end to the new node
end if
```

Algorithm 11 Factor Tree

Require: R, t , k , $queue_{leaf}$

Ensure: The k-factor tree of t

```
do the first k−1 phases using Suffix_Tree algorithm, filling queue_leaf with each new leaf created
for i from k to |t| do
    if queue_leaf is not empty then
        set lastLeaf to the leaf at the end of queue_leaf
    else
        add t_i from last position reached during the last insertion
        if a leaf is created then
            add this leaf at the end of queue_leaf
            set lastLeaf to this leaf
        else
            set lastLeaf to the leaf reached
        end if
    end if
    Phase (R, t , k, i, queue_leaf, lastLeaf)
    remove the leaf at the head of queue_leaf and set its end value to i
end for
return R
```

Algorithm 12 Function **Phase** (Suffix tree and k-factor tree construction)

Require: R, t, k, i, $\underline{queue_{leaf}}$, $lastLeaf$
Ensure: One phase of the construction of the suffix tree and of the k-factor tree. The underlined parts stand only for the k-factor tree construction.

$endPhase \leftarrow false$
repeat
  $forward \leftarrow length(Father(lastLeaf), lastLeaf) - 1$
  if $S_l(Father(lastLeaf))$ is undefined and $Father(lastLeaf)! = R$ **then**
    $forward \leftarrow forward + length(Father(Father(lastLeaf)), Father(lastLeaf))$
    if $Father(Father(lastLeaf))$ is R **then**
      AddString(R,t,$i - forward + 1$,i)
    else
      AddString($S_l(Father(Father(lastLeaf)))$,$t$,$i - forward$,$i$)
    end if
  else
    if $Father(lastLeaf)$ is R **then**
      AddString(R,t,$i - forward + 1$,i)
    else
      AddString($S_l(Father(lastLeaf))$,t,$i - forward$,i)
    end if
  end if
  if a node was created during the previous step **then**
    set the suffix link of this node to the last node reached during the insertion
  end if
  if a leaf was created in the call to AddString **then**
    set $lastLeaf$ to this leaf
    $\underline{\text{add this leaf at the end of } queue_{leaf}}$
  end if
  if no node was created during the call to AddString **then**
    $endPhase \leftarrow true$
  end if
until not $endPhase$

Algorithm 13 Suffix Tree

Require: t
Ensure: The suffix tree of t

Add to R a leaf L with edge label t_0
$lastLeaf \leftarrow L$
for i **from** 1 **to** $|t| - 1$ **do**
  Phase (R, t , k, i, $lastLeaf$)
end for
return R

Algorithm 14 Lower_Part_Tree

Require: R, t , k , d, i, $queue_{leaf\_low}$, $lastLeafLow$
Ensure: A construction phase of the lower part of the gapped-factor tree

$endPhaseLow \leftarrow false$
repeat
  $forward \leftarrow length(Father(lastLeafLow), lastLeafLow) - 1$
  if $S_l(t, Father(lastLeafLow))$ is defined but not labeled with the good character **then**
    Point the $Father(lastLeafLow)$ node as the node created during the previous step
  end if
  if $S_l(t_\alpha, Father(lastLeafLow))$ is undefined and $Father(lastLeafLow)! = R$ **then**
    $forward \leftarrow forward + length(Father(Father(lastLeafLow)), Father(lastLeafLow))$
    if $Father(Father(lastLeafLow))$ is R **then**
      $AddString(R,t,i - forward + 1,i)$
    else
      AddString($S_l(t_\alpha, F(F(lastLeafLow)))t$,$i - forward$,$i$)
    end if
  else
    if $Father(lastLeafLow)$ is R **then**
      $AddString(R,t,i - forward + 1,i)$
    else
      $AddString(S_l(t_\alpha, F(lastLeafLow),t,i - forward,i)$
    end if
  end if
  if a node was created during the previous step **then**
    set the suffix link labeled t_α of this node to the last node reached during the insertion
  end if
  if a leaf was created during the call to AddString **then**
    set $lastLeafLow$ to this leaf
    add this leaf at the end of $queue_{leaf\_low}$
  end if
  if no node was created in the call to AddString **then**
    $endPhaseLow \leftarrow true$
  end if
  if the width of the last position reached during the fast insertion was $\leq k + d$ **then**
    $endPhaseLow \leftarrow true$
  end if
until not $endPhaseLow$

NOTE : α is the first character in the invisible region on the $lastLeafLow$ path

Algorithm 15 GappedFactor_Tree

Require: R, t , k , d, $queue_{leaf\_up}$, $queue_{leaf\_low}$, $queue_{invisible}$

Ensure: Complete construction algorithm of a gapped factor tee

```
do the first k phases using Suffix_Tree algorithm, filling queue_leaf_up with each new leaf created
for i from k to |t| do
  if index of node at the head of queue_invisible = k + 1 then
    create a new edge from this node labeled t_i
    add the new leaf at the end of queue_leaf_low
    remove the node at head of queue_invisible
  end if
  if queue_leaf_up is not empty then
    set lastLeafUp to the leaf at the end of queue_leaf_up
  else
    add t_i from last position reached during the last insertion on the upper part of the tree
    if a leaf is created then
      add this leaf at the end of queue_leaf_up
      set lastLeafUp to this leaf
    else
      set lastLeafUp to the leaf reached
    end if
  end if
  Phase(R, t, k, i, queue_leaf_up, lastLeafUp)
  remove the pseudo leaf at the head of queue_leaf_up
  if the pseudo leaf is new then
    set the pseudo leaf index value to 0 (invisible zone)
    add the pseudo leaf at the end of queue_invisible
  end if
  if i > k + d //the lower part of the tree is on construction then
    if queue_leaf_low is not empty then
      set lastLeafLow to the leaf at the end of queue_leaf_low
    else
      add t_i from last position reached during the last insertion on the low part of the tree
      if a leaf is created then
        add this leaf at the end of queue_leaf_low
        set lastLeafLow to this leaf
      else
        set lastLeafLow to the leaf reached
      end if
    end if
    Lower_Part_Tree(R, t, k, d, i, queue_leaf_low, lastLeafLow)
    if i ≥ k + d + k // End the extention of the tree then
      remove the leaf at the head of queue_leaf_low
      set the leaf end value to i
    end if
  end if
end for
return R
```

Sparse Compact Directed Acyclic Word Graphs

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

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

Abstract. The suffix tree of string w represents all suffixes of w, and thus it supports full indexing of w for exact pattern matching. On the other hand, a *sparse suffix tree* of w represents only a subset of the suffixes of w, and therefore it supports sparse indexing of w. There has been a wide range of applications of sparse suffix trees, e.g., natural language processing and biological sequence analysis. *Word suffix trees* are a variant of sparse suffix trees that are defined for strings that contain a special word delimiter #. Namely, the word suffix tree of string $w = w_1 w_2 \cdots w_k$, consisting of k words each ending with #, represents only the k suffixes of w of the form $w_i \cdots w_k$. Recently, we presented an algorithm which builds word suffix trees in $O(n)$ time with $O(k)$ space, where n is the length of w. In addition, we proposed *sparse directed acyclic word graphs* (*SDAWGs*) and an on-line algorithm for constructing them, working in $O(n)$ time and space. As a further achievement of this research direction, this paper introduces yet a new text indexing structure named *sparse compact directed acyclic word graphs* (*SCDAWGs*). We show that the size of SCDAWGs is smaller than that of word suffix trees and SDAWGs, and present an SCDAWG construction algorithm that works in $O(n)$ time with $O(k)$ space and in an on-line manner.

1 Introduction

Suffix trees have played a very central role in combinatorial pattern matching as they have wide applications such as data compression [10,12,6] and bioinformatics [11,2,5]. Suffix trees are fairly useful since they can be constructed in linear time and space in the input string length [14]. On the other hand, there have been great demands to deal with the common case that only certain suffixes of the input string are relevant. Suffix trees that contain only a subset of all suffixes are called *sparse suffix trees*. Among several versions of sparse suffix trees, we in this paper concentrate on *word suffix trees* introduced in [1].

Let D be a dictionary of words and w be a string in D^+ of length n, namely, w be a sequence $w_1 \cdots w_k$ of k words in D. The word suffix tree of w w.r.t. D is a tree structure which represents only the k suffixes of w in the form $w_i \cdots w_k$. Although the normal suffix tree of w requires $O(n)$ space, the word suffix tree of w w.r.t. D needs only $O(k)$ space. One typical application of word suffix trees is a word- and phrase-level index for documents written in a natural language. Note that normal suffix trees report *all* occurrences of a keyword in the text string, which may cause unwanted matchings (e.g., an occurrence of "other" in "mother" is possibly retrieved). Andersson et al. introduced an algorithm to build the word suffix tree for w w.r.t. D with $O(k)$ space, but in $O(n)$ *expected* time [1]. Lately, we invented a faster algorithm that constructs word suffix trees with $O(k)$ space and in $O(n)$ time *in the worst case* [8]. This is optimal, since the whole string w needs to be read at least once.

It is noteworthy that our word suffix tree construction algorithm gives a practical solution to linear-time construction of sparse suffix trees for arbitrary subsets of suffixes. Given a set S of $k-1$ positions in string w, we insert a unique word delimiter # into w at the positions listed in S. Now we get a string which consists of k words, each separated by #. The word suffix tree of this modified string is alternative to the sparse suffix tree of w w.r.t. S. In the matching phase, we simply ignore any #'s in the edge labels of the tree.

In this paper, we introduce a new data structure named *sparse compact directed acyclic word graphs* (*SCDAWGs*) as an alternative to the word suffix trees and to the sparse suffix trees as well. SCDAWGs are a sparse text indexing version of *compact directed acyclic word graphs* (*CDAWGs*) of [4]. We define SCDAWGs based on 'word-position-sensitive' equivalence relations on string w and dictionary D, and show the asymptotic size of SCDAWGs to be $O(k)$. Moreover, the fact is that SCDAWGs are a minimization of sparse suffix trees, and therefore require no more space than sparse suffix trees. Finally, we present an on-line algorithm for building SCDAWGs, which is based on the on-line algorithm for building normal CDAWGs in [7]. By using the minimum DFA M_D which accepts D, and by tailoring suffix links accordingly, the modified algorithm constructs SCDAWGs. Since our algorithm directly constructs SCDAWGs (namely, not constructing sparse suffix trees as an intermediate), it works with space linear in the output size. We also show that the proposed algorithm runs in $O(n)$ time. Furthermore, our algorithm can be seen as a generalization of the normal CDAWG construction algorithm of [7]. Assume just for now $D = \Sigma$, and consider a DFA which accepts Σ with only two states that are a single initial state and a single final state. Then this DFA plays the same role as the auxiliary '$\perp$' node used in [7], and as a result normal CDAWGs are generated.

2 Preliminaries

2.1 Notations

Let Σ be a finite set of symbols, called an *alphabet*. Throughout this paper we assume that Σ is fixed. A finite sequence of symbols is called a *string*. We denote the length of string w by $|w|$. The empty string is denoted by ε, that is, $|\varepsilon| = 0$. Let Σ^* be the set of strings over Σ. For any symbol $a \in \Sigma$, we define a^{-1} such that $a^{-1}a = \varepsilon$.

Strings x, y, and z are said to be a *prefix*, *substring*, and *suffix* of string $w = xyz$, respectively. A prefix, substring, and suffix of string w are said to be *proper* if they are not w. Let $\mathit{Prefix}(w)$ and $\mathit{Suffix}(w)$ be the set of the prefixes and suffixes of string w, respectively. For set S of strings, let $\mathit{Prefix}(S) = \bigcup_{w \in S} \mathit{Prefix}(w)$.

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

The i-th symbol of string w is denoted by $w[i]$ for $1 \le i \le |w|$, and the substring of string w that begins at position i and ends at position j is denoted by $w[i..j]$ for $1 \le i \le j \le |w|$. For convenience, let $w[i..j] = \varepsilon$ for $i > j$. For any strings $x, w \in \Sigma^*$, let

$$\mathit{Begpos}_w(x) = \{i \mid x = w[i..i+|x|-1]\}, \text{ and}$$
$$\mathit{Endpos}_w(x) = \{j \mid x = w[j-|x|+1..j]\}.$$

Let D be a set of strings called a *dictionary*. A *factorization* of string w w.r.t. D is a list $w_1, \ldots, w_k$ of strings in D such that $w = w_1 \cdots w_k$ and $w_i \in D$ for each

$1 \leq i \leq k$. In the rest of the paper, we assume that $D = \Sigma^*\#$ where $\#$ is a special symbol not belonging to Σ, and that $w \in D^+$. Then, a factorization of w w.r.t. D is always unique, since D clearly satisfies the prefix property because of $\#$ not being in Σ.

For any string $w = w_1 \cdots w_k \in D^+$, let u be any prefix of w. Then we can write as $u = w_1 \cdots w_\ell v$ with $1 \leq \ell < k$, where v is a prefix of $w_{\ell+1}$. For any $1 \leq i \leq \ell$, let $u_i = w_i \cdots w_\ell v$, and for convenience, let $u_{\ell+1} = v$ and $u_{\ell+2} = \varepsilon$. Now, we define a word-oriented subset $\mathit{Suffix}_D(u)$ of $\mathit{Suffix}(u)$ as follows:

$$\mathit{Suffix}_D(u) = \{u_i \mid 1 \leq i \leq \ell + 2\}.$$

Namely, $\mathit{Suffix}_D(u)$ consists only of u, the suffixes of u which immediately follow any $\#$ in u, and the empty string ε.

Example 2. Let $\Sigma = \{\mathtt{a}, \mathtt{b}\}$, $D = \Sigma^*\#$, $w = \mathtt{ab\#b\#aa\#}$, and $u = \mathtt{ab\#b\#a}$. Then $\mathit{Suffix}_D(u) = \{\mathtt{ab\#b\#a}, \mathtt{b\#a}, \mathtt{a}, \varepsilon\}$.

Further, we define set $\mathit{Wordpos}_D(u)$ of the word-starting positions in u, as follows:

$$\mathit{Wordpos}_D(u) = \{|u| - |s| + 1 \mid s \in \mathit{Suffix}_D(u) - \{\varepsilon\}\}.$$

Example 3. For the running string $u = \mathtt{ab\#b\#a}$, $\mathit{Wordpos}_D(u) = \{1, 4, 6\}$.

2.2 Equivalence Classes on Strings over D

For set S of integers and integer i, we denote $S \oplus i = \{j + i \mid j \in S\}$ and $S \ominus i = \{j - i \mid j \in S\}$. Let $w \in D^+$. For any $u \in \mathit{Prefix}(\mathit{Suffix}_D(w))$ and $x, y \in (\Sigma \cup \{\#\})^*$, we define the begin- and end-equivalence relations $\equiv_u^B$ and $\equiv_u^E$, as follow:

$$\begin{aligned} x \equiv_u^B y &\Leftrightarrow \mathit{Begpos}_u(x) \cap \mathit{Wordpos}_D(u) = \mathit{Begpos}_u(y) \cap \mathit{Wordpos}_D(u), \\ x \equiv_u^E y &\Leftrightarrow \mathit{Endpos}_u(x) \cap (\mathit{Wordpos}_D(u) \oplus |x| \ominus 1) \\ &\qquad = \mathit{Endpos}_u(y) \cap (\mathit{Wordpos}_D(u) \oplus |y| \ominus 1). \end{aligned}$$

We note that the above equivalence relations are 'word-position-sensitive' versions of the equivalence relations introduced in [3], where the intersections with $\mathit{Wordpos}_D(u)$ make them word-position-sensitive. We denote by $[x]_u^B$ and $[x]_u^E$ the equivalence classes of x w.r.t. $\equiv_u^B$ and $\equiv_u^E$, respectively.

Proposition 4. *All strings that are not in $\mathit{Prefix}(\mathit{Suffix}_D(u))$ form one equivalence class under $\equiv_u^B$ (and $\equiv_u^E$), called the* degenerate class.

Proof. For any $x \notin \mathit{Prefix}(\mathit{Suffix}_D(u))$, clearly $\mathit{Wordpos}_D(u) = \emptyset$. Thus $\mathit{Begpos}_u(x) \cap \mathit{Wordpos}_D(u) = \emptyset$. Hence, any strings not belonging to $\mathit{Prefix}(\mathit{Suffix}_D(u))$ form one equivalence class. Similar discussion holds for $\equiv_u^E$. □

Proposition 5. *For any strings $x, y \in \mathit{Prefix}(\mathit{Suffix}_D(u))$, if $x \equiv_u^B y$, then either $x \in \mathit{Prefix}(y)$, or vice versa.*

Proof. Assume, without loss of generality, that $|x| \leq |y|$. Since $x \equiv_u^B y$, $\mathit{Begpos}_u(x) \cap \mathit{Wordpos}_D(u) = \mathit{Begpos}_u(y) \cap \mathit{Wordpos}_D(u)$. Let S be this set of positions. For any $i \in S$, we have that $x = u[i..i + |x| - 1]$ and $y = u[i..i + |y| - 1]$. Since $|x| \leq |y|$, $x \in \mathit{Prefix}(y)$. □

Example 6. For the running string $u = \mathtt{ab\#b\#a}$, $[\mathtt{b\#a}]_u^B = \{\mathtt{b\#a}, \mathtt{b\#}, \mathtt{b}\}$. For any pair of strings $x, y \in [\mathtt{b\#a}]_u^B$, we have $x \in \mathit{Prefix}(y)$ or $y \in \mathit{Prefix}(x)$.

Proposition 7. *For any strings $x, y \in \mathit{Prefix}(\mathit{Suffix}_D(u))$, if $x \equiv_u^E y$, then either $x \in \mathit{Suffix}_D(y)$, or vice versa.*

Proof. Assume, without loss of generality, that $|x| \leq |y|$. Since $x \equiv_u^E y$, $\mathit{Endpos}_u(x) \cap (\mathit{Wordpos}_D(u) \oplus |x| \ominus 1) = \mathit{Endpos}_u(y) \cap (\mathit{Wordpos}_D(u) \oplus |y| \ominus 1)$. Let S be this set of positions. For any $i \in S$, we have that $x = u[i-|x|+1..i]$ and $y = u[i-|y|+1..i]$. Since $x \in \mathit{Prefix}(u[i-|x|+1..|u|])$, $u[i-|x|+1..|w|] \in \mathit{Suffix}_D(u)$, and $\mathit{Wordpos}_D(u) = \{|u| - |s| + 1 \mid s \in \mathit{Suffix}_D(u) - \{\varepsilon\}\}$, we have $i - |x| + 1 \in \mathit{Wordpos}_D(u)$. Similarly, $i - |y| + 1 \in \mathit{Wordpos}_D(u)$. Since $|x| \leq |y|$, we have $x \in \mathit{Suffix}_D(y)$. $\square$

Example 8. For the running string $u = \mathtt{ab\#b\#a}$, $[\mathtt{ab\#b}]_u^E = \{\mathtt{ab\#b}, \mathtt{b}\}$. Then $\mathtt{b} \in \mathit{Suffix}_D(\mathtt{ab\#b})$.

From Propositions 5 and 7, each non-degenerate equivalence class under $\equiv_u^B$ or $\equiv_u^E$ has a unique longest member, which is called the *representative* of the equivalence class. For any $x \in \mathit{Prefix}(\mathit{Suffix}_D(u))$, the representatives of $[x]_u^E$ and $[x]_u^B$ are denoted by $\overset{u}{\overleftarrow{x}}$ and $\overset{u}{\overrightarrow{x}}$, respectively.

For any $x \in \mathit{Prefix}(\mathit{Suffix}_D(u))$ such that $\overset{u}{\overrightarrow{x}} = x\alpha$ and $\overset{u}{\overleftarrow{x}} = \beta x$ with $\alpha, \beta \in (\Sigma \cup \{\#\})^*$, we denote $\overset{u}{\overleftrightarrow{x}} = \beta x \alpha$.

Proposition 9. *For any $x \in \mathit{Prefix}(\mathit{Suffix}_D(u))$, $\overset{u}{\overleftrightarrow{x}} = \overset{u}{\overleftarrow{(\overset{u}{\overrightarrow{x}})}} = \overset{u}{\overrightarrow{(\overset{u}{\overleftarrow{x}})}}$.*

Proof. Let $\overset{u}{\overrightarrow{x}} = x\alpha$ and $\overset{u}{\overleftarrow{x}} = \beta x$ with $\alpha, \beta \in (\Sigma \cup \{\#\})^*$. Then, $\overset{u}{\overleftrightarrow{x}} = \beta x \alpha$. Since $\overset{u}{\overrightarrow{x}} = x\alpha$, we have

$$\begin{aligned}
x \equiv_u^B x\alpha &\Leftrightarrow \mathit{Begpos}_u(x) \cap \mathit{Wordpos}_D(u) = \mathit{Begpos}_u(x\alpha) \cap \mathit{Wordpos}_D(u) \\
&\Leftrightarrow (\mathit{Begpos}_u(x) \oplus |x\alpha| \ominus 1) \cap (\mathit{Wordpos}_D(u) \oplus |x\alpha| \ominus 1) \\
&\qquad = (\mathit{Begpos}_u(x\alpha) \oplus |x\alpha| \ominus 1) \cap (\mathit{Wordpos}_D(u) \oplus |x\alpha| \ominus 1) \\
&\Leftrightarrow (\mathit{Endpos}_u(x) \oplus |\alpha|) \cap (\mathit{Wordpos}_D(u) \oplus |x\alpha| \ominus 1) \\
&\qquad = \mathit{Endpos}_u(x\alpha) \cap (\mathit{Wordpos}_D(u) \oplus |x\alpha| \ominus 1).
\end{aligned} \tag{1}$$

On the other hand, since $\overset{u}{\overleftarrow{x}} = \beta x$, we have

$$\begin{aligned}
x \equiv_u^E \beta x &\Leftrightarrow \mathit{Endpos}_u(x) \cap (\mathit{Wordpos}_D(u) \oplus |x| \ominus 1) \\
&\qquad = \mathit{Endpos}_u(\beta x) \cap (\mathit{Wordpos}_D(u) \oplus |\beta x| \ominus 1) \\
&\Leftrightarrow (\mathit{Endpos}_u(x) \oplus |\alpha|) \cap (\mathit{Wordpos}_D(u) \oplus |x| \oplus |\alpha| \ominus 1) \\
&\qquad = (\mathit{Endpos}_u(\beta x) \oplus |\alpha|) \cap (\mathit{Wordpos}_D(u) \oplus |\beta x| \oplus |\alpha| \ominus 1) \\
&\Leftrightarrow (\mathit{Endpos}_u(x) \oplus |\alpha|) \cap (\mathit{Wordpos}_D(u) \oplus |x\alpha| \ominus 1) \\
&\qquad = (\mathit{Endpos}_u(\beta x) \oplus |\alpha|) \cap (\mathit{Wordpos}_D(u) \oplus |\beta x\alpha| \ominus 1).
\end{aligned} \tag{2}$$

Let

$$\begin{aligned}
A &= (\mathit{Endpos}_u(\beta x\alpha)) \cap (\mathit{Wordpos}_D(u) \oplus |\beta x\alpha| \ominus 1), \text{ and} \\
B &= (\mathit{Endpos}_u(\beta x) \oplus |\alpha|) \cap (\mathit{Wordpos}_D(u) \oplus |\beta x\alpha| \ominus 1)
\end{aligned}$$

We show $A = B$. Since $\mathit{Endpos}_u(\beta x\alpha) \subseteq (\mathit{Endpos}_u(\beta x) \oplus |\alpha|)$, it is clear that $A \subseteq B$. For any $i \in B$, let $k = i - |x| - |\alpha| + 1$. Then $u[k..k + |x| - 1] = x$ and thus $k \in \mathit{Begpos}_u(x)$. Let $j = i - |\beta x| - |\alpha| + 1$. Then $u[j..j + |\beta x| - 1] = \beta x$. We have $j \in \mathit{Wordpos}_D(u)$ since $i \in B$. By Proposition 7 we have $x \in \mathit{Suffix}_D(\beta x)$, and therefore $\beta x[|\beta|] = \beta[|\beta|] = u[j + |\beta| - 1] = \#$. Hence $j + |\beta| \in \mathit{Wordpos}_D(u)$. On the other hand, $j + |\beta| = i - |\beta x| - |\alpha| + 1 + |\beta| = k$, thus $k \in \mathit{Wordpos}_D(u)$. Since $u[k..k + |x| - 1] = x$ and $\overset{u}{\overrightarrow{x}} = x\alpha$, $u[k..k + |x\alpha| - 1] = x\alpha$. Now we get

$$
\begin{aligned}
u[j..j + |\beta x\alpha| - 1] &= u[j..j + |\beta| - 1]u[j + |\beta|..j + |\beta x\alpha| - 1] \\
&= u[j..j + |\beta| - 1]u[k..k + |x\alpha| - 1] \\
&= \beta x\alpha.
\end{aligned}
$$

Thus $i = j + |\beta x\alpha| - 1 \in A$, and we obtain $B \subseteq A$.

From Equations (1) and (2), and $A = B$, we get $x\alpha \equiv_u^E \beta x\alpha$. It is easy to see that $\beta x\alpha$ is the representative of $[x\alpha]_u^E$. Finally, $\overset{u}{\overleftarrow{(\overset{u}{\overrightarrow{x}})}} = \overset{u}{\overleftarrow{x\alpha}} = \beta x\alpha = \overset{u}{\overleftrightarrow{x}}$. Similarly we can show $\overset{u}{\overrightarrow{(\overset{u}{\overleftarrow{x}})}} = \overset{u}{\overleftrightarrow{x}}$. $\square$

3 Sparse Compact Directed Acyclic Word Graphs

In this section we introduce our new text indexing structure, *sparse compact directed acyclic word graphs* (*SCDAWGs*).

3.1 Definitions and Size Bounds

We first give a formal definition of sparse (word) suffix trees.

Definition 10 (Sparse suffix tree). *The* sparse suffix tree *of string $w \in D^+$, denoted by $\mathit{SSTree}_D(w)$, is a tree (V, E) such that*

$$
\begin{aligned}
V &= \{\overset{w}{\overrightarrow{x}} \mid x \in \mathit{Prefix}(\mathit{Suffix}_D(w))\}, \\
E &= \left\{ (\overset{w}{\overrightarrow{x}}, a\beta, \overset{w}{\overrightarrow{xa}}) \;\middle|\; \begin{array}{l} x, xa \in \mathit{Prefix}(\mathit{Suffix}_D(w)),\ a \in \Sigma \cup \{\#\}, \\ \beta \in (\Sigma \cup \{\#\})^*, \text{ and } \overset{w}{\overrightarrow{x}} a\beta = \overset{w}{\overrightarrow{xa}} \end{array} \right\}.
\end{aligned}
$$

Theorem 11 ([1]). *For any string $w = w_1 \cdots w_k \in D^+$, $\mathit{SSTree}_D(w)$ has $O(k)$ nodes and edges.*

To prove the above theorem, it suffices to show the two following claims:

Claim. $\mathit{SSTree}_D(w)$ has at most k leaves.

Proof. Recall that $w = w_1 \cdots w_k$ and that $|\mathit{Suffix}_D(x) - \{\varepsilon\}| = k$. For any $x \in \mathit{Prefix}(\mathit{Suffix}_D(w)) - (\mathit{Suffix}_D(w) - \{\varepsilon\})$, it follows from Definition 10 that there exist a symbol $a \in \Sigma \cup \{\#\}$ and a string $\beta \in (\Sigma \cup \{\#\})^*$ satisfying $\overset{w}{\overrightarrow{x}} a\beta = \overset{w}{\overrightarrow{xa}} \in \mathit{Prefix}(\mathit{Suffix}_D(w))$. Thus there can be at most k leaves in $\mathit{SSTree}_D(w)$. $\square$

Claim. All internal nodes of $\mathit{SSTree}_D(w)$ are branching (of out-degree at least 2).

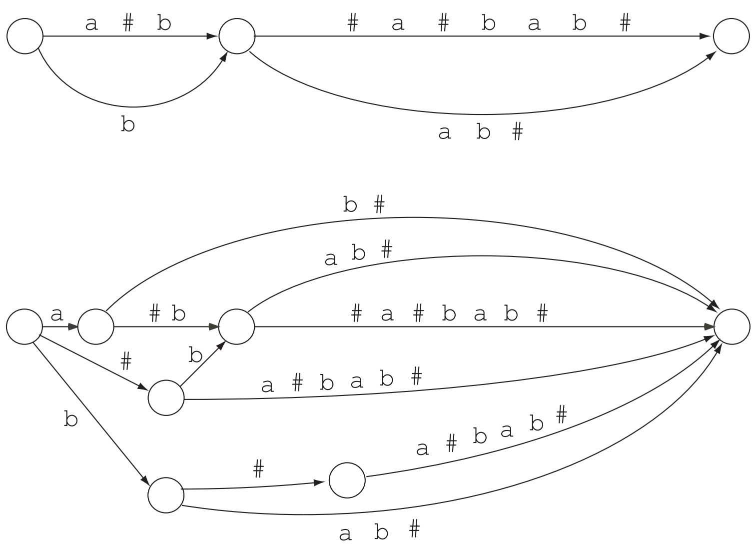


Figure 1. $SCDAWG_D(w)$ with $w = \mathtt{a\#b\#a\#bab\#}$ and $D = \{\mathtt{a}, \mathtt{b}\}^*\#$ is shown on the upper, and normal $CDAWG(w)$ is shown on the lower for comparison. Observe that $SCDAWG_D(w)$ contains only suffixes in $Suffix_D(w)$, while $CDAWG(w)$ has all the suffixes in $Suffix(w)$.

Proof. Assume for contrary that $\overset{w}{\overrightarrow{x}} = x$ and internal node $\overset{w}{\overrightarrow{x}}$ is not branching, i.e., there exists a unique symbol $a \in \Sigma \cup \{\#\}$ such that $Begpos_w(\overset{w}{\overrightarrow{x}}) = Begpos_w(\overset{w}{\overrightarrow{x}} \cdot a)$. Then we have $x \equiv^B_w xa$, which contradicts with the precondition that $\overset{w}{\overrightarrow{x}} = x$. Thus all internal nodes of $SSTree_D(w)$ are branching. □

Now we define *sparse compact directed acyclic word graphs* (*SCDAWGs*), as follows.

Definition 12 (Sparse compact directed acyclic word graph). *The* sparse compact directed acyclic word graph *of string $w \in D^+$, denoted by $SCDAWG_D(w)$, is a DAG (V, E) such that*

$$V = \{[\overset{w}{\overrightarrow{x}}]^E_w \mid x \in Prefix(Suffix_D(w))\},$$
$$E = \left\{ ([\overset{w}{\overrightarrow{x}}]^E_w, a\beta, [\overset{w}{\overrightarrow{xa}}]^E_w) \,\middle|\, \begin{array}{l} x, xa \in Prefix(Suffix_D(w)),\ a \in \Sigma \cup \{\#\}, \\ \beta \in (\Sigma \cup \{\#\})^*,\ and\ \overset{w}{\overrightarrow{x}} a\beta = \overset{w}{\overrightarrow{xa}} \end{array} \right\}.$$

$SCDAWG_D(w)$ has single *source node* $[\overset{w}{\overrightarrow{\varepsilon}}]^E_w = [\varepsilon]^E_w$ of in-degree zero, and single *sink node* $[\overset{w}{\overrightarrow{w}}]^E_w = [w]^E_w$ of out-degree zero.

We associate each node $[\overset{w}{\overrightarrow{x}}]^E_w$ of $SCDAWG_D(w)$ with $length([\overset{w}{\overrightarrow{x}}]^E_w) = |\overset{w}{\overleftarrow{(\overset{w}{\overrightarrow{x}})}}| = |\overset{w}{\overleftrightarrow{x}}|$.

Figure 1 shows $SCDAWG_D(w)$ with $w = \mathtt{a\#b\#a\#bab\#}$ and $D = \{\mathtt{a}, \mathtt{b}\}^*\#$, together with normal $CDAWG(w)$ representing all suffixes of w.

Due to the reflexivity of equivalence relations, for any string x, we have $x \in [x]^E_w$. Consequently, from Definitions 10 and 12, and Theorem 11, we obtain the following theorem regarding the asymptotic size bound of SCDAWGs.

Theorem 13. *For any string $w = w_1 \cdots w_k \in D^+$, $SCDAWG_D(w)$ has $O(k)$ nodes and edges.*

Notice that $SCDAWG_D(w)$ is a minimized version of $SSTree_D(w)$ by the end-equivalence classes. As a matter of fact, $SCDAWG_D(w)$ can be constructed by applying to $SSTree_D(w)$ the DAG minimization algorithm of [13], in time proportional to the number of edges in $SSTree_D(w)$. Due to [8], $SSTree_D(w)$ can be constructed in $O(n)$ time and $O(k)$ space, thus it is possible to build $SCDAWG_D(w)$ in $O(n)$ time and $O(k)$ space. However, this indirect construction wastes extra time and space of once building $SSTree_D(w)$. In the following section, we present our on-line, linear-time algorithm that directly constructs $SCDAWG_D(w)$ in $O(n)$ time and $O(k)$ space. Hence our algorithm consumes only linear space in the output size.

4 On-line Construction Algorithm for SCDAWGs

In this section we present our SCDAWG construction algorithm. Our algorithm is on-line, namely, it sequentially processes the input string $w \in D^+$ from left to right, one by one. To discuss this on-line construction, we extend the definition of $SCDAWG_D(\cdot)$ to any prefix u of $w \in D^+$, by replacing string w with its arbitrary prefix u in Definition 12.

4.1 Suffix Links

In this section we define the *suffix links* of SCDAWGs. We modify the suffix links of normal CDAWGs so that they are suitable for constructing SCDAWGs. The tailored suffix links are essential to the linearity of our SCDAWG construction algorithm.

Let us consider the minimum DFA M_D which accepts $D = \Sigma^*\#$. Then it is easy to see that M_D requires only constant space, with a unique final state (refer to the left of Figure 2). Let q_s and q_f be the initial and final states of M_D, respectively. Then we attach M_D to the SCDAWG, by replacing q_f with the source node of the SCDAWG. Now we are ready to define the suffix links of SCDAWGs.

Definition 14 (Suffix links of SCDAWGs). *For any node $[\overset{u}{\overrightarrow{x}}]^E_u$ of $SDAWG_D(u)$, let z be the shortest member of $[\overset{u}{\overrightarrow{x}}]^E_u$.*

1. *If $\overset{u}{\overrightarrow{x}} \neq u$ and $z \in \Sigma^*$, the suffix link from node $[\overset{u}{\overrightarrow{x}}]^E_u$ goes to the initial state q_s of M_D;*
2. *If $\overset{u}{\overrightarrow{x}} \neq u$ and $z \in (\Sigma \cup \{\#\})^+$, the suffix link from node $[\overset{u}{\overrightarrow{x}}]^E_u$ goes to to node $[y]^E_u$, where y is the longest string in $Suffix_D(z)$ such that $y \notin [\overset{u}{\overrightarrow{x}}]^E_u$;*
3. *Otherwise (If $\overset{u}{\overrightarrow{x}} = u$), the suffix link from $[\overset{u}{\overrightarrow{x}}]^E_u$ is undefined.*

The suffix link of the node in Group 3 is undefined, as it is never used in our construction algorithm to be shown later. See Figure 2 showing $SCDAWG_D(u)$ and its suffix links, where $u = \texttt{a\#b\#a\#bab\#b}$.

4.2 Updating $SCDAWG_D(u)$ to $SCDAWG_D(ua)$

In what follows, we consider to update $SCDAWG_D(u)$ to $SCDAWG_D(ua)$ with $u, ua \in Prefix(w)$, $a \in \Sigma \cup \{\#\}$, and $w \in D^+$.

The next proposition describes what happens to *Wordpos* and *Endpos*.

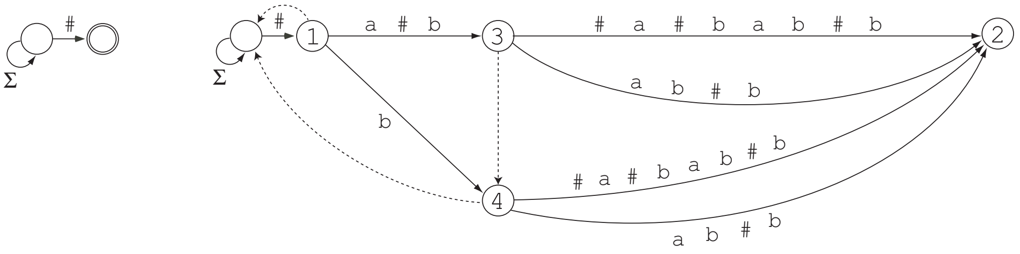


Figure 2. To the left is the minimum DFA M_D accepting dictionary $D = \Sigma^*\#$. To the right is $SCDAWG_D(u)$ with $u = \texttt{a\#b\#a\#bab\#b}$, where Node 1 is its source node. The suffix links are displayed by broken arrows. Nodes 1 and 4 are those of Group 1, Nodes 3 is that of Group 2, and Node 2 is that of Group 3 of Definition 14.

Proposition 15 ([9]). *Let $w \in D^+$ and $u, ua \in Prefix(w)$ with $a \in \Sigma \cup \{\#\}$. Then,*

$$Wordpos_D(ua) = \begin{cases} Wordpos_D(u) \cup \{|ua|\}, & \text{if } u[|u|] = \#; \\ Wordpos_D(u), & \text{otherwise.} \end{cases}$$

Also, for any string $x \in (\Sigma \cup \{\#\})^*$,

$$Endpos_{ua}(x) = \begin{cases} Endpos_u(x) \cup \{|ua|\}, & \text{if } x \in Suffix(ua); \\ Endpos_u(x), & \text{otherwise.} \end{cases}$$

From here on we consider what happens to the nodes of $SCDAWG_D(u)$ when updated to $SCDAWG_D(ua)$.

Proposition 16. *Let $w \in D^+$. For any $u, ua \in Prefix(w)$ with $a \in \Sigma \cup \{\#\}$, $\equiv^B_{ua}$ and $\equiv^E_{ua}$ are a refinement of $\equiv^B_u$ and $\equiv^E_u$, respectively.*

Let $lrs_D(ua)$ denote the longest string in $Suffix_D(ua) \cap Prefix(Suffix_D(u))$. Then we have:

Proposition 17. *Let $w \in D^+$. For any $u, ua \in Prefix(w)$ with $a \in \Sigma \cup \{\#\}$, $[ua]^E_{ua} = Suffix_D(u) \cdot a - Suffix_D(lrs_D(ua))$.*

The above proposition implies that the new sink node $[ua]^E_{ua}$ can be created by extending the incoming edges of the old sink node $[u]^E_u$ with symbol a, and by inserting a new a-labeled edge from each node $[s]^E_u$ to the old sink node, where $s \in Suffix_D(u) - Suffix_D(lrs_D(ua) \cdot a^{-1}) - [u]^E_u$.

Locating $lrs_D(ua)$ in the SCDAWG can be done according to the following proposition.

Proposition 18. *Let $w \in D^+$. For any u, $ua \in Prefix(w)$ with $a \in \Sigma \cup \{\#\}$, $lrs_D(ua) \in Suffix_D(lrs_D(u)) \cdot a$.*

The above proposition implies that we can locate $lrs_D(ua)$ by checking, for each $t \in Suffix_D(lrs_D(u))$, the transitivity from $[t]^E_u$ with new symbol a in the SCDAWG, in the decreasing order of the lengths. When we encounter the first string $y \in Suffix_D(lrs_D(u))$ such that $ya \in Suffix_D(ua)$, then ya is $lrs_D(ua)$. Locating $[t]^E_u$ can be done efficiently by using suffix links of Definition 14.

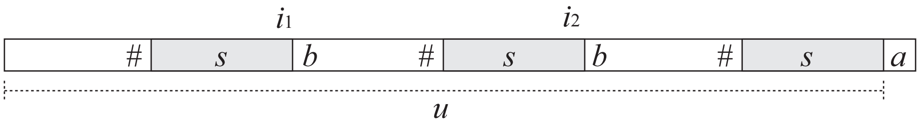


Figure 3. We postpone creating a node corresponding to $s \in \mathit{Suffix}_D(u) - \mathit{Suffix}_D(\mathit{lrs}_D(ua) \cdot a^{-1}) - [u]^E_u$ until we get the first symbol a satisfying $a \neq b = u[i+1]$ for any $i \in \mathit{Endpos}_u(s) \cap (\mathit{Wordpos}_D(u) \oplus |s| \ominus 1)$.

Creating new nodes. While searching for $\mathit{lrs}_D(ua)$, due to Proposition 17, we create a new a-labeled edge from the nodes corresponding to strings $s \in \mathit{Suffix}_D(u) - \mathit{Suffix}_D(\mathit{lrs}_D(ua) \cdot a^{-1}) - [u]^E_u$. However, the following proposition suggests a difficulty of linear time maintenance of those nodes at each stage of updating the SCDAWG.

Proposition 19. *Let $w \in D^+$ and $u \in \mathit{Prefix}(w)$. For any $t \in \mathit{Suffix}_D(u)$, $\overset{u}{\overrightarrow{t}} = t$.*

What is worse, it is possible that $\overset{ua}{\overrightarrow{t}} \neq t$. Thus if we explicitly create a node for every $s \in \mathit{Suffix}_D(u) - \mathit{Suffix}_D(\mathit{lrs}_D(ua) \cdot a^{-1}) - [u]^E_u$ for each $u \in \mathit{Prefix}(w)$, it can take $O(n^2)$ time in total. To avoid this, we 'postpone' creating such a node until we get the first symbol a satisfying $a \neq b = u[i+1]$ for any $i \in \mathit{Endpos}_u(s) \cap (\mathit{Wordpos}_D(u) \oplus |s| \ominus 1)$. (see Figure 3.) This timing coincides with when we insert a new a-labeled edge to the old sink node as mentioned above.

Due to Proposition 7, the new node $[s]^E_{ua}$ can contain more than one string from $\mathit{Suffix}_D(s)$. The equivalence test can be performed according to Lemma 21 given below. Before that, we show Lemma 20 which supports Lemma 21.

Lemma 20. *Let $w \in D^+$, $u \in \mathit{Prefix}(w)$, and $x \in \mathit{Prefix}(\mathit{Suffix}_D(u))$. Let $\overset{u}{\overrightarrow{x}} = z_1$. For any $i \in \mathit{Begpos}_{z_1}(x)$ such that $i > 1$, we have $z_1[i-1] \neq \#$. Similarly, let $\overset{u}{\overleftarrow{x}} = z_2$. For any $j \in \mathit{Begpos}_{z_2}(x)$ such that $j \leq |z_2| - |x|$, we have $z_2[j-1] \neq \#$.*

Proof. Assume for contrary that $z_1[i-1] = \#$. Then, $\mathit{Begpos}_u(x) \cap \mathit{Wordpos}_D(u) = (\mathit{Begpos}_u(z_1) \cup (\mathit{Begpos}_u(z_1) \oplus (i-1)) \cap \mathit{Wordpos}_D(u)$. Since $i > 1$, $\mathit{Begpos}_u(z_1) \oplus (i-1) \neq \mathit{Begpos}_u(z_1)$. Moreover, $\mathit{Begpos}_u(z_1) \oplus (i-1) \subseteq \mathit{Wordpos}_D(u)$. Thus, $(\mathit{Begpos}_u(z_1) \cup (\mathit{Begpos}_u(z_1) \oplus (i-1)) \cap \mathit{Wordpos}_D(u) \neq \mathit{Begpos}_u(z_1) \cap \mathit{Wordpos}_D(u)$, which implies that $\mathit{Begpos}_u(x) \cap \mathit{Wordpos}_D(u) \neq \mathit{Begpos}_u(z_1) \cap \mathit{Wordpos}_D(u)$. However, this contradicts with $x \equiv^B_u z_1$. Similar arguments hold for $\overset{u}{\overleftarrow{x}} = z_2$. □

Lemma 21. *Let $w \in D^+$ and $u \in \mathit{Prefix}(w)$. For any $x, y \in \mathit{Prefix}(\mathit{Suffix}_D(u))$ such that $y \in \mathit{Suffix}_D(x)$,*

$$x \equiv^E_u y \Leftrightarrow [\overset{u}{\overrightarrow{x}}]^E_u = [\overset{u}{\overrightarrow{y}}]^E_u.$$

Proof. If $x \equiv^E_u y$, then $\overset{u}{\overleftarrow{x}} = \overset{u}{\overleftarrow{y}}$. By Proposition 9, $(\overset{u}{\overleftarrow{\overset{u}{\overrightarrow{x}}}}) = (\overset{u}{\overrightarrow{\overset{u}{\overleftarrow{x}}}})$ and $(\overset{u}{\overleftarrow{\overset{u}{\overrightarrow{y}}}}) = (\overset{u}{\overrightarrow{\overset{u}{\overleftarrow{y}}}})$. Thus we get $(\overset{u}{\overleftarrow{\overset{u}{\overrightarrow{x}}}}) = (\overset{u}{\overleftarrow{\overset{u}{\overrightarrow{y}}}})$, which implies that $[\overset{u}{\overrightarrow{x}}]^E_u = [\overset{u}{\overrightarrow{y}}]^E_u$.

Now suppose $[\overset{u}{\overrightarrow{x}}]^E_u = [\overset{u}{\overrightarrow{y}}]^E_u$. Let $\overset{u}{\overrightarrow{x}} = x\alpha$ and $\overset{u}{\overleftarrow{x}} = \beta x$ for some strings $\alpha, \beta \in (\Sigma \cup \{\#\})^*$. Let $z = \overset{u}{\overleftrightarrow{x}}$. Then $z = \beta x \alpha$ by definition. Since $y \in \mathit{Suffix}_D(x)$, there

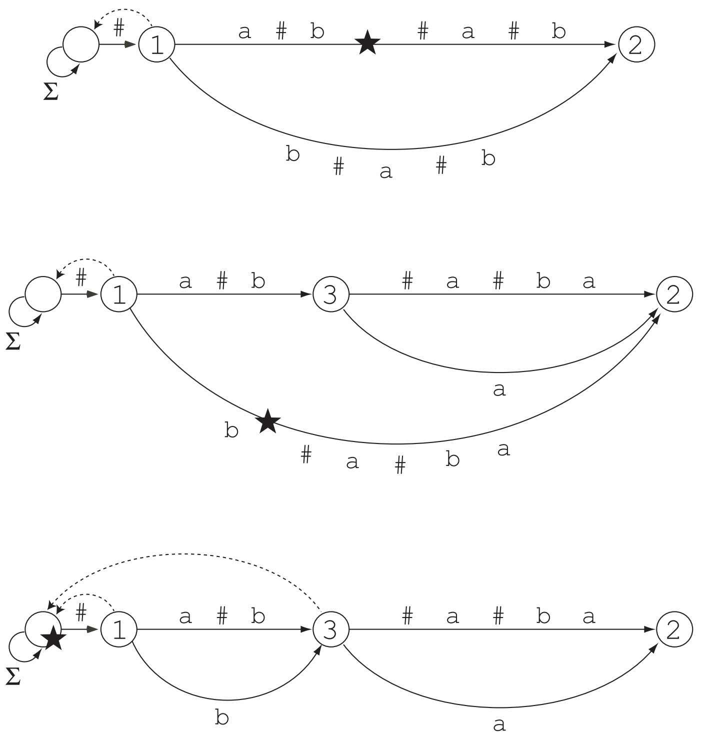


Figure 4. A new node is created in the update of $SCDAWG_D(u)$ to $SCDAWG_D(u\mathtt{a})$, where $u = \mathtt{a\#b\#a\#b}$. The stars represent the location from which we check a transitivity with a new symbol $\mathtt{a}$. When the transitivity check failed on an edge (namely, not right on a node), we divide the edge into two at the check point (where the star lies) and create a new node there (Node 3 in this figure) together with a new edge labeled with the new character leading to the sink node. Assume that the next transitivity check point also lies on an edge. We apply Lemma 21 and if it is the case, we 'shorten' the edge till the check point and merge this shortened edge to the above created node, as seen in the conversion from the second graph to the third one of this figure.

exists some string $\gamma \in (\Sigma \cup \{\#\})^*$ such that $\gamma y = x$. Because $\overset{u}{\overleftrightarrow{x}} = \overset{u}{\overleftrightarrow{y}}$, $z = \beta\gamma y\alpha$. By Lemma 20, $\overset{u}{\overrightarrow{y}} = y\alpha$ and $\overset{u}{\overleftarrow{y}} = \beta\gamma y$. Hence,

$$
\begin{aligned}
& \mathit{Endpos}_u(x) \cap (\mathit{Wordpos}_D(u) \oplus |x| \ominus 1) \\
= {} & \mathit{Endpos}_u(\beta x) \cap (\mathit{Wordpos}_D(u) \oplus |\beta x| \ominus 1) \\
= {} & \mathit{Endpos}_u(\beta\gamma y) \cap (\mathit{Wordpos}_D(u) \oplus |\beta\gamma y| \ominus 1) \\
= {} & \mathit{Endpos}_u(y) \cap (\mathit{Wordpos}_D(u) \oplus |y| \ominus 1),
\end{aligned}
$$

which implies that $x \equiv^E_u y$. □

Figure 4 displays how a new node is created.

Splitting a node. Lastly, we remark that there is a possibility that a certain node of $SCDAWG_D(u)$ can be split into two nodes in $SCDAWG_D(ua)$, as shown in the following lemma that has inherently been shown in [9].

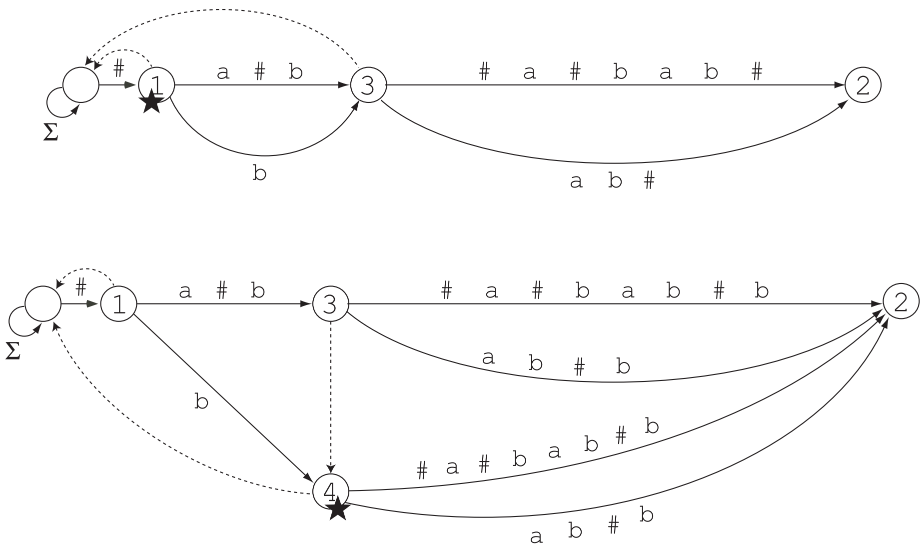


Figure 5. Illustration for node splitting. Node 3 is split in the update of $SDAWG_D(u)$ to $SCDAWG_D(u\mathtt{b})$ with $u = \mathtt{a\#b\#a\#bab\#}$. The stars represent the location from which we check a transitivity with a new symbol.

Lemma 22. *Let $w \in D^+$, and let $u, ua \in Prefix(w)$ with $a \in \Sigma \cup \{\#\}$. Let $z = lrs_D(ua)$. Then, for any $x \in Prefix(Suffix_D(u))$, we have*

$$[x]_u^E = \begin{cases} [\overset{u}{\overleftrightarrow{x}}]_{ua}^E \cup [z]_{ua}^E, & \textit{if } z \in [x]_u^E \textit{ and } z \neq \overset{u}{\overleftrightarrow{x}}; \\ [x]_{ua}^E, & \textit{otherwise.} \end{cases}$$

To node $[x]_u^E$ such that $z \in [x]_u^E$, we examine whether $z = \overset{u}{\overleftrightarrow{x}}$ or not by checking the length of $\overset{u}{\overleftrightarrow{x}}$ and z, as follows. Consider edge $([y]_u^E, \alpha, [x]_u^E)$ such that $\overset{u}{\overleftrightarrow{y}} \cdot \alpha = z$. Then, it is easy to see that

$$z = \overset{u}{\overleftrightarrow{x}} \Leftrightarrow |\overset{u}{\overleftrightarrow{y}} \cdot \alpha| = |\overset{u}{\overleftrightarrow{x}}| \Leftrightarrow |\overset{u}{\overleftrightarrow{y}}| + |\alpha| = |\overset{u}{\overleftrightarrow{x}}| \Leftrightarrow length([y]_u^E) + |\alpha| = length([x]_u^E).$$

Setting the length of the initial state q_s of M_D to be -1, no contradiction occurs even in case that $z = \varepsilon$.

See Figure 5 for a concrete example of node splitting.

Pseudo Code. A pseudo code of our on-line algorithm to build SCDAWGs is shown in Figures 6 and 7. Any edge label $\alpha \in (\Sigma \cup \{\#\})^+$ is implemented as an ordered pair (i, j) of positions such that $u[i..j] = \alpha$, in order to implement the SCDAWG with $O(k)$ space. To neglect to extend the existing in-coming edges of *sink* node (refer to Proposition 17), we implement by (i, ∞) the label of any edge leading to the sink node. This is the same idea as in [14].

For each $i = 1, \ldots, n$, we call function *Update* which converts $SCDAWG_D(w[1..i-1])$ to $SCDAWG_D(w[1..i])$ and returns the location of $lrs_D(w[1..i])$ by pair (s, k). Here, s is the lowest node in the path that spells out $lrs_D(w[1..i])$ from the source node. Let $lrs_D(w[1..i]) = xy$ such that x belongs to the equivalence class of node s and y is the rest. Then, k is the integer such that $w[k : i] = y$.

```
Input:   w = w[1..n] ∈ D+ and M_D with initial state q_s and final state q_f.
Output:  SCDAWG_D(w).
{
    /* We assume Σ = {w[−1], w[−2], . . . , w[−m]} */.
    /* Replace the edge labels of M_D with appropriate integer pairs */.
    length(q_f) = 0;   length(q_s) = −1;   length(sink) = ∞;
    source = q_f;   link(source) = q_s;   link(sink) = nil;
    (s, k) = (source, 1);
    for (i = 1; i ≤ n; i++)  (s, k) = Update(s, (k, i));
}

(node,integer)-pair Update(s, (k, i)) {
    oldr = nil; s′ = nil;
    while (CheckEndPoint(s, (k, i − 1), w[i]) == false) {
        if (k ≤ i − 1) {  /* (s, (k, i − 1)) is implicit. */
            if (s′ == Extension(s, (k, p − 1))) {
                let (s, (k′, p′), s′) be the w[k]-edge from s;
                replace the edge by edge (s, (k′, k′ + p − k − 1), r);
                (s, k) = Canonize(link(s), (k, p − 1));
                continue;
            }
            else {
                s′ = Extension(s, (k, p − 1));
                r = CreateNode(s, (k, p − 1));
            }
        } else r = s;     /* (s, (k, i − 1)) is explicit. */
        create new edge (r, (i, ∞), sink);
        if (oldr ≠ nil) link(oldr) = r;
        oldr = r;
        (s, k) = Canonize(link(s), (k, i − 1));
    }
    if (oldr ≠ nil) link(oldr) = s;
    return SplitNode(s, (k, i));
}
```

Figure 6. Main routine and function *Update* of our SCDAWG construction algorithm. For any node s, $link(s)$ denotes the node to which the suffix link of s goes. By 'implicit' we mean that the location is on an edge, and by 'explicit' we mean that it is on a node.

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

node Extension(s, (k, p)) {
    if (k > p) return s;    /* (s, (k, p)) is explicit. */
    find the w[k]-edge (s, (k', p'), s') from s;
    return s';
}

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

node CreateNode(s, (k, p)) {
    let (s, (k', p'), s') be the w[k]-edge from s;
    create new node r;
    replace the edge by edges (s, (k', k' + p − k), r) and (r, (k' + p − k + 1, p'), s');
    length(r) = length(s) + (p − k + 1);
    return r;
}

(node,integer)-pair SplitNode(s, (k, p)) {
    (s', k') = Canonize(s, (k, p));
    if (k' ≤ p) return (s', k');    /* (s', (k', p)) is implicit. */
    /* (s', (k', p)) is explicit. */
    if (length(s') == length(s) + (p − k + 1)) return (s', k');
    create node r' as a duplication of s' with the out-going edges;
    link(r') = link(s');    link(s') = r';
    length(r') = length(s) + (p − k + 1);
    do {
        replace the w[k]-edge from s to s' by edge (s, (k, p), r');
        (s, k) = Canonize(link(s), (k, p − 1));
    } while ((s', k') = Canonize(s, (k, p)));
    return (r', p + 1);
}
```

Figure 7. The other functions of our on-line SCDAWG construction algorithm. All these functions are identical to those in [7].

Function *CheckEndPoint* returns **true** if the location indicated by triple $(s, (k, i))$ corresponds to $lrs_D(w[1..i])$, and **false** otherwise. Due to Proposition 17, in the **while** loop of function *Update*, we create new edges $(r, (i, \infty), sink)$. This is continuously done until finding $lrs_D(w[1..i])$, according to Proposition 18. Consider the case that $(s, (k, i))$ lies on an edge, and that we have created a new node r in the first **else** condition of the **while** loop. Due to Proposition 21, when the second **if** condition is satisfied, we shorten the edge and redirect it to node r . Note that, since s' is initially set to **nil**, this second **if** condition can be satisfied only after the **else** condition is once satisfied and s' gets a non-**nil** value. After creating new edge $(r, (i, \infty), sink)$, we traverse the suffix link of node s to find $lrs_D(w[1..i])$.

After the insertion of all the new edges, we call function *SplitNode* that splits the node corresponding to $lrs_D(w[1..i])$ into two nodes, when needed. This operation is due to Lemma 22.

We remark that the only difference between our algorithm and the on-line algorithm of [7] for constructing normal CDAWGs is the initialization steps of the main routine where we set the source of the SCDAWG to the final state q_f of M_D and the suffix link of the source to the initial state q_s of M_D. These simple modifications make the proposed algorithm construct SCDAWGs together with their suffix links.

For the correctness of the algorithm, we attach an end-marker \$ to any input string $w \in D^+$, which appears nowhere in w. Possible problems that may be caused by the 'delay' of creating new nodes, can be cleared by this end-marker, since \$ appears nowhere in w.

Theorem 23. *For any string $w \in D^+$, the algorithm of Figures 6 and 7 correctly constructs $SCDAWG_D(w\$)$.*

Now the only remaining matter is the time complexity of the algorithm. The following theorem can be proven by the same idea as the linearity proof of the normal CDAWG construction algorithm in [7].

Theorem 24. *For any $w \in D^+$ such that $w = w_1 \cdots w_k$ and $|w| = n$, the algorithm of Figures 6 and 7 runs in $O(n)$ time using $O(k)$ space.*

5 Conclusions and Open Problems

In this paper we introduced a new text indexing structure, sparse compact directed acyclic word graphs (SCDAWGs) for strings $w = w_1 \cdots w_k$ over dictionary $D = \Sigma^*\#$. We showed that SCDAWGs require only $O(k)$ space and are strictly smaller than sparse (word) suffix trees. Furthermore, we presented an on-line algorithm that builds SCDAWGs in $O(n)$ time, where $n = |w|$. The proposed algorithm correctly builds SCDAWGs with the help of the minimum DFA M_D accepting D, and the tailored suffix links. SCDAWGs are expected to become a space-economical alternative to sparse suffix trees in application areas such as natural language processing, biological sequence analysis, etc.

Here are some open problems regarding sparse text indexing structures:

1. Exact numbers of nodes and edges of SCDAWGs. Being a tree with k leaves and only branching internal nodes, any sparse suffix tree can have at most $2k-1$ nodes and $2k-2$ edges. Thus, it is guaranteed that any SCDAWG has less nodes and edges than these.

2. Would it be possible to construct *sparse suffix arrays* efficiently? Sparse suffix arrays can be obtained from the leaves of the corresponding sparse suffix trees, but is it possible to build sparse suffix arrays directly, and in $O(n)$ time with $O(k)$ space?

References

1. A. Andersson, N. J. Larsson, and K. Swanson: *Suffix trees on words*. Algorithmica, 23(3) 1999, pp. 246–260.
2. H. Bannai, S. Inenaga, A. Shinohara, M. Takeda, and S. Miyano: *Efficiently finding regulatory elements using correlation with gene expression*. Journal of Bioinformatics and Computational Biology, 2(2) 2004, pp. 273–288.
3. A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M. T. Chen, and J. Seiferas: *The smallest automaton recognizing the subwords of a text*. Theoretical Computer Science, 40 1985, pp. 31–55.
4. A. Blumer, J. Blumer, D. Haussler, R. McConnell, and A. Ehrenfeucht: *Complete inverted files for efficient text retrieval and analysis*. Journal of the ACM, 34(3) 1987, pp. 578–595.
5. B. Dorohonceanu and C. G. Nevill-Manning: *Accelerating protein classification using suffix trees*, in Proc. 8th International Conference on Intelligent Systems for Molecular Biology (ISMB'00), AAAI Press, 2000, pp. 128–133.
6. S. Inenaga, T. Funamoto, M. Takeda, and A. Shinohara: *Linear-time off-line text compression by longest-first substitution*, in Proc. 10th International Symp. on String Processing and Information Retrieval (SPIRE'03), vol. 2857 of Lecture Notes in Computer Science, Springer-Verlag, 2003, pp. 137–152.
7. S. Inenaga, H. Hoshino, A. Shinohara, M. Takeda, S. Arikawa, G. Mauri, and G. Pavesi: *On-line construction of compact directed acyclic word graphs*. Discrete Applied Mathematics, 146(2) 2005, pp. 156–179.
8. S. Inenaga and M. Takeda: *On-line linear-time construction of word suffix trees*, in Proc. 17th Ann. Symp. on Combinatorial Pattern Matching (CPM'06), vol. 4009 of Lecture Notes in Computer Science, Springer-Verlag, 2006, pp. 60–71.
9. S. Inenaga and M. Takeda: *Sparse directed acyclic word graphs*, in Proc. 13th International Symp. on String Processing and Information Retrieval (SPIRE'06), Lecture Notes in Computer Science, Springer-Verlag, 2006, To appear.
10. N. J. Larsson: *Extended application of suffix trees to data compression*, in Proc. Data Compression Conference '96 (DCC'96), IEEE Computer Society, 1996, pp. 190–199.
11. L. Marsan and M.-F. Sagot: *Extracting structured motifs using a suffix tree - algorithms and application to promoter consensus identification*, in Proc. 4th Annual International Conference on Computational Molecular Biology (RECOMB'00), ACM, 2000, pp. 210–219.
12. J. C. Na, A. Apostolico, C. S. Iliopoulos, and K. Park: *Truncated suffix trees and their application to data compression*. Theoretical Computer Science, 304(1–3) 2003, pp. 87–101.
13. D. Revuz: *Minimisation of acyclic deterministic automata in linear time*. Theoretical Computer Science, 92(1) 1992, pp. 181–189.
14. E. Ukkonen: *On-line construction of suffix trees*. Algorithmica, 14(3) 1995, pp. 249–260.

Reachability on Suffix Tree Graphs

Yasuto Higa[1], Hideo Bannai[1], Shunsuke Inenaga[1,2], and Masayuki Takeda[1,3]

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

Abstract. We analyze the complexity of graph reachability queries on ST-*graph*s, defined as directed acyclic graphs (DAGs) obtained by merging the suffix tree of a given string and its suffix links. Using a simplified reachability labeling algorithm presented by Agrawal *et al.* (1989), we show that for a random string of length n, its ST-graph can be preprocessed in $O(n \log n)$ expected time and space to answer reachability queries in $O(\log n)$ time. Furthermore, we present a series of strings that require $\Theta(n\sqrt{n})$ time and space to answer reachability queries in $O(\log n)$ time for the same algorithm. Exhaustive computational calculations for strings of length $n \leq 33$ have revealed that the same strings are also the worst case instances of the algorithm. We therefore conjecture that reachability queries can be answered in $O(\log n)$ time with a worst case time and space preprocessing complexity of $\Theta(n\sqrt{n})$.

Keywords: algorithms and data structures, suffix trees, graph reachability

1 Introduction

The reachability query for two nodes u, v of a given directed graph is to answer whether or not there exists a path in the graph that starts from u and ends at v. For any given graph, the query can be answered in $O(n + e)$ time by conducting a simple depth-first traversal on the graph, where n is the number of nodes and e is the number of edges in the graph.

There have been several studies on preprocessing a graph in order to answer reachability queries more efficiently [5,1,7,3,9]. A simple approach would be to construct the transitive closure of the graph, achieving $O(1)$ time query at the cost of $O(n^2)$ time and space for the preprocessing. For graphs with specific properties, there exists methods with smaller complexity bounds. Graph reachability for planar graphs with a single source node and sink node was considered in [5]. Reachability queries for such graphs can be answered in $O(1)$ time given $O(n + e)$ time and $O(n)$ space preprocessing. For partial lattices, a method which achieves $O(1)$ time query with $O(n^2)$ time and $O(n\sqrt{n})$ space preprocessing was shown in [7], where n is the size of the ground set. When considering arbitrary graphs with n vertices and e edges, it has been shown in [3] that for any labeling scheme, there exists a graph such that the reachability labeling has total size of $\Omega(n\sqrt{e})$.

In this paper, we consider the graph reachability query problem on ST-*graph*s, which are DAGs derived from suffix trees and suffix links. ST-graphs are not planar, are not partial lattices. A suffix tree of a given string is a data structure that captures important information concerning the substrings of the string [10]. We present and analyze a version of the interval labeling algorithm of [1] tailored for ST-graphs. It can be shown that for a random string, the ST-graph can be preprocessed in $O(n \log n)$

expected time and space to answer reachability queries in $O(\log n)$ time. Furthermore, we present a series of strings for which their ST-graphs require $\Theta(n\sqrt{n})$ time and space of preprocessing when the algorithm is applied. Exhaustive computational calculations indicate that the series gives the worst case instances of the algorithm for the strings of length up to 33, strongly supporting that the worst case complexity of the preprocessing is $\Theta(n\sqrt{n})$ time and space. Assuming this is true, this would break the $O(n^2)$ barrier for total time and space used when conducting $O(n)$ queries.

Reachability on ST-graphs solve the problem of whether or not the string represented by the path from the root to the given query nodes are substrings of each other. The algorithm has possible applications in substring pattern set discovery, where the objective is to find best set of substrings that characterizes a given set (or sets) of strings: Consider two substring patterns such that one is a substring of the other. The set of strings which contain the former pattern as a substring is obviously a subset of the set of strings which have the latter pattern as a substring. This property may allow us choose the best pattern set more efficiently, for example, by quickly detecting non-interesting pattern sets.

2 Preliminaries

Let Σ be a finite *alphabet*. An element of Σ^* is called a *string*. Strings x, y, and z are said to be a *prefix*, *substring*, and *suffix* of string $w = xyz$, respectively. The length of a string w is denoted by $|w|$. The empty string is denoted by ε, that is, $|\varepsilon| = 0$. Unless otherwise noted, we shall only consider strings of a fixed alphabet. Also, we assume that all strings end with a unique character $\$ \in \Sigma$ that does not occur anywhere else in the strings.

A *suffix tree* $suftree(s)$ for a given string s is a rooted tree whose edges are labeled with non-empty substrings of s, satisfying the following characteristics. For any node v in the suffix tree, let $path(v)$ denote the string spelled out by concatenating the edge labels on the path from the root to v. For each leaf node v, $path(v)$ is a distinct suffix of s, and for each suffix of s, there exists such a leaf v. Furthermore, each internal node has at least two children, and the first character of the labels on the edges to its children are distinct. The parent of node v is denoted by $parent_s(v)$, and the set of children of node v is denoted by $children_s(v)$. The *length* of node v is defined to be $|path(v)|$. The *depth* of node v with respect to the suffix tree is the number of edges on the path from the root to the node, and is always less than or equal to $|path(v)|$. The *height* of a suffix tree is the maximum depth of all nodes with respect to the suffix tree. Also, let $subtree_s(v)$ be the subtree of the suffix tree rooted at node v.

For a node v where $path(v) = \sigma x$ for some $\sigma \in \Sigma$ and $x \in \Sigma^*$, we denote the *suffix link* of v as $parent_l(v) = u$ where $path(u) = x$. It is easy to see that a unique $parent_l(v)$ exists for each node v in $suftree(s)$, except for the root node. Therefore, the suffix links also form a tree structure, which we denote by $suflinktree(s)$. Let $children_l(v) = \{u : parent_l(u) = v\}$. Note that the depth of node v with respect to the suffix link tree is always equivalent to $|path(v)|$. Also, let $subtree_l(v)$ be the subtree of the suffix link tree rooted at node v.

2.1 ST-graphs

Let V be the set of nodes of $suftree(s)$. Let us denote the set of (backward) edges of $suftree(s)$ by $E_s = \{(v, parent_s(v)) : v \in V\}$ and the set of edges of the $suflinktree(s)$

by $E_l = \{(v, parent_l(v)) : v \in V\}$. We define the *ST-graph* of a string s as the directed graph $G = (V, E)$ where $E = E_s \cup E_l$. It is well known that the suffix tree and its suffix links for a string of length n can be constructed and represented in $O(n)$ time and space [10,6,8,4]. Figure 1 shows an example of an ST-graph for the string `ababbabbba$`. It is easy to see that the graph is not planar nor a partial lattice.

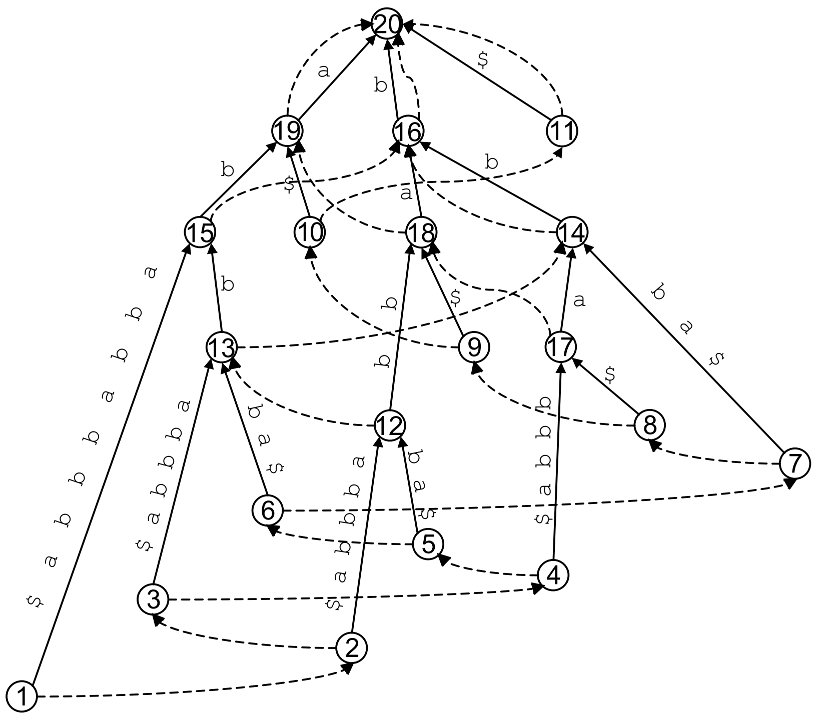


Figure 1. A graph induced from the suffix tree of string `ababbabbba$`. Solid edges represent edges of the suffix tree. The dashed edges represent suffix links.

node	interval	labels	node	interval	labels
1	[1,1]	[1,1]	11	[1,11]	[1,11]
2	[1,2]	[1,2]	12	[12,12]	[1,5],[12,12]
3	[1,3]	[1,3]	13	[12,13]	[1,6],[12,13]
4	[1,4]	[1,4]	14	[12,14]	[1,8],[12,14]
5	[1,5]	[1,5]	15	[15,15]	[1,6],[12,13],[15,15]
6	[1,6]	[1,6]	16	[12,16]	[1,9],[12,16],[17,18]
7	[1,7]	[1,7]	17	[17,17]	[1,8],[17,17]
8	[1,8]	[1,8]	18	[17,18]	[1,9],[12,12],[17,18]
9	[1,9]	[1,9]	19	[17,19]	[1,10],[12,13],[15,15],[17,19]
10	[1,10]	[1,10]	20	[1,20]	[1,20]

Table 1. Post-order interval and labels assigned to ST-graph of Fig. 1 by Algorithm 1

The problem we shall consider in this paper is as follows:

Problem 1 (ST-graph reachability query). Given the ST-graph $G = (V, E)$ of string s and an arbitrary pair of node $u, v \in V$, $rquery(u, v)$ returns `true` if there exists a path from node u to v in G, and `false` otherwise.

The query $rquery(u, v)$ is equivalent to the query of whether the string $path(v)$ is a substring of $path(u)$ or not.

Lemma 2. *Given an ST-graph $G = (V, E)$ of string s and nodes $u, v \in V$,*

$$\mathit{rquery}(u, v) = \texttt{true} \iff \mathit{path}(v) \text{ is a substring of } \mathit{path}(u)$$

Proof. ($\Rightarrow$) Suppose v is reachable from u. An edge $(p, q) \in E_s$ implies that $\mathit{path}(q)$ is a substring (prefix) of $\mathit{path}(p)$. An edge $(p, r) \in E_l$ implies that $\mathit{path}(r)$ is a substring (suffix) of $\mathit{path}(p)$. Since any path from u to v consists of edges in $E_s \cup E_l$, this implies that v is a substring of u.
($\Leftarrow$) Suppose $\mathit{path}(v)$ is a substring of $\mathit{path}(u)$, i.e. there exists $x, z \in \Sigma^*$ such that $\mathit{path}(u) = x\mathit{path}(v)z$. If $x = \varepsilon$, then $\mathit{path}(v)$ is a prefix of $\mathit{path}(u)$ which implies $u \in \mathit{subtree}_s(v)$, and that v is reachable from u using edges of E_s. For $x = x_1 \cdots x_k$ ($k \geq 1$), the nodes reachable from u using edges in E_l will have corresponding paths: $x_2 \cdots x_k\mathit{path}(v)z$, $x_3 \cdots x_k\mathit{path}(v)z$, ..., $\mathit{path}(v)z$. Let v' be the node where $\mathit{path}(v') = \mathit{path}(v)z$. Then, since $\mathit{path}(v)$ is a prefix of $\mathit{path}(v')$, v is reachable from v', and therefore reachable from u. □

Corollary 3. *For any two distinct nodes $u, v \in V$ in an ST-graph (V, E), if there exists a path from u to v, then there exists a path: $v_0 \to v_1 \to \ldots \to v_t$ where $t > 0$, $u = v_0$, $v_t = v$, and for some $0 \leq i \leq t$, $(v_{j-1}, v_j) \in E_l$ for all $1 \leq j \leq i$ and $(v_{j-1}, v_j) \in E_s$ for all $i + 1 \leq j \leq t$.*

Proof. Since there exists a path from u to v, $\mathit{path}(v)$ is a substring of $\mathit{path}(u)$ by Lemma 2. The Corollary follows from the argument for the ($\Leftarrow$) part of the proof of Lemma 2. □

3 Interval Labeling Algorithm

In this paper, we analyze the complexity of the interval labeling algorithm for general DAGs presented in [1], when applied to *ST*-graphs. The original algorithm works as follows: First, create a spanning tree of the DAG by examining the nodes in topological order. The parent of each node is chosen so that the number of ancestors of each node is the largest. Then, intervals based on a postorder numbering of the spanning tree are assigned to each node. Further, the intervals of each node are propagated to all its ancestor nodes of the DAG, and as a result, each node will hold a set of interval labels. During the propagation, for a given set of interval labels at each node, *redundant* intervals which are subsumed by larger intervals in the same set are removed.

The interval labeling algorithm of [1] modified for *ST*-graphs is given in Algorithm 1, and Algorithm 2 shows how to answer $\mathit{rquery}(,)$ using the labels. Algorithm 1 has been simplified as follows: First, the topological ordering and spanning tree construction is not necessary. This is because each node v will only have at most two parents, one from the suffix tree, and the other from the suffix link tree. It is easy to verify that the path from the root to v using the edges of the suffix link tree is always at least as long as the path using the edges of the suffix tree. Therefore, the suffix link tree already corresponds to the desired spanning tree, by which intervals are assigned to each node based on a postorder numbering. Second, although the original interval labeling algorithm requires labels to be propagated across all edges, this is not required for *ST*-graphs, and are propagated only across edges in E_s.

The correctness of the algorithm can be proved as follows. Suppose node v is reachable from node u. Then, node v must hold an interval label which subsumes the interval of u for Algorithm 2 to correctly answer $\mathit{rquery}(u, v)$. From lines 1-1 in Algorithm 1, node v holds all intervals of nodes in $subtree_s(v)$, which includes the interval for node v_i that can be reached by traversing suffix links starting from u as in Corollary 3. Since the interval of v_i subsumes the interval of u defined by the postorder numbering assigned in lines 1-1, v will contain an interval label which subsumes the interval of u.

4 Complexity

In this section, we will derive estimates on the complexity of Algorithm 1. In particular, we will consider the expected running time, as well as lower bounds for the worst case.

Lemma 4. *Assuming a constant size alphabet, Algorithm 1 runs in time linear in the total number of labels assigned to the nodes of the ST-graph.*

Proof. The maximum in-degree of each node is bounded by $O(|\Sigma|)$. This is because for the suffix tree, all labels on the edges must begin with a different character of the alphabet. For the suffix link tree, $|\{v : path(v) = \sigma path(u), \sigma \in \Sigma\}| \leq |\Sigma|$ for any node u. Therefore, assuming that $|\Sigma|$ is constant, merging the sorted labels (and removing subsumed intervals) from the in-coming nodes can be done in time linear in the total size of the in-coming labels. □

From Lemma 4, we have only to count the number of labels that will be assigned to the ST-graph by Algorithm 1 in order to estimate the complexity of the algorithm.

4.1 Expected running time

A simple bound relating the height of the suffix tree and the total number of labels assigned to the ST-graph is shown in the following lemma.

Lemma 5. *For an ST-graph for a string of length n whose suffix tree has height h, the total number of interval labels assigned by Algorithm 1 is at most $O(nh)$.*

Proof. Notice that since the interval labels are only propagated through edges of the suffix tree, the maximum number of labels at a given node v is bounded by the number of nodes in $subtree_s(v)$. Therefore, at a given depth of the suffix tree, there can only be a maximum total of $O(n)$ labels, i.e., the total number of nodes in the suffix tree, since the subtrees of nodes of the same depth cannot intersect. This results in a maximum total of $O(nh)$ labels for all nodes. □

Theorem 6. *The expected running time of Algorithm 1 for a random string of length n is $O(n \log n)$.*

Proof. It is known that the expected height of the suffix tree of a random string of length n is $O(\log n)$ [2]. The theorem follows from Lemma 4 and Lemma 5. □

4.2 Worst Case Lower Bounds

In this subsection, we will give a lower bound for the worst case complexity of Algorithm 1. We will present a series of strings of length n whose ST-graphs will have $\Theta(n\sqrt{n})$ labels assigned by the algorithm. Prior to this, we show related properties of suffix trees and suffix link trees.

Algorithm 1: Assign labels to each node of the ST-graph.

Input: ST-graph $\mathsf{G(V,E)}$ of string s
Output: Labeled ST-graph ($\mathsf{v.int}$ and $\mathsf{v.labels}$ for all $\mathsf{v} \in \mathsf{V}$)
1 **foreach** *node* $\mathsf{v} \in \mathsf{V}$ *in post-order of suflinktree*(s) **do**
2 | $\mathsf{v.int} := [\mathbf{min}\{\text{post-order number of } subtree_l(\mathsf{v})\}, \text{post-order number of } \mathsf{v}]$;
3 | $\mathsf{v.labels} := \{\mathsf{v.int}\}$;
4 **endfch**
5 **foreach** *node* $\mathsf{v} \in \mathsf{V}$ *in post-order of suftree*(s) **do**
6 | $\mathsf{v.labels}$:= merge and sort $\mathsf{v.labels}$ and $\{\mathsf{c.labels} \ : \ \mathsf{c} \in children_s(\mathsf{v})\}$;
7 | Remove $[i,j] \in \mathsf{v.labels}$ if $\exists [i',j'] \in \mathsf{v.labels}$ s.t. $i' \leq i \leq j \leq j'$;
8 **endfch**

Algorithm 2: $rquery(u,v)$ on ST-graphs using labels assigned by Algorithm 1.

Input: Labeled ST-graph $\mathsf{G(V,E)}$ of string s and nodes $\mathsf{u},\mathsf{v} \in \mathsf{V}$
Output: $rquery(\mathsf{u},\mathsf{v})$
1 $[i,j] = \mathsf{u.int}$;
2 **if** $\exists [i',j'] \in \mathsf{v.label}$ *such that* $i' \leq i \leq j \leq j'$ **then return** `true`;
3 **return** `false`;

Properties of ST-graphs. For $|\Sigma| = 2$, there can only be one string of a given length n, and the structure of the ST-graph is determined uniquely (recall that all strings terminate with a uniquely occurring character $\$ \in \Sigma$). It is not difficult to show that the total number of labels is $3(n-1) = O(n)$ in such case. In what follows we therefore consider the case for $|\Sigma| \geq 3$.

Lemma 7. *The number of interval labels assigned to each node of the ST-graph for any string s is bounded by the number of leaves in suflinktree(s). More generally, the exact number of labels assigned to node v is $\boldsymbol{min}_{W \subseteq subtree_s(v)}\{|W| \ : \ subtree_s(v) \subseteq \cup_{w \in W} subtree_l(w)\}$.*

Proof. Let ℓ be the number of leaves in *suflinktree*(s). In the post-order traversal on *suflinktree*(s) (lines 1-1 of Algorithm 1), each node receives exactly one interval label, and there are exactly ℓ different values for the first element of the intervals. In post-order traversal on *suftree*(s) (lines 1-1 of Algorithm 1), we remove subsumed intervals and therefore each node gets at most ℓ interval labels. The exact number of labels follows from a similar argument. □

Lemma 8. *If and only if $|children_s(v)| = 2$ for any internal node $v \neq root$ of suftree(s), the number of internal nodes (excluding the root) is the maximum, which is $n-3$.*

Proof. Because of \$ there are always n leaves in *suftree*(s). Since there is always an edge labeled with \$ from the root, $|children_s(root)| = |\Sigma|$. □

Lemma 9. *Assume suftree(s) satisfies the condition in Lemma 8. For the three following groups of the internal nodes of suftree(s),*

n_1*: internal nodes with two leaf-children;*
n_2*: internal nodes with one leaf-child and one internal-child;*
n_3*: internal nodes with two internal-children;*

where $n_1 + n_2 + n_3 = n - 3$, *we have*

$$n_2 = n - 2n_1 - 1, \text{ and}$$
$$n_3 = n_1 - 2.$$

Proof. Because of \$, *suftree*(s) always has n leaves. Since there are n_1 internal nodes with two leaf-children, and since the leaf corresponding to suffix \$ is a child of the root, we have $n_2 = n - 2n_1 - 1$. Finally $n_3 = n - 3 - n_1 - n_2 = n_1 - 2$. □

Lemma 10. *Assume suftree*(s) *satisfies the condition in Lemma 8. For any node* v, *if* $|children_l(v)| \geq 2$, *then* v *is a group* n_3 *node in suftree*(s).

Proof. Let $x = path(v)$, and let $u, w \in children_l(v)$. Since u and w are nodes in the suffix tree, there exists at least four possible suffixes whose paths must pass node v. We denote these paths as $x\sigma_1 y_1, x\sigma_2 y_2, x\sigma_3 y_2, x\sigma_4 y_4$ where $\sigma_i \in \Sigma$ and $y_i \in \Sigma^*$ for $1 \leq i \leq 4$. It must be that $\sigma_{i_1} = \sigma_{i_2}$ and $\sigma_{i_3} = \sigma_{i_4}$ for some $\{i_1, i_2, i_3, i_4\} = \{1, 2, 3, 4\}$. Since all four paths must be distinct, it follows that there must exist distinct child nodes of v where $x\sigma_{i_1} y_{i_1}$ and $x\sigma_{i_2} y_{i_2}$ diverge, and $x\sigma_{i_3} y_{i_3}$ and $x\sigma_{i_4} y_{i_4}$ diverge respectively. □

Lemma 11. *Assume suftree*(s) *satisfies the condition in Lemma 8. The number* ℓ *of leaves in suflinktree*(s) *is at most* $n_1 + 1$.

Proof. Let ℓ be the number of leaves in *suflinktree*(s). Since all leaves of *suftree*(s) are included in one path of suffix links and this path leads to the root, the maximum number of internal nodes v of *suflinktree*(s), such that $|children_l(v)| \geq 2$, is $\ell - 3$ excepting the root. From Lemma 10, this leads to $\ell - 3$ internal nodes with two internal-children in *suftree*(s). Therefore, $\ell \leq n_3 + 3 = n_1 + 1$ by Lemma 9. □

Lemma 12. *Assume suftree*(s) *satisfies the condition in Lemma 8. If* $\ell = n_3 + 3$, *then the longest internal node in group* n_3 *has at most* $\ell - 1$ *labels.*

Proof. It follows from Lemmas 9, 10, and 11. □

Lower Bound. Consider the following series of strings of length $n = \frac{3}{2}i(i+3) + 5$ where $i = 1, 2, 3, \ldots$:

$$\mathtt{X_i} = \mathtt{ab^iab^{i+1}ab^iab^1ab^iab^{i-1}ab^iab^2ab^iab^{i-2}a\cdots ab^kab^iab^{i-k}a\cdots ab^iab^{\lceil i/2\rceil}ab^iab^ia\$}$$

In what follows, we analyze $\mathtt{X_i}$ in terms of internal nodes of *suftree*$(\mathtt{X_i})$, shown in Figure 2. We consider the structure of *suftree*$(\mathtt{X_i})$ in Lemmas 14, 15, 16, 18, and the structure of *suflinktree*$(\mathtt{X_i})$ in Lemmas 19, 20, 21, 22. An important point in the lemmas is that if a substring p appears only once in the string, then there is no explicit internal node v such that $path(v) = p$, and the substring will correspond to a position on an edge leading to a leaf node (or the leaf node itself). Also, if $p\sigma_1$ and $p\sigma_2$ are both substrings of the string for distinct $\sigma_1, \sigma_2 \in \Sigma$, then there exists a node v such that $path(v) = p$.

Lemma 13. *suftree*$(\mathtt{X_i})$ *satisfies the condition in Lemma 8.*

Proof. Any occurrence of `a` in $\mathtt{X_i}$ is followed by either `b` or \$, and any occurrence of `b` in $\mathtt{X_i}$ is followed by either `a` or `b`. Moreover, \$ appears only at the end of $\mathtt{X_i}$. Thus, for any internal node v of *suftree*$(\mathtt{X_i})$, we have $|children_s(v)| = 2$. □

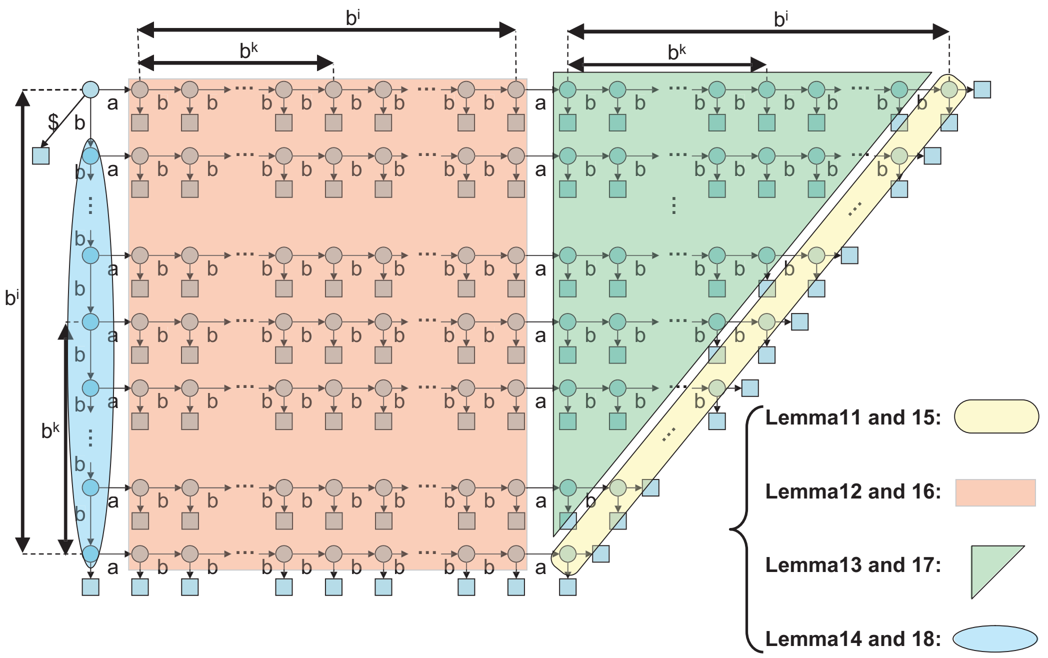


Figure 2. Illustration for *suftree*($\mathtt{X_i}$). The four groups of the internal nodes are dealt in Lemmas 14, 15, 16, and 18, respectively.

Lemma 14. *For any k $(0 \le k \le i)$, there exists an internal node corresponding to* $\mathtt{b^{i-k}ab^iab^k}$*, and this node belongs to group n_1 of suftree*($\mathtt{X_i}$)*.*

Proof. We have three cases to consider:

- When $k = 0$.
  Since there are two strings $\mathtt{b^iab^iab}$ and $\mathtt{b^iab^ia\$}$ in $\mathtt{X_i}$, there is an internal node for $\mathtt{b^iab^ia}$. Since there is no other occurrence of $\mathtt{b^iab^ia}$, the two children of this node are both a leaf node and thus it belongs to group n_1.
- When $0 < k < i$.
  Consider a substring $\mathtt{Y_i}$ of $\mathtt{X_i}$ such that

$$\mathtt{Y_i} = \mathtt{ab^{k-1}ab^ia}\underline{\mathtt{b^{i-k+1}ab^iab^k}}\mathtt{ab^ia}\underline{\mathtt{b^{i-k}ab^iab^{k+1}}}\mathtt{ab^ia}.$$

  Then, $\mathtt{b^{i-k}ab^iab^k}$ appears twice in this substring, as underlined.
  Now we show that each of $\mathtt{b^{i-k}ab^iab^ka}$ and $\mathtt{b^{i-k}ab^iab^kb}$ appears only once in $\mathtt{X_i}$. For $\mathtt{b^{i-k}ab^iab^ka}$, it is clear because $\mathtt{ab^ka}$ appears exactly once in $\mathtt{X_i}$. String $\mathtt{b^{i-k}ab^iab^kb} = \mathtt{b^{i-k}ab^iab^{k+1}}$ appears in $\mathtt{Y_i}$ (the second underlined part). This is the only occurrence of $\mathtt{b^{i-k}ab^iab^{k+1}}$ in $\mathtt{X_i}$, because the prefix $\mathtt{b^{i-k}a}$ would appear in substrings $\mathtt{ab^{i-k+x}ab^iab^{k-x}a}$ with $x \ge 0$, but then the suffix $\mathtt{b^iab^{k+1}}$ cannot match.
- When $k = i$.
  By similar discussions to the case that $k = 0$. ☐

Lemma 15. *For any x $(0 \le x \le i)$ and y $(0 \le y \le i)$, there exists an internal node corresponding to* $\mathtt{b^xab^y}$*, and this node belongs to group n_2.*

Proof. We have three cases to consider:

– When $y = 0$.
Since there are more than one occurrences of $\mathtt{b^iab}$, there are more than one occurrences of $\mathtt{b^xab}$. In addition, since there is exactly one occurrence of $\mathtt{b^ia\$}$, there is exactly one occurrence of $\mathtt{b^xa\$}$.
– When $0 < y < i$.
Since there are more than one occurrences of $\mathtt{b^xab^i}$, there are more than one occurrences of $\mathtt{b^xab^{y+1}}$. In addition, since there is exactly one occurrence of $\mathtt{b^iab^ya}$ for each $0 < y < i$, $\mathtt{b^xab^ya}$ appears exactly once.
– When $y = i$.
There is exactly one occurrence of $\mathtt{b^xab^ia}$ for each $0 \le x \le i$. Since there is exactly one occurrence of $\mathtt{b^iab^{i+1}}$, $\mathtt{b^xab^{i+1}}$ appears exactly once for each $0 \le x \le i$.

□

Lemma 16. *For any y ($0 \le y < k$), where $1 \le k \le i$, there exists an internal node corresponding to $\mathtt{b^{i-k}ab^iab^y}$, and this node belongs to group n_2.*

Proof. By similar arguments to Lemmas 14 and 15. □

The next corollary follows Lemmas 15 and 16.

Corollary 17. *For any k ($0 \le k \le i$), let $path(u) = \mathtt{b^{i-k}a}$ and $path(v) = \mathtt{b^{i-k}ab^iab^k}$. Then, the path from u to v contains $i + k + 2$ internal nodes (including u and v).*

Lemma 18. *For any x ($0 < x \le i$), there is an internal node corresponding to $\mathtt{b^x}$. The node for $\mathtt{b^i}$ belongs to group n_2, and the other nodes for $\mathtt{b^x}$ with $0 < x < i$ belong to group n_3.*

Proof. Since $\mathtt{b^ia}$ and $\mathtt{b^{i+1}}$ appear in $\mathtt{X_i}$, there is an internal node for $\mathtt{b^x}$ for any $0 < x \le i$. Since $\mathtt{b^ia}$ appears more than once and $\mathtt{b^{i+1}}$ appears exactly once, the node for $\mathtt{b^i}$ belongs to group n_2. All the nodes for $\mathtt{b^x}$ with $0 < x < i$ belong to group n_3, since both $\mathtt{b^xa}$ and $\mathtt{b^{x+1}}$ appear more than once. □

From here on, we consider the suffix links of *suftree*($\mathtt{X_i}$).

Lemma 19. *For any k ($0 \le k \le i$), let v be the internal node such that $path(v) = \mathtt{b^{i-k}ab^iab^k}$. Then we have $|children_l(v)| = 0$.*

Proof. For contrary, assume that $|children_l(v)| \ge 1$. Then there must exist a node corresponding to either $\mathtt{ab^{i-k}ab^iab^k}$ or $\mathtt{b^{i-k+1}ab^iab^k}$. However, we show that these strings can appear at most once in $\mathtt{X_i}$.

First, we consider $\mathtt{ab^{i-k}ab^iab^k}$:

– When $k = 0$. $\mathtt{ab^iab^ia}$ appears only once (at the end of $\mathtt{X_i}$).
– When $0 < k < i$. Prefix $\mathtt{ab^{i-k}a}$ appears only once in $\mathtt{X_i}$.
– When $k = i$. Prefix $\mathtt{aa}$ never appears in $\mathtt{X_i}$.

Now let us consider $\mathtt{b^{i-k+1}ab^iab^k}$. By Lemma 14, there is an internal node corresponding to $\mathtt{b^{i-k+1}ab^iab^{k-1}}$ and this node belongs to group n_1. This implies that $\mathtt{b^{i-k+1}ab^iab^k}$ can appear at most once in $\mathtt{X_i}$. □

The above lemma implies that the internal nodes v such that $path(v) = \mathtt{b^{i-k}ab^iab^k}$ are the leaves of *suflinktree*($\mathtt{X_i}$). See also the upper diagram in Figure 3.

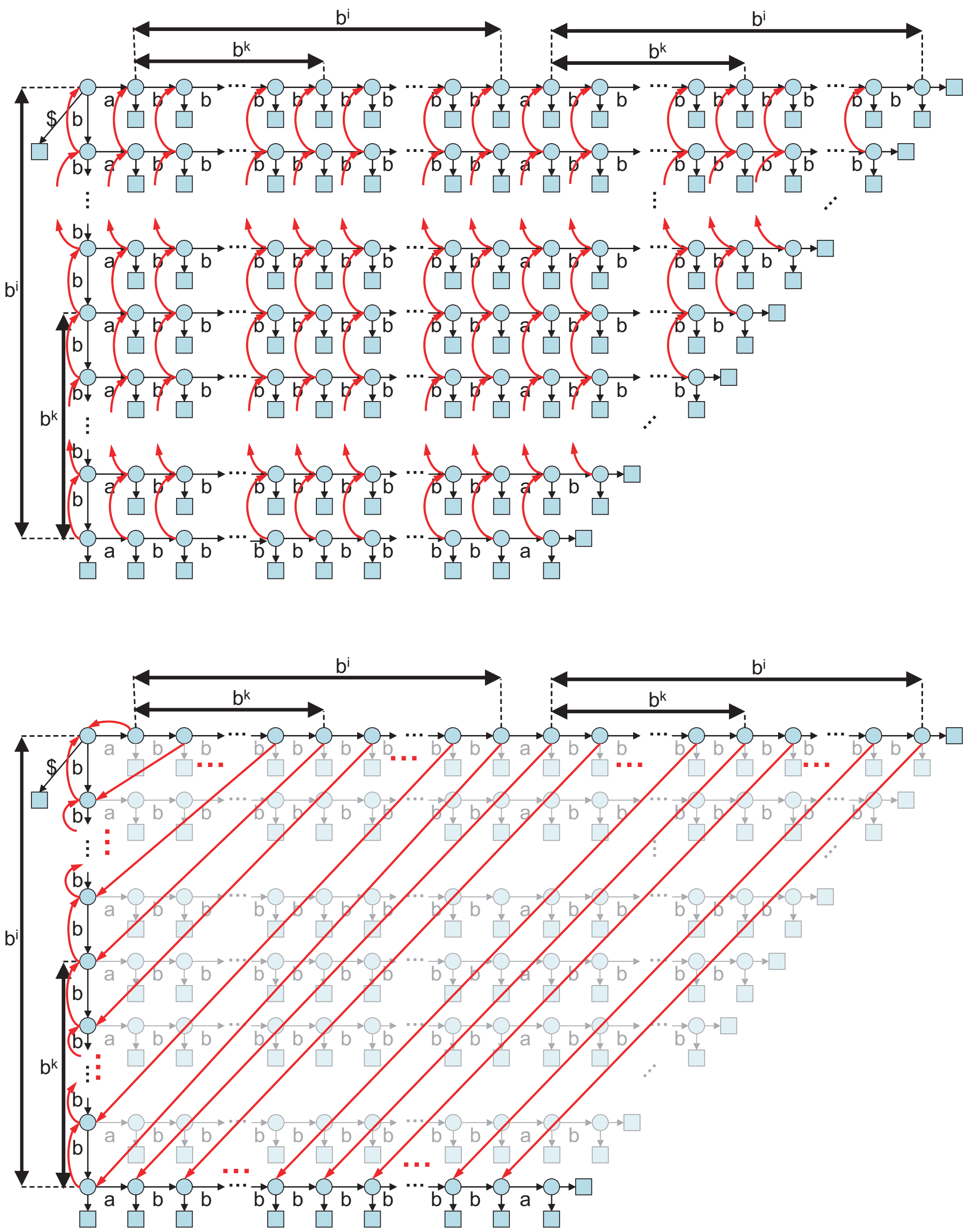


Figure 3. Illustration for the suffix links of *suftree*($\mathtt{X_i}$). For the sake of visibility, the suffix links of $\mathtt{X_i}$ are shown in two rounds. Moreover, for simplicity, the suffix links for the leaves are omitted here.

Lemma 20. *For any internal node $v_{x,y}$ such that $path(v_{x,y}) = \mathtt{b}^{\mathtt{x}}\mathtt{ab}^{\mathtt{y}}$ $(0 \le x \le i$ and $0 \le y \le i)$, we have $|children_l(v_{x,y})| = 1$.*

Proof. We have three cases to consider:

- When $x = 0$.
  There is no occurrence of $\mathtt{aab}^{\mathtt{y}}$, and we have two distinct occurrences of $\mathtt{b}^{\mathtt{i}}\mathtt{ab}^{\mathtt{i}}$ in $\mathtt{X_i}$. Thus we have $|children_l(v_{0,y})| = 1$.
- When $0 < x < i$.
  $\mathtt{ab}^{\mathtt{x}}\mathtt{ab}^{\mathtt{y}}$ appears only once, since there is only one occurrence of $\mathtt{ab}^{\mathtt{x}}\mathtt{ab}^{\mathtt{i}}$. Because

$\mathtt{b}^{\mathtt{i}}\mathtt{ab}^{\mathtt{y}}$ appears more than once, there are more than one occurrence of $\mathtt{b}^{\mathtt{x}}\mathtt{ab}^{\mathtt{y}}$. Thus we have $|children_l(v_{x,y})| = 1$.

– When $x = i$.
We have at least two occurrences of $\mathtt{ab}^{\mathtt{i}}\mathtt{ab}^{\mathtt{y}}$. Since $\mathtt{b}^{\mathtt{i+1}}$ appears only once, there is only one occurrence of $\mathtt{b}^{\mathtt{i+1}}\mathtt{ab}^{\mathtt{y}}$. Thus we have $|children_l(v_{i,y})| = 1$.

□

Lemma 21. *For any internal node v_z such that $path(v_z) = \mathtt{b}^{\mathtt{i-k}}\mathtt{ab}^{\mathtt{i}}\mathtt{ab}^{\mathtt{z}}$ $(0 \le z < k)$, where $1 \le k \le i$, we have $|children_l(v_z)| = 1$.*

Proof. By similar arguments to Lemma 20. □

Lemma 22. *For any internal node v_x such that $path(v_x) = \mathtt{b}^{\mathtt{x}}$ $(0 < x < i)$, we have $|children_l(v_x)| = 2$. In addition, for the node v_i such that $path(v_i) = \mathtt{b}^{\mathtt{i}}$, we have $|children_l(v_i)| = 1$.*

Proof. Due to Lemma 15, there exist internal nodes u_x such that $path(u_x) = \mathtt{ab}^{\mathtt{x}}$ for each $0 < x \le i$, and therefore have $(u_x, v_x) \in E_l$. Due to Lemma 18, we have $(v_{x+1}, v_x) \in E_l$ for $0 < x < i$. For $x = i$, however, we have $(v_{i+1}, v_i) \notin E_l$ because $\mathtt{b}^{\mathtt{i+1}}$ does not correspond to a node (since it occurs only once in the string). □

From the above lemmas, only the nodes corresponding to $\mathtt{b}^{\mathtt{x}}$ $(0 < x < i)$ are of in-degree two in *suflinktree*($\mathtt{X_i}$). Plus, only the root node is of in-degree three in *suflinktree*($\mathtt{X_i}$). See also Figure 3 for these observations.

The number of nodes covered in these lemmas are as follows:

Lemma	11 or 15	12 or 16	13 or 17	14 or 18
The number of nodes	$i+1$	$(i+1)^2$	$i(i+1)/2$	i

the sum of these nodes is $\frac{3}{2}i^2 + \frac{9}{2}i + 2 = n - 3$, which indicates that we have discussed all nodes of the *ST*-graph for the string $\mathtt{X_i}$.

Now we are finally ready to show the lower bound on the number of interval labels.

Theorem 23. *The number of labels assigned to a suffix tree by Algorithm 1 is $\Omega(n\sqrt{n})$.*

Proof. From Lemmas 7, 12, 13, 14, 15, 16, 18, 19, 20, 21, 22, Corollary 17, we have $\ell = i + 2$ and achieve the following equations on the total number of interval labels that are assigned to *suftree*($\mathtt{X_i}$):

$$\begin{aligned}
&(2+3+\cdots+\ell-1)\times(i+1)+\ell\times\Big\{1+\sum_{k=0}^{i}\{(i+k+2)-(\ell-2)\}\Big\}\\
&+\{(\ell-1)+(\ell-2)+\cdots 3\}+1\times(n+1)\\
=&\{2+3+\cdots+(i+1)\}\times(i+1)+(i+2)\times\{1+\sum_{k=0}^{i}(k+2)\}\\
&+\{(i+1)+i+\cdots 3\}+\frac{3i(i+3)}{2}+6\\
=&\frac{i(i+3)(i+1)}{2}+\frac{(i+2)\{2+i(i+1)+4(i+1)\}}{2}+\frac{(i-1)(i+4)}{2}+\frac{3i(i+3)}{2}+6\\
=&\frac{2i^3+15i^2+31i+20}{2}. \qquad (1)
\end{aligned}$$

Since $n = \frac{3}{2}i(i+3)+5$ and $i \geq 1$, we have

$$i = \frac{\sqrt{24n-39}}{6} - \frac{3}{2}.$$

By substituting this for the i's in Equation (1), the total number of interval labels assigned to *suftree*($\mathtt{X_i}$) is shown to be $\Theta(n\sqrt{n})$. □

The worst case upper bound for Algorithm 2 is $O(\log n)$: From the argument in Lemma 4, the labels at each node can be stored as sorted arrays. Also, the maximum number of labels at each node is bounded by $O(n)$ (Lemma 7). Therefore, line 2 in Algorithm 2 can be run in $O(\log n)$ time using a standard binary search on the label array.

5 Computational Experiments

We exhaustively enumerated all strings of length $n \leq 33$ consisting of $\{\mathtt{a}, \mathtt{b}\}$ and ending with \$, and applied Algorithm 1 to each string. For each n, the number of labels in the worst case was recorded. The results are shown in Table 2.

n	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
#labels	3	6	9	12	15	18	22	26	30	34	39	44	49	54	59	64
n	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33
#labels	69	74	79	85	91	97	103	109	115	121	127	133	139	145	151	158

Table 2. The maximum number of labels that is assigned by Algorithm 1 to any string of length n.

We note that for $n \leq 7$, the worst case corresponds to the string $\mathtt{a}^{\mathtt{n-1}}\$$, and therefore the total number of labels is $3(n-1)$. For $n = 11, 20, 32$, the worst case instances contain $\mathtt{X}_1$, $\mathtt{X}_2$, $\mathtt{X}_3$, and the total number of labels is as calculated in Theorem 23. Generally for $7 \leq n \leq 33$, we found that the total number of labels in the worst case can be written exactly with the following formula:

$$\mathbf{max}\{f(\lfloor\sqrt{(2n-3)/3}\rfloor), f(\lceil\sqrt{(2n-3)/3}\rceil)\}$$

where $f(k) = (k+2)n - k(k^2+3)/2 - 3$. We have also confirmed for smaller n and with larger sized alphabets, the worst case instances will only contain one type of character (excluding \$) for $n \leq 7$, and two types of characters (excluding \$) for $n \geq 7$, therefore corresponding to the instances given above. (At $n = 7$, both types had equal worst case label size of 18.) This seems natural since Lemma 8 indicates that the more types of characters used, the less number of nodes there are in the ST-graph.

Although we have not been able to give a rigorous proof for an upperbound of $O(n\sqrt{n})$, the above results strongly suggest this bound.

6 Discussion

We presented an algorithm that can processes an ST-graph for a string of length n, so that reachability queries between arbitrary pairs of nodes can be answered in $O(\log n)$

time. The expected time and space complexity of the preprocessing algorithm for a random string is $O(n \log n)$. We also presented a series of strings for which the algorithm requires $\Theta(n\sqrt{n})$ time and space for preprocessing. Exhaustive computational search for $n \leq 33$ showed that the strings of the series also achieve the worst case instances of the algorithm. Although we have not been able to give a direct proof, this provides strong evidence that the worst case time complexity of the algorithm is also $\Theta(n\sqrt{n})$.

Since a suffix tree can have a height of $O(n)$, a naïve consideration of Lemma 5 only gives an $O(n^2)$ bound for the number of labels for an ST-graph of a suffix tree of height $O(n)$, rather than $O(n\sqrt{n})$. There seems to be a delicate tradeoff between deep paths in the suffix tree and deep paths in the suffix link tree. For example, the suffix tree for string $\mathtt{a}^\mathtt{n}\$$ will have a path of depth $O(n)$. However, the number of total labels in this case is also limited to $O(n)$, since there are only two leaves in the suffix link tree, which bounds the number of labels for each node to 2 (Lemma 7). There also exists strings $\mathtt{a}^\mathtt{k}\underbrace{\mathtt{bbccddeeff...}}_{\mathtt{k}\text{ character types}}\$$ $(n = 3k+1)$, where their suffix trees have a path of depth $O(n)$, and their suffix link trees have $O(n)$ leaves. However, the total number of labels in this case is also $O(n)$, since all but two of the leaves in the suffix link tree are very shallow.

6.1 Open Problems

There are several open problems that are of interest concerning reachability queries on ST-graphs.

1. Whether or not there exists an algorithm which can do better: $O(n\sqrt{n})$ preprocessing and $O(1)$ query, or ultimately $O(n)$ preprocessing and $O(1)$ query.
2. Whether or not we can simulate reachability queries on suffix *trie* graphs. All nodes in the suffix tree correspond to a substring of the original string. However, there can exist substrings in the string without a corresponding explicit node. An implicit node of a suffix tree is a position in the suffix tree which ends somewhere in the middle of an edge. So far we have considered reachability queries between explicit nodes of the ST-graph. Although reachability between implicit nodes of the ST-graph is straightforward with respect to the ST-graph, the result does not correspond to the substring relation between implicit nodes, as it does for explicit nodes shown in Lemma 2.

Acknowledgments

This work was supported in part by The Ministry of Education, Culture, Sports, Science and Technology, Grant-in-Aid for Young Scientists (B).

References

1. R. Agrawal, A. Borgida, and H. V. Jagadish: *Efficient management of transitive relationships in large data and knowledge bases*, in Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data, 1989, pp. 253–262.
2. A. Apostolico and W. Szpankowski: *Self-alignments in words and their applications*. Journal of Algorithms, 13(3) 1992, pp. 446–467.
3. E. Cohen, E. Halperin, H. Kaplan, and U. Zwick: *Reachability and distance queries via 2-hop labels*, in Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms, 2002, pp. 937–946.
4. D. Gusfield: *Algorithms on Strings, Trees, and Sequences*, Cambridge University Press, 1997.
5. T. Kameda: *On the vector representation of the reachability in planar directed graphs*. Information Processing Letters, 3(3) 1975, pp. 75–77.
6. E. M. McCreight: *A space-economical suffix tree construction algorithm*. J. ACM, 23(2) 1976, pp. 262–272.
7. M. Talamo and P. Vocca: *A data structure for lattice representation*. Theoretical Computer Science, 175(2) 1997, pp. 373–392.
8. E. Ukkonen: *On-line construction of suffix trees*. Algorithmica, 14(3) 1995, pp. 249–260.
9. H. Wang, H. He, J. Yang, P. S. Yu, and J. X. Yu: *Dual labeling: answering graph reachability queries in constant time*, in Proc. 22nd International Conference on Data Engineering, 2006, to appear.
10. P. Weiner: *Linear pattern matching algorithms*, in Proc. 14th IEEE Annual Symp. on Switching and Automata Theory, 1973, pp. 1–11.

FM-KZ: An Even Simpler Alphabet-Independent FM-Index

Rafał Przywarski[1], Szymon Grabowski[1], Gonzalo Navarro[2], and Alejandro Salinger[3]

[1] Computer Engineering Dept., Tech. Univ. of Łódź, Poland.
sgrabow@kis.p.lodz.pl
[2] Dept. of Computer Science, Univ. of Chile, Chile.
[3] David R. Cheriton School of Computer Science, University of Waterloo, Canada.

Abstract. In an earlier work [6] we presented a simple FM-index variant, based on the idea of Huffman-compressing the text and then applying the Burrows-Wheeler transform over it. The main drawback of using Huffman was its lack of synchronizing properties, forcing us to supply another bit stream indicating the Huffman codeword boundaries. In this way, the resulting index needed $O(n(H_0+1))$ bits of space but with the constant 2 (concerning the main term). There are several options aiming to mitigate the overhead in space, with various effects on the query handling speed. In this work we propose Kautz-Zeckendorf coding as a both simple and practical replacement for Huffman. We dub the new index FM-KZ. We also present an efficient implementation of the rank operation, which is the main building brick of the FM-KZ. Experimental results show that our index provides an attractive space/time tradeoff in comparison with existing succinct data structures, and in the DNA test it even wins both in search time and space use. An additional asset of our solution is its relative simplicity.

1 Introduction

A *full-text index* is a data structure that enables to determine the *occ* occurrences of a short pattern $P = p_1p_2 \ldots p_m$ in a large text $T = t_1t_2 \ldots t_n$ without a need of scanning over the whole text T. Text and pattern are sequences of characters over an alphabet Σ of size σ. The pattern may appear at any position in T, and its length is also arbitrary. In practice one wants to know not only the value *occ*, i.e., how many times the pattern appears in the text (*counting query*) but also the text positions of those *occ* occurrences (*reporting query*, and usually also a text context around them (*display query*).

Classic full-text indexes, albeit powerful and versatile, need space several times greater than the text itself. Hence, a natural interest in *succinct* full-text indexes has been observed in recent years. A comprehensive survey of existing techniques in this very active research area can be found in [13].

A truly exciting perspective has been originated in the work of Ferragina and Manzini [3]; they showed a full-text index may discard the original text, as it contains enough information to recover the text. We denote a structure with such a property with the term *self-index*.

The FM-index of Ferragina and Manzini [3] was the first self-index with space complexity expressed in terms of kth order (empirical) entropy and pattern search time linear only in the pattern length. Its space complexity, however, contains an exponential dependence on the alphabet size; a weakness eliminated in a practical implementation [4] for the price of not achieving the optimal search time anymore. Therefore, it has been interesting both from the point of theory and practice to construct an index with nicely bound both space and time complexities, preferably with no (or mild) dependence on the alphabet size.

The large alphabet dependence of the original FM-index shows up not only in the space usage, but also in the time to show an occurrence position and display text substrings. The FM-index needs up to $5H_k n + O\left((\sigma \log \sigma + \log \log n)\frac{n}{\log n} + n^\gamma \sigma^{\sigma+1}\right)$ bits of space, where $0 < \gamma < 1$. The time to search for a pattern and obtain the number of its occurrences in the text is the optimal $O(m)$. The text position of each occurrence can be found in $O\left(\sigma \ \log^{1+\varepsilon} n\right)$ time, for some $\varepsilon > 0$ that appears in the sublinear terms of the space complexity. Finally, the time to display a text substring of length L is $O\left(\sigma \ (L + \log^{1+\varepsilon} n)\right)$. The last operation is important not only to show a text context around each occurrence, but also because a self-index replaces the text and hence it must provide the functionality of retrieving any desired text substring.

One of the proposals to eliminate an exponential dependence on the alphabet size was Huffman FM-index [6]: It was based on the *backward search* idea of [4] but the novelty was to Huffman encode the text (and the pattern) so as to reduce the alphabet to binary. As a result, any dependence on the alphabet size was removed. We showed that our index can operate using $n(2H_0 + 3 + \varepsilon)(1 + o(1))$ bits, for any $\varepsilon > 0$. No alphabet dependence is hidden in the sublinear terms.

At search time, our index finds the number of occurrences of the pattern in $O(m(H_0 + 1))$ average time. The text position of each occurrence can be reported in worst case time $O\left(\frac{1}{\varepsilon}(H_0 + 1) \ \log n\right)$. Any text substring of length L can be displayed in $O\left((H_0 + 1) \ L\right)$ average time, in addition to the mentioned worst case time to find a text position.

Since the original presentation, its implementation has been optimized and also a variant with 4-ary Huffman has been checked [7]. Albeit not among the most succinct, the 4-ary Huffman FM-index appears to be among the fastest and thus practical indices.

In this paper we present an alternative to Huffman coding variants. Instead, we use Kautz-Zeckendorf coding [9,15], capable of instant detection of codeword boundaries. To give the flavor of this idea, we note that in its basic variant, the Kautz-Zeckendorf code has no codeword with any two adjacent 1's.

2 The FM-index Structure

The FM-index [3] is based on the *Burrows-Wheeler transform (BWT)* [1], which produces a permutation of the original text, denoted by $T^{bwt} = bwt(T)$. String T^{bwt} is a result of the following *forward* transformation: (1) Append to the end of T a special end marker \$, which is lexicographically smaller than any other character; (2) form a *conceptual* matrix $\mathcal{M}$ whose rows are the cyclic shifts of the string $T\$$, sorted in lexicographic order; (3) construct the transformed text L by taking the last column of $\mathcal{M}$. The first column is denoted by F.

The *suffix array (SA)* $\mathcal{A}$ of text $T\$$ is essentially the matrix $\mathcal{M}$: $\mathcal{A}[i] = j$ iff the ith row of $\mathcal{M}$ contains string $t_j t_{j+1} \cdots t_n \$ t_1 \cdots t_{j-1}$. Given the suffix array, the search for the occurrences of the pattern $P = p_1 p_2 \cdots p_m$ is trivial. The occurrences form an interval $[sp, ep]$ in $\mathcal{A}$ such that suffixes $t_{\mathcal{A}[i]} t_{\mathcal{A}[i]+1} \cdots t_n$, $sp \leq i \leq ep$, contain the pattern as a prefix. This interval can be searched for by using two binary searches in time $O(m \log n)$.

The suffix array of text T is represented implicitly by T^{bwt}. The novel idea of the FM-index is to store T^{bwt} in compressed form, and to simulate the search in the suffix array. To describe the search algorithm, we need to introduce the *backward* BWT that produces T given T^{bwt}:

1. Compute the array $C[1 \dots \sigma]$ storing in $C[c]$ the number of occurrences of characters $\{\$, 1, \dots, c-1\}$ in the text T. Notice that $C[c]+1$ is the position of the first occurrence of c in F (if any).
2. Define the *LF-mapping* $LF[1 \dots n+1]$ as $LF[i] = C[L[i]] + Occ(L, L[i], i)$, where $Occ(X, c, i)$ equals the number of occurrences of character c in the prefix $X[1, i]$.
3. Reconstruct T backwards as follows: set $s = 1$ and $T[n] = L[1]$ (because $\mathcal{M}[1] = \$T$); then, for each $n-1, \dots, 1$ do $s \leftarrow LF[s]$ and $T[i] \leftarrow L[s]$.

We are now ready to describe the search algorithm given in [3] (Fig. 1). It finds the interval of $\mathcal{A}$ containing the occurrences of the pattern P. It uses the array C and function $Occ(X, c, i)$ defined above. Using the properties of the backward BWT, it is easy to see that the algorithm maintains the following invariant [3]: *At the ith phase, the variable sp points to the first row of $\mathcal{M}$ prefixed by $P[i, m]$ and the variable ep points to the last row of $\mathcal{M}$ prefixed by $P[i, m]$*. The correctness of the algorithm follows from this observation.

Algorithm FM_Search(P,T^{bwt})

```
i = m;
sp = 1; ep = n;
while ((sp ≤ ep) and (i ≥ 1) do
      c = P[i];
      sp = C[c] + Occ(T^bwt, c, sp − 1)+1;
      ep = C[c] + Occ(T^bwt, c, ep);
      i = i − 1;
if (ep < sp) then return "not found" else return "found (ep − sp + 1) occs".
```

Figure 1. Algorithm for counting the number of occurrences of $P[1 \dots m]$ in $T[1 \dots n]$

Ferragina and Manzini [3] describe an implementation of $Occ(T^{bwt}, c, i)$ that uses a compressed form of T^{bwt}. They show how to compute $Occ(T^{bwt}, c, i)$ for any c and i in constant time. However, to achieve this they need exponential space (in the size of the alphabet). In a practical implementation [4] this was avoided, but the constant time guarantee for answering $Occ(T^{bwt}, c, i)$ was no longer valid.

The FM-index can also show the text positions where P occurs, and display any text substring. The details are deferred to Section 5.

3 Rank and Select Queries on Bit Arrays

A crucial building block we use is a data structure to perform *rank* operations over a bit array. Given a bit sequence $B[1 \dots n]$, $rank(B, i)$ is the number of 1's in $B[1 \dots i]$, $rank(B, 0) = 0$. This function can be computed in constant time using only $o(n)$ extra bits [8,11,2]. The solution, as well as its more practical implementation variants, are described in [5]; here we present a novel implementation, which seems to be fastest in practice.

For an input bit array B of size n and a given parameter bs we create a lookup table N with $\lceil n/2^{bs} \rceil$ entries. Namely, for each $k = 0 \ldots \lfloor n/2^{bs} \rfloor - 1$ we compute: $N[k] = rank(B, (k{+}1){*}2^{bs})$. If $\lceil n/2^{bs} \rceil > \lfloor n/2^{bs} \rfloor$, then we also compute: $N[\lfloor n/2^{bs} \rfloor] = rank(B, n)$. The above structure needs $32 * \lceil n/2^{bs} \rceil = O(n)$ bits, where the constant 32 is the number of bits per entry of N.

Now, we calculate $rank(B, i)$ as follows.
If $i < 2^{bs}$, then $rank(B, i) = popcount(B, 0 \ldots i)$. Otherwise, $rank(B, i) = N[\lfloor i/2^{bs} \rfloor - 1] + popcount(B, (\lfloor i/2^{bs} \rfloor * 2^{bs}) \ldots i)$. The operation $popcount(B, a \ldots b)$ returns the number of set bits in the interval $B[a \ldots b]$, $a \leq b$, making use of a precomputed table. As long as the interval width is on the order of machine word, this is a constant time operation.

Sometimes we need to calculate the inverse function, $select(B, j)$, which gives the position of the j-th bit set in B. It can also be implemented in constant time using $o(n)$ additional space [8,11,2]. More practical implementations exist [5], but it is always significantly slower than $rank$, and also more rarely needed.

4 First Huffman, then Burrows-Wheeler

We focus now on our index representation, starting from the original variant. Imagine that we compress our text $T\$$ using Huffman. The resulting bit stream will be of length $n' < (H_0 + 1)n$, since (binary) Huffman poses a maximum representation overhead of 1 bit per symbol[4]. Let us call T' this sequence. Let us also define a second bit array Th, of the same length of T', such that $Th[i] = 1$ iff i is the starting position of a Huffman codeword in T'. Th is also of length n'. (We will not, however, represent T' nor Th in our index.)

The idea is to search the binary text T' instead of the original text T. Let us apply the Burrows-Wheeler transform over text T', so as to obtain $B = (T')^{bwt}$. The terminator character, "\$", is excluded from T' so as to have a binary alphabet.

More precisely, let $\mathcal{A}'[1 \ldots n']$ be the suffix array for text T', that is, a permutation of the set $1 \ldots n'$ such that $T'[A'[i] \ldots n'] < T'[A'[i+1] \ldots n']$ in lexicographic order, for all $1 \leq i < n'$. In a lexicographic comparison, if a string x is a prefix of y, assume $x < y$. Suffix array $\mathcal{A}'$ will not be explicitly represented. Rather, we represent bit array $B[1 \ldots n']$, such that $B[i] = T'[\mathcal{A}'[i] - 1]$ (except that $B[i] = T[n']$ if $\mathcal{A}'[i] = 1$). We also represent another bit array $Bh[1 \ldots n']$, such that $Bh[i] = Th[\mathcal{A}'[i]]$. This tells whether position i in $\mathcal{A}'$ points to the beginning of a codeword.

Our goal is to search B exactly like the FM-index. For this sake we need array C and function Occ. Since the alphabet is binary, however, Occ can be easily computed: $Occ(B, 1, i) = rank(B, i)$ and $Occ(B, 0, i) = i - rank(B, i)$. Also, array C is so simple for the binary text that we can do without it: $C[0] = 0$ and $C[1] = n' - rank(B, n')$, that is, the number of zeros in B (of course value $n' - rank(B, n')$ should be precomputed in practice). Therefore, $C[c] + Occ(T^{bwt}, c, i)$ is replaced in our index by $i - rank(B, i)$ if $c = 0$ and $n' - rank(B, n') + rank(B, i)$ if $c = 1$.

There is a small twist, however, due to the fact that we are not putting a terminator to our binary sequence T' and hence no terminator appears in B. Let us call "#" the terminator of the binary sequence so that it is not confused with the terminator "\$" of $T\$$. In the position $p_\#$ such that $\mathcal{A}'[p_\#] = 1$, we should have $B[p_\#] = \#$.

[4] Note that these n and H_0 refer to $T\$$, not T. However, the difference between both is only $O(\log n)$, and will be absorbed by the $o(n)$ terms that will appear later.

Instead, we are setting $B[p_\#]$ to the last bit of T'. This is the last bit of the Huffman codeword assigned to the terminator "\$" of $T\$$. Since we can freely switch left and right siblings in the Huffman code, we will ensure that this last bit is zero. Hence the correct B sequence would be of length $n'+1$, starting with 0 (which corresponds to $T'[n']$, the character preceding the occurrence of "#", since $\# < 0 < 1$), and it would have $B[p_\#] = \#$. To obtain the right mapping to our binary B, we must correct $C[0] + Occ(B, 0, i) = i - rank(B, i) + [i < p_\#]$, that is, add 1 to the original value when $i < p_\#$. The computation of $C[1] + Occ(B, 1, i)$ remains unchanged.

Therefore, by preprocessing B to solve *rank* queries, we can search B exactly as the FM-index. The extra space required by the *rank* structure is $o(H_0 n)$, without any dependence on the alphabet size. Overall, we have used at most $n(2H_0 + 2)(1 + o(1))$ bits for our representation. This will grow slightly in the next sections due to additional requirements.

Our search pattern is not the original P, but its binary coding P' using the Huffman code we applied to T. Converting P to P' takes $O(m)$ time. If we assume that the characters in P have the same distribution of T, then the length of P' is $< m(H_0 + 1)$. This is the number of steps to search B using the FM-index search algorithm.

The answer to that search, however, is different from that of the search of T for P. The reason is that the search of T' for P' returns the number of suffixes of T' that start with P'. Certainly these include the suffixes of T that start with P, but also other superfluous occurrences may appear. These correspond to suffixes of T' that do not start a Huffman codeword, yet they start with P'.

This is the reason why we have marked the suffixes that start a Huffman codeword in Bh. In the range $[sp, ep]$ found by the search for P' in B, every bit set in $Bh[sp \dots ep]$ represents a true occurrence. Hence the true number of occurrences can be computed as $rank(Bh, ep) - rank(Bh, sp - 1)$.

Figure 2 depicts the search algorithm.

Algorithm Huff-FM_Search(P',B,Bh)
(1) $i = m'$;
(2) $sp = 1$; $ep = n'$;
(3) **while** $((sp \le ep)$ **and** $(i \ge 1))$ **do**
(4) **if** $P'[i] = 0$ **then**
$sp = (sp - 1) - rank(B, sp - 1) + 1 + 1 - [sp - 1 \ge p_\#]$;
$ep = ep - rank(B, ep) + 1 - [ep \ge p_\#]$;
else $sp = n' - rank(B, n') + rank(B, sp - 1) + 1$;
$ep = n' - rank(B, n') + rank(B, ep)$;
(7) $i = i - 1$;
(8) **if** $ep < sp$ **then** $occ = 0$ **else** $occ = rank(Bh, ep) - rank(Bh, sp - 1)$;
(9) **if** $occ = 0$ **then return** "not found" **else return** "found (occ) occs".

Figure 2. Algorithm for counting the number of occurrences of $P'[1 \dots m']$ in $T'[1 \dots n']$

Therefore, the search complexity is $O(m(H_0 + 1))$, assuming that the zero-order distributions of P and T are similar. It is well-known that the longest Huffman codeword does not exceed $O(m \log n)$ bits. From this we immediately obtain the worst case search cost of $O(m \log n)$ for our index. This matches the worst case search time of the compressed suffix array (CSA) of Sadakane [14]. An exceptional situation occurs when P contains a character not present in T. This is easier, however, as we immediately know that P does not occur in T.

Is it in fact possible to achieve $O(m \log n)$ search complexity also for the worst case, for the price of $2n$ extra bits. Basically, the idea is to use a length-limited Huffman coding variant but we omit the details and analysis due to lack of space. This idea, however, does not have much importance in practice because extremely skew symbol distributions almost never happen and thus optimizing the worst case is hardly worth any effort.

5 Reporting Occurrences and Displaying the Text

Up to now we have focused on the search time, that is, the time to determine the suffix array interval containing all the occurrences. In practice, one needs also the text positions where they appear, as well as a text context. Since self-indexes replace the text, in general one needs to extract any text substring from the index.

Given the suffix array interval that contains the *occ* occurrences found, the FM-index reports each such position in $O(\sigma \log^{1+\varepsilon} n)$ time, for any $\varepsilon > 0$ (which appears in the sublinear space component). The CSA can report each in $O(\log^{\varepsilon} n)$ time, where ε is paid in the nH_0/ε space. Similarly, a text substring of length L can be displayed in time $O(\sigma(L + \log^{1+\varepsilon} n))$ by the FM-index and $O(L + \log^{\varepsilon} n)$ by the CSA.

Our index can do better than the FM-index in this respect, although not as well as the CSA. Using $(1+\varepsilon)n$ additional bits, we can report each occurrence position in $O(\frac{1}{\varepsilon}(H_0 + 1) \log n)$ time and display a text context in time $O(L \log \sigma + \log n)$ in addition to the time to find an occurrence position. On average, assuming that random text positions are involved, the overall complexity to display a text interval becomes $O((H_0+1)(L+\frac{1}{\varepsilon}\log n))$. Those complexities hold for all the variants of our solution: based on the binary or higher arity Huffman, or on the Kautz-Zeckendorf coding. Still, the overall idea of reporting and displaying via sampling sorted suffixes at regular intervals was first presented in the seminal work on the FM-index, and is now widely used in the field. Details can be found e.g., in [7].

A related query type concerns displaying the text around each pattern occurence. More generally, we want to display a text substring $T[l \ldots r]$ of length $L = r - l + 1$. Again, we make use of a known technique, on the overall obtaining the following time complexities [7]: $O((H_0+1)(L+\frac{1}{\varepsilon}\log n))$ in the average case, and $O(L \log \sigma + (H_0 + 1)\frac{1}{\varepsilon} \log n)$ in the worst case.

6 *K*-ary Huffman

The purpose of the idea of compressing the text before constructing the index is to remove the sharp dependence of the alphabet size of the FM index. This compression is done using a binary alphabet. In general, we can use Huffman over a coding alphabet of $k > 2$ symbols and use $\lceil \log k \rceil$ bits to represent each symbol. We call this generalization the k-ary FM-Huffman. Varying the value of k yields interesting time/space tradeoffs. We use only powers of 2 for k values, so each symbol can be represented without wasting space.

The space usage varies in different aspects. Array B increases its size since the compression ratio gets worse. B has length $n' < (H_0^{(k)} + 1)n$ symbols, where $H_0^{(k)}$ is the zero order entropy of the text computed using base k logarithm, that is, $H_0^{(k)} = -\sum_{i=1}^{\sigma} \frac{n_i}{n} \log_k\left(\frac{n_i}{n}\right) = H_0/\log_2 k$. Therefore, the size of B is bounded by $n' \log k = (H_0 + \log k)n$ bits. The size of Bh is reduced since it needs one bit per symbol, and hence its size is n'. The total space used by these structures is then $n'(1 + \log k) < n(H_0^{(k)} + 1)(1 + \log k)$, which is not larger than the space requirement of the binary version, $2n(H_0 + 1)$, for $1 \leq \log k \leq H_0$.

The *rank* structures also change their size. The *rank* structures for Bh are computed in the same way of the binary version, and therefore they reduce their size, using $o(H_0^{(k)}n)$ bits. For B, we no longer can use the *rank* function to simulate Occ. Instead, we need to calculate the occurrences of each of the k symbols in B. For this sake, we precalculate sublinear structures for each of the symbols, including k tables that count the occurrences of each symbol in a chunk of $b = \lceil \log_k(n)/2 \rceil$ symbols. Hence, we need $o(kH_0^{(k)}n)$ bits for this structures. In total, we need $n(H_0^{(k)} + 1)(1 + \log k) + o(H_0^{(k)}n(k + 1))$ bits.

Regarding the time complexities, the pattern has length $< m(H_0^{(k)} + 1)$ symbols, so this is the search complexity, which is reduced as we increase k. For reporting queries and displaying text, we need the same additional structures TS, ST and S that for the binary version. The k-ary version can report the position of an occurrence in $O\left(\frac{1}{\epsilon}(H_0^{(k)} + 1) \log n\right)$ time, which is the maximum distance between two sampled positions. Similarly, the time to display a substring of length L becomes $O((H_0^{(k)} + 1)(L + \frac{1}{\epsilon} \log n))$ on average and $O(L \log \sigma + (H_0^{(k)} + 1)\frac{1}{\epsilon} \log n)$ in the worst case.

7 Kautz-Zeckendorf Coding

The condition for getting rid of the Bh array is to have a coding for which the bit stream enables instant synchronization at codeword boundaries. A solution could be based on the representation of integers, first advocated by Kautz [9] for its synchronization properties, which presents each number in a unique form as a sum of Fibonacci numbers. This technique is better known from a work by Zeckendorf [15], therefore we will call it Kautz-Zeckendorf coding.

Consider the Fibonacci sequence $f_1 = 1$, $f_2 = 2$, and $f_{i+2} = f_{i+1} + f_i$. The resulting sequence of *Fibonacci numbers* is 1, 2, 3, 5, 8, 13, ... It is easy to prove by induction that any integer number N can be uniquely decomposed into a sum of Fibonacci numbers, where each number is summed at most once and no two consecutive numbers are used in the decomposition. (If two consecutive numbers f_i and f_i and f_{i+1} appear in the decomposition we can use f_{i+2} instead.) Thus we can represent N as a bit vector, whose i-th bit is set iff the i-th Fibonacci number is used to represent N. No two consecutive bits can be set in this representation because this would mean that we used two consecutive numbers in the decomposition. This can be generalized to k consecutive ones [9]. The recurrence is now $f_i = i$ for $i \leq k$ and $f_{i+k} = f_{i+k-1} + f_{i+k-2} + \ldots + f_{i+1} + f_i$. In this representation we do not permit a sequence of k consecutive numbers in the decomposition, and thus no stream of k 1's appears in the bit vector.

We use this encoding as follows. We sort the source symbols by frequency and then assign the binary encoding of number N to the N-th most frequent symbol. In addition, all the encodings are prepended with a sequence of k 1's followed by one 0. Note that nowhere else in the encoding are there k adjacent 1's.

If, during the LF-mapping, we read a 0 and then k successive 1's from T', we know that we are at a codeword beginning. Thus, Bh is no longer needed. A practical side-effect is also that there is no need for *select* to find the successive matches: they all are in a contiguous range of the matrix rows. All the rest of the operatory remains unchanged.

Let us consider the performance of Kautz-Zeckendorf coding with the two most practical (at least for natural languages) parameters, $k = 2$ and $k = 3$. The regular expressions for all valid codewords in those cases are $110(0|10) * (\varepsilon|1)$ and $1110(0|10|110) * (\varepsilon|1|11)$, respectively. We calculated the average codeword length for the 80 MB English text used in Section 8. Note that all we needed to know for this estimation was the knowledge of zero-order symbol distribution in the text. For $k = 2$ and $k = 3$ the average lengths were 5.696 and 6.420 bits per symbol, respectively. The only component of the index, apart from the B array, is the *rank* structure for B. The fastest *rank* in the new implementation needs 25% of the text size. Taking this figure, we obtain approximately $0.89n$ and $1.00n$ overall space occupancy, respectively. Those results are better than of any other variant of our index, but the price is a longer search time. Note that even less space can be obtained with a *rank* implementation using 10% of the text size [5], for a relatively little slow-down. Other options can be better for other text types, e.g., for DNA using $k = 1$ (actually a unary code) is a better choice.

8 Experimental Results

We implemented our indexes, both the original, the k-ary and the KZ versions, making some practical considerations that differ from the theoretical ones. The main difference is the calculation of *rank* and *Occ*, where we used the solution described in [5], for the older index variants, or the new *rank* implementation described in Section 3. The new indexes will be called FM-KZ1 and FM-KZ2, corresponding to the parameters $k = 1$ and $k = 2$, respectively.

In this section we show experimental results on counting, reporting and displaying queries and compare the efficiency to existing indexes. The indexes used for the experiments were the FM-index implemented by Navarro [12], Sadakane's CSA [14], the RLFM index [10], the SSA index [10] and the LZ index [12]. Other indexes whose implementations are available were not included because they are not comparable to the FM Huffman / FM-KZ index due either to their large space requirement or their high search times .

We considered three types of text for the experiments: 80 MB of English text obtained from the TREC-3 collection [5] (files `WSJ87-89`), 60 MB of DNA and 55 MB of protein sequences, both obtained from the BLAST database of the NCBI[6] (files `month.est_others` and `swissprot` respectively).

Our experiments were run on an Intel(R) Xeon(TM) processor at 3.06 GHz, 2 GB of RAM and 512 KB cache, running Gentoo Linux 2.6.10. We compiled the code with `gcc 3.3.5` using optimization option `-O9`.

Now we show the results regarding the space used by our index and later the results of the experiments divided in query type.

[5] Text Retrieval Conference, http://trec.nist.gov
[6] National Center for Biotechnology Information, http://www.ncbi.nlm.nih.gov

8.1 Space results

For the experiments we considered the binary, the 4-ary, and the KZ versions of our index. It is interesting to know how the space requirement of the Huffman-based index varies according to the parameter k. Table 1 (left) shows the space that the index takes as a fraction of the text for different values of k and the three types of files considered. These values do not include the space required to report positions and display text.

We can see that the space requirements are lowest for $k = 4$. For higher values this space increases, although staying reasonable until $k = 16$. With higher values the spaces are too high for these indexes to be comparable to the rest. It would be interesting to study the time performance to the versions of the index with $k = 8$ and $k = 16$. With $k = 8$ we do not expect an improvement on the query time since $\log k$ is not a power (reasons omitted) of 2 and therefore the computation of *Occ* is slower. The version with $k = 16$ could lead to a reduction in query time, but the access to 4 machine words for the calculation of *Occ* could negatively affect it. It is important to say that this values are only relevant for the English text and proteins, since it does not make sense to use them for DNA.

It is also interesting to see how the space requirement of the index is divided among its different structures. Table 1 (right) shows the space used by each of the structures for the index with $k = 2$ and $k = 4$ for the three types of texts considered.

k	Fraction of text		
	English	DNA	Proteins
2	1,68	0,76	1,45
4	1,52	0,74	1,30
8	1,60	0,91	1,43
16	1,84	—	1,57
32	2,67	—	1,92
64	3,96	—	—

	FM-Huffman $k = 2$			**FM-Huffman** $k = 4$		
Structure	Space [MB]			Space [MB]		
	English	DNA	Proteins	English	DNA	Proteins
B	48.98	16.59	29.27	49.81	18.17	29.60
Bh	48.98	16.59	29.27	24.91	9.09	14,80
$Rank(B)$	18,37	6,22	10,97	37,36	13,63	22,20
$Rank(Bh)$	18,37	6,22	10,97	9,34	3,41	5,55
Total	134,69	45,61	80,48	121,41	44,30	72,15
Text	80,00	60,00	55,53	80,00	60,00	55,53
Fraction	1.68	0.76	1.45	1.52	0.74	1.30

Table 1. On the left, space requirement of our index for different values of k. The value corresponding to the row $k = 8$ for DNA actually corresponds to $k = 5$, since this is the total number of symbols to code in this file. Similarly, the value of row $k = 32$ for the protein sequence corresponds to $k = 24$. On the right, detailed comparison of $k = 2$ versus $k = 4$. We omit the spaces used by the Huffman table, the constant-size tables for *Rank*, and array C, since they are negligible.

For higher values of k the space used by B will increase since the use of more symbols for the Huffman codes increases the resulting space. On the other hand, the size of Bh decreases at a rate of $\log k$ and so do its *rank* structures. However, the space of the *rank* structures of B increases rapidly, as we need k structures for an array that reduces its size at a rate of $\log k$, which is the reason of the large space requirement for high values of k.

Now, let us take a look at the FM-KZ1 and FM-KZ2 space/time behavior. For DNA, the FM-KZ1 is a clear winner: among the fastest and definitely the most succinct, also it is hard to imagine a simpler full-text index (as the encoding is merely the unary code).

On the English text, FM-KZ2 is takes about $1.0n$ space, much less than other indexes from our family, but is also considerably slower, e.g. more than 1.5 times slower than FM Huffman with $k = 4$.

8.2 Counting queries

For the three files, we show the search time as a function of the pattern length, varying from 10 to 100, with a step of 10. For each length we used 1000 patterns taken from random positions of each text. Each search was repeated 1000 times. We obtained an average error of 2.6% with a confidence of 95%. Figure 3 (left) shows the time for counting the occurrences for each index and for the three files considered. As the CSA index needs a parameter to determine its space for this type of queries, we adjusted it so that it would use approximately the same space that the binary FM-Huffman index.

We also show the average search time per character along with the minimum space requirement of each index to count occurrences. Unlike the CSA, the other indexes do not need a parameter to specify their size for counting queries. Therefore, we show a point as the value of the space used by the index and its search time per character. For the CSA index we show a line to resemble the space-time tradeoff for counting queries. The time per character for each pattern length is the search time divided by the value of the length. The time per character shown on the plot is the average of these times for each length. Figure 3 (right) shows the search time per character for each index and for each type of text.

8.3 Reporting queries

We measured the time that each index took to search for a pattern and report the positions of the occurrences found. From the English text and the DNA sequence we took 1000 random patterns of length 10. From the protein sequence we used patterns of length 5. We measured the time per occurrence reported varying the space requirement for every index except the LZ, which has a fixed size. For the CSA we set the two parameters, namely the size of the structures to report and the structures to count, to the same value, since this turns out to be optimal. Our measures have a 2.2% error with 95% confidence. Figure 4 shows the times per occurrence reported for each index as a function of its size.

8.4 Displaying text

We measured the time that each index took to show the first character of a text context around the occurrences found. More precisely, this is the time of searching for a pattern, locating the position of an occurrence and showing one character of the

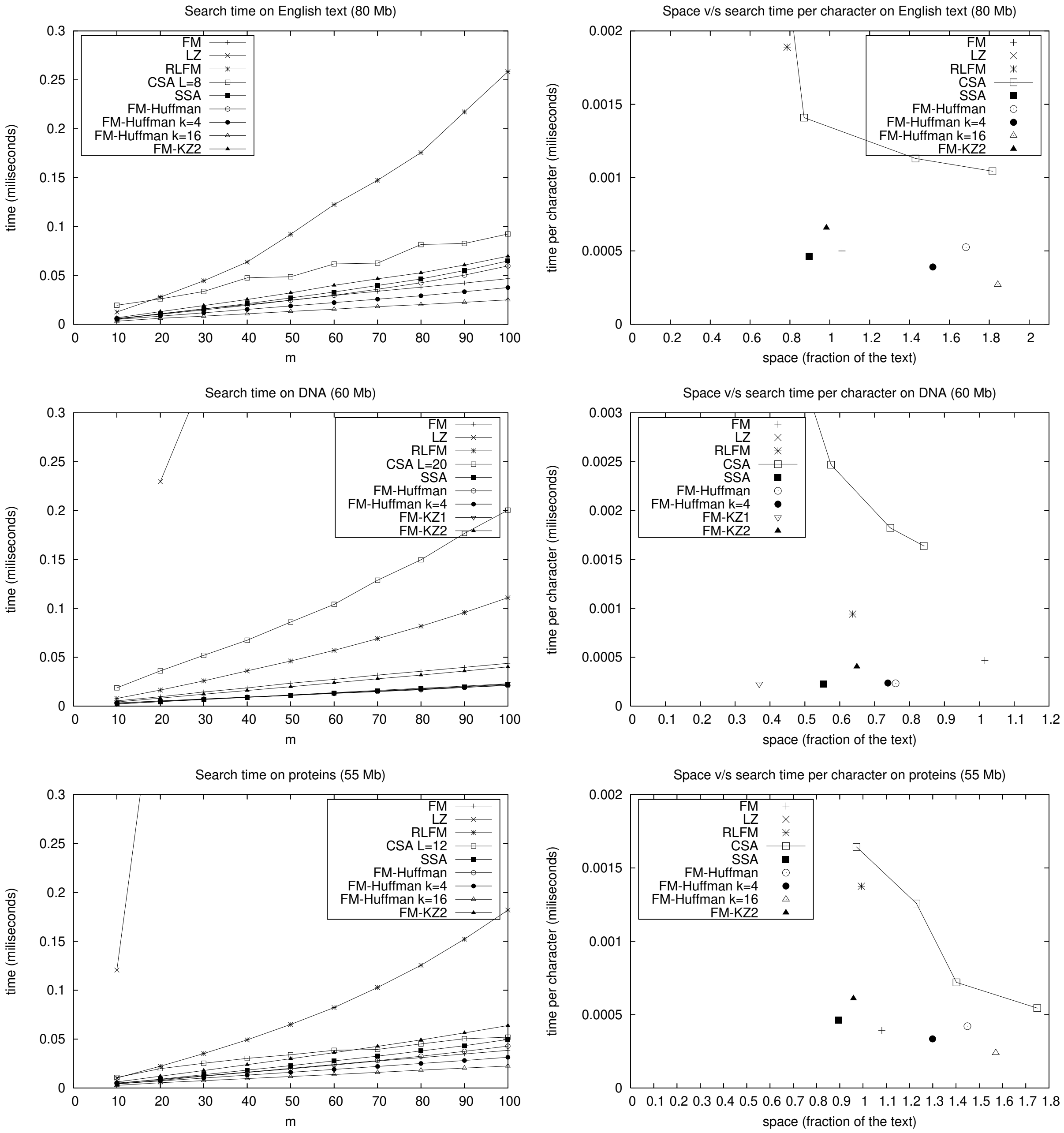


Figure 3. On the left, search time as a function of the pattern length over, English (80 MB), DNA (60 MB), and a proteins (55 MB). The times of the LZ index do not appear on the English text plot, as they range from 0.5 to 4.6 ms. In the DNA plot, the time of the LZ index for $m = 10$ is 2.6. The reason of this increase is the large number of occurrences of these patterns, which influences the counting time for this index. On the right, average search time per character as a function of the size of the index.

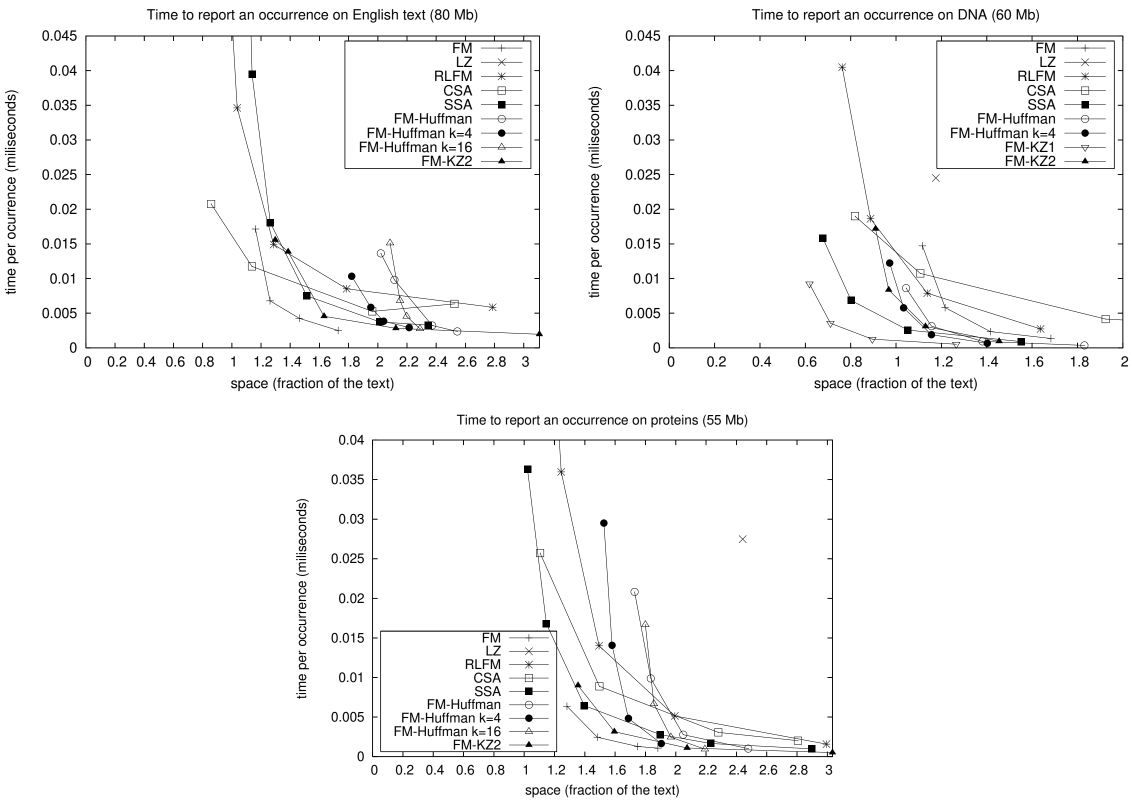


Figure 4. Time to report the positions of the occurrences as a function of the size of the index. We show the results of searching on 80 MB of English text, 60 MB of DNA and finally 55 MB of proteins.

text in the context area of the position located. Usually this character is the one at the position of the occurrence, but it can also be a different close one, depending on each index. We measured this time as a function of the size used by each index. We used the same 1000 patterns used for the reporting experiment, obtaining an average error of 1.6% with 95% confidence. Figure 5 (left) shows the time to display the first character as a function of the space requirement for each index and for each type of text.

In addition, we measured the time to display a context per character displayed. That is, we searched for the 1000 patterns and displayed 100 characters around each of the positions of the occurrences found. We subtracted from this time the time to display the first character and divided it by the amount of characters displayed. For this experiment, we obtained an average error of 6% with 95% confidence. Figure 5 (right) shows this time along with the minimum space required for each index for the counting functionality, since the display time per character does not depend on the size of the index. This is not true for the CSA index, whose time to display per character does depend on its size. For this index we show the time measured as a function of its size.

8.5 Analysis of Results

We can see that our FM-Hufman $k = 4$ and $k = 16$ indexes are among the fastest for counting queries for the three types of files. The binary FM-Huffman index takes the

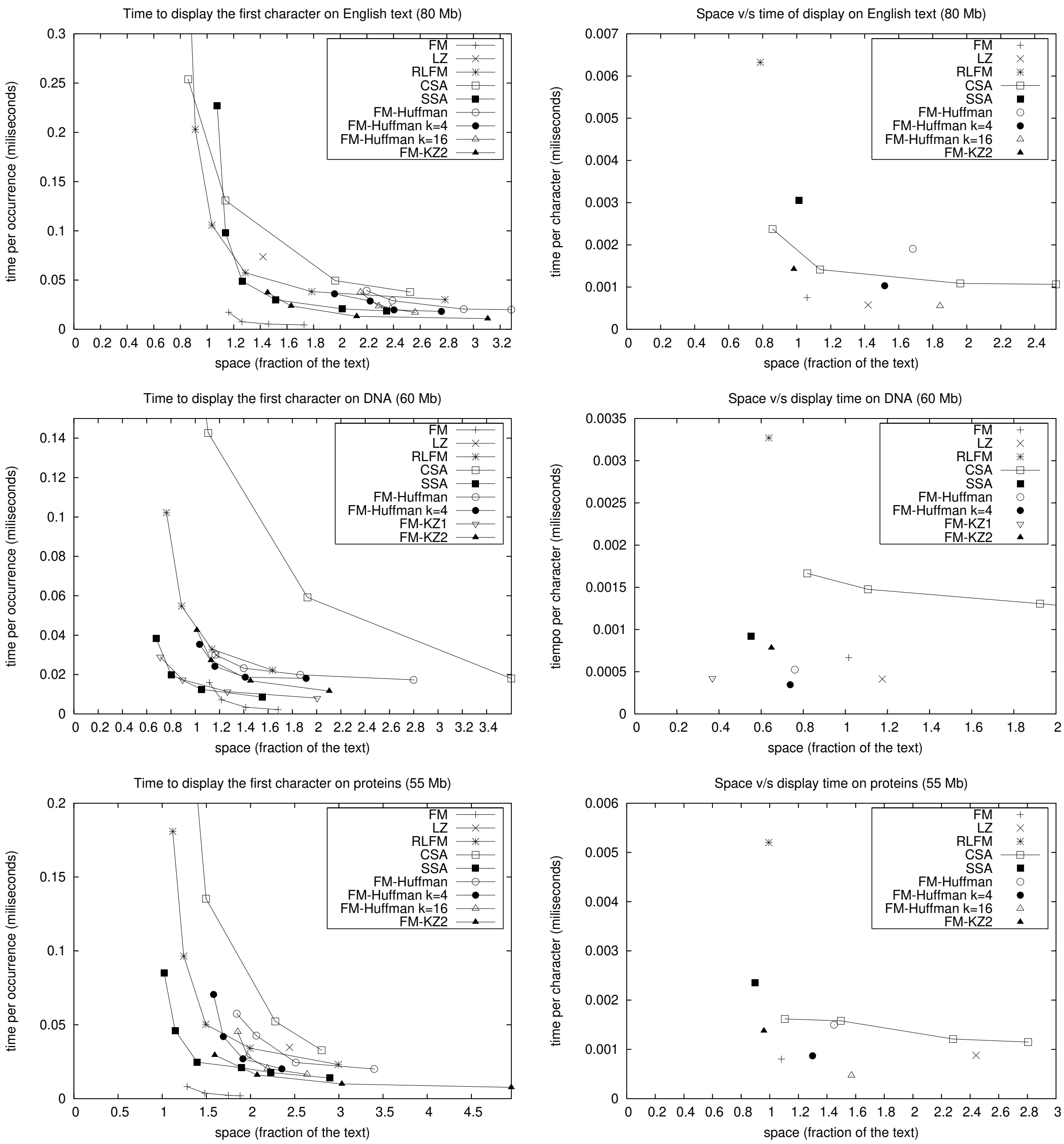


Figure 5. On the left, time to show the first character of a text context around the positions of the occurrences as a function of the size of the index. From top to bottom, we show the results of searching 80 MB of English text, 60 MB of DNA and 55 MB of proteins. In the plot of the DNA sequence, the point corresponding to the LZ index is covered. Its value is: space=1.18, time=0.03. On the right, time per character displayed around an occurrence and space for each index.

same time that $k = 4$ version for DNA and it is a little bit slower that the FM-index for the other two files. As expected, all those versions are faster than CSA, RLFM and LZ, the latter not being competitive for counting queries. Regarding the space usage, the FM-index turns out to be a better tradeoff alternative for the English text and protein sequences, since it uses less space than our index and has low search times. For DNA, all the Huffman based versions of our index are good alternatives, considering their low space requirement and search time.

Still, the new player, FM-KZ index, is a particularly good choice for DNA. It is way ahead of the competition in the space use, while belonging to the fastest. At the same time its simplicity is striking.

Considering both speed and space use, for the English text and the proteins, the SSA index is the best choice, still, our variants come close, especially for proteins.

For reporting queries, our index loses to the FM-index for English and proteins, mainly because of its large space requirement. Also, it only surpasses the RLFM and CSA for large space usages. For DNA, however, our index, with the two versions, is better than the FM-index. This reduction in space is due to the low zero-order entropy of the DNA, which makes our index compact and fast.

Regarding the time for displaying the first character, the FM-index is faster than our index. Again, our index takes more space than the other indexes to get competitive time for English and proteins, and reduces its space for DNA. Regarding display time per character, our index with $k = 4$ is the fastest for DNA with a low space requirement, becoming an interesting alternative for this type of query.

The version of our index with $k = 4$ improved both the space and time with respect to the binary version and it became a very good alternative for counting and reporting queries, especially for DNA, due to the low zero-order entropy of this text.

9 Conclusions

We have focused in this paper on a practical data structure inspired by the FM-index [3], which removes its sharp dependence on the alphabet size σ. Our key idea is to encode the text with the Kautz-Zeckendorf coding, offering instant synchronization at codeword boundaries (a property missing in Huffman coding, thus implying a significant space penalty in FM indexes), at still being quite succinct. While not competitive to the best succinct indexes in theory, our solutions fare well in practice, and are simpler conceptually and easier to implement than the other structures.

Acknowledgements

This work was partially funded by Fondecyt Grant-1-050493, Chile (Gonzalo Navarro).

References

1. M. Burrows and D. Wheeler: *A block sorting lossless data compression algorithm*, Tech. Rep. 124, Digital Equipment Corporation, 1994.
2. D. Clark: *Compact Pat Trees*, PhD thesis, University of Waterloo, Canada, 1996.
3. P. Ferragina and G. Manzini: *Opportunistic data structures with applications*, in Proc. 41st IEEE Symposium on Foundations of Computer Science (FOCS), 2000, pp. 390–398.
4. P. Ferragina and G. Manzini: *An experimental study of an opportunistic index*, in Proc. 12th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 2001, pp. 269–278.

5. R. González, S. Grabowski, V. Mäkinen, and G. Navarro: *Practical implementation of rank and select queries*, in Poster Proceedings Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA'05), Greece, 2005, CTI Press and Ellinika Grammata, pp. 27–38.
6. S. Grabowski, V. Mäkinen, and G. Navarro: *First Huffman, then Burrows-Wheeler: An alphabet-independent FM-index*, in Proceedings of the 11 Symposium on String Processing and Information Retrieval (SPIRE'04), LNCS v. 3246, 2004, pp. 210–211.
7. S. Grabowski, V. Mäkinen, G. Navarro, and A. Salinger: *A simple alphabet-independent FM-index*, in Proceedings of the 10th Prague Stringology Conference (PSC'05), 2005, pp. 230–244.
8. G. Jacobson: *Space-efficient static trees and graphs*, in Proc. 30th IEEE Symposium on Foundations of Computer Science (FOCS), 1989, pp. 549–554.
9. W. H. Kautz: *Fibonacci codes for synchronization control.* IEEE Transactions on Information Theory, 11 1965, pp. 242–292.
10. V. Mäkinen and G. Navarro: *Succinct suffix arrays based on run-length encoding.* Nordic Journal of Computing, 12(1) 2005, pp. 40–66.
11. I. Munro: *Tables*, in Proc. 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS), LNCS v. 1180, 1996, pp. 37–42.
12. G. Navarro: *Indexing text using the Ziv-Lempel trie.* Journal of Discrete Algorithms, 2(1) 2004, pp. 87–114.
13. G. Navarro and V. Mäkinen: *Compressed full-text indexes (2nd version)*, Tech. Rep. TR/DCC-2006-6, Department of Computer Science, University of Chile, Chile, Apr. 2006.
14. K. Sadakane: *Compressed text databases with efficient query algorithms based on the compressed suffix array*, in Proc. 11th International Symposium on Algorithms and Computation (ISAAC), LNCS v. 1969, 2000, pp. 410–421.
15. E. Zeckendorf: *Représentation des nombres naturels par une somme de nombres de Fibonacci ou de nombres Lucas.* Bull. Soc. Roy. Sci. Liège, 41 1972, pp. 179–182.

Author Index